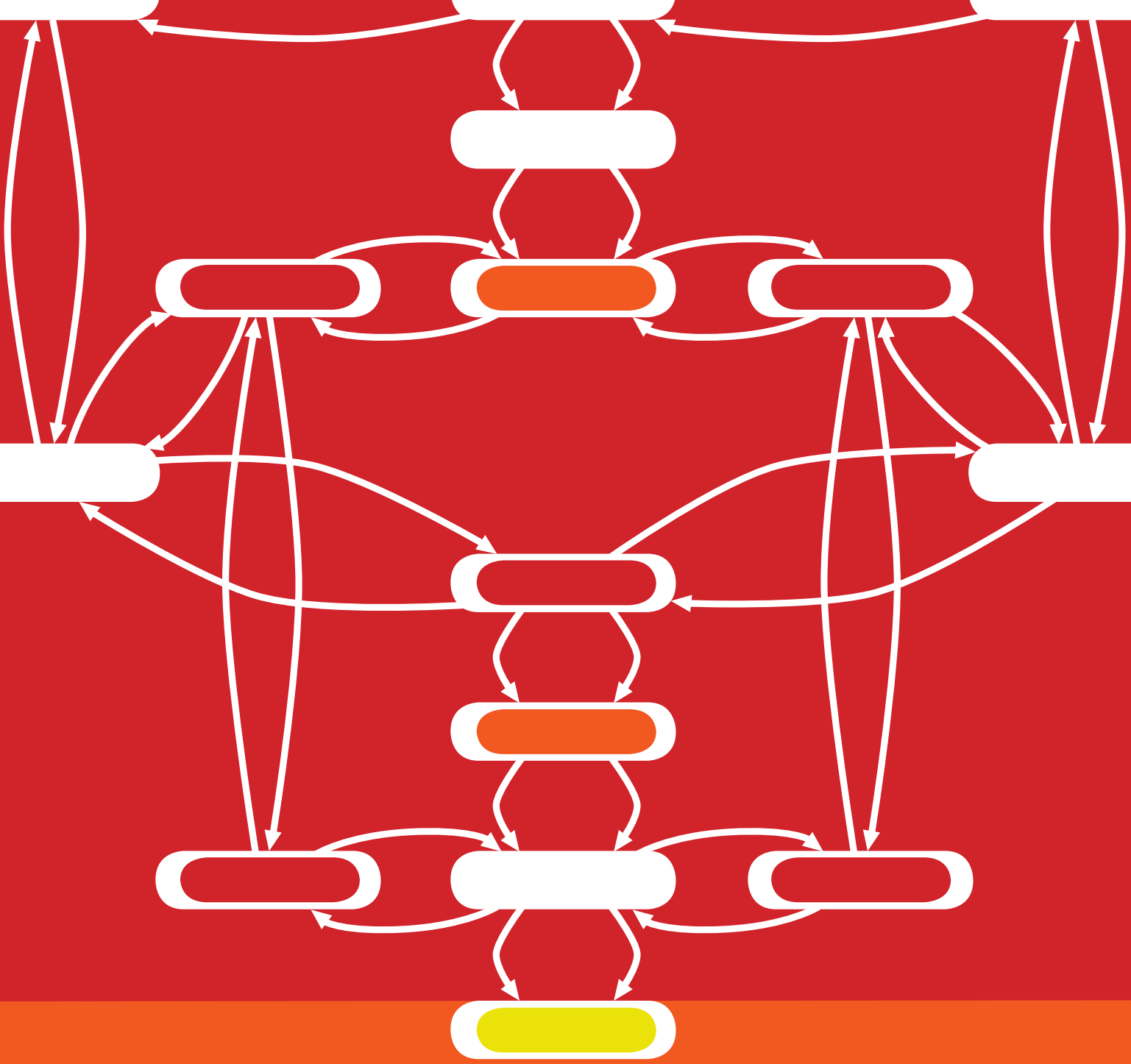


Principles of Model Checking

Christel Baier and Joost-Pieter Katoen

Principles of Model Checking

Principles of Model Checking

Christel Baier
Joost-Pieter Katoen

The MIT Press
Cambridge, Massachusetts
London, England

©Massachusetts Institute of Technology

All rights reserved. No part of this book may be reproduced in any form by any electronic of mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

MIT Press books may be purchased at special quantity discounts for business or sales promotional use. For information, please email special_sales@mitpress.mit.edu or write to Special Sales Department, The MIT Press, 55 Hayward Street, Cambridge, MA 02142.

This book was set in Aachen and Dresden by Christel Baier and Joost-Pieter Katoen. Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Baier, Christel.
Principles of model checking / Christel Baier and Joost-Pieter Katoen ; foreword by Kim Guldstrand Larsen.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-262-02649-9 (hardcover : alk. paper) 1. Computer systems–Verification. 2. Computer software–Verification. I.
Katoen, Joost-Pieter. II. Title.
QA76.76.V47B35 2008
004.2'4–dc22

2007037603

10 9 8 7 6 5 4 3 2 1

To Michael, Gerda, Inge, and Karl

To Erna, Fons, Joost, and Tom

Contents

Foreword

Society is increasingly dependent on dedicated computer and software systems to assist us in almost every aspect of daily life. Often we are not even aware that computers and software are involved. Several control functions in modern cars are based on embedded software solutions, e.g., braking, airbags, cruise control, and fuel injection. Mobile phones, communication systems, medical devices, audio and video systems, and consumer electronics in general are containing vast amounts of software. Also transport, production, and control systems are increasingly applying embedded software solutions to gain flexibility and cost-efficiency.

A common pattern is the constantly increasing complexity of systems, a trend which is accelerated by the adaptation of wired and wireless networked solutions: in a modern car the control functions are distributed over several processing units communicating over dedicated networks and buses. Yet computer- and software-based solutions are becoming ubiquitous and are to be found in several safety-critical systems. Therefore a main challenge for the field of computer science is to provide formalisms, techniques, and tools that will enable the efficient design of correct and well-functioning systems despite their complexity.

Over the last two decades or so a very attractive approach toward the correctness of computer-based control systems is that of model checking. Model checking is a formal verification technique which allows for desired behavioral properties of a given system to be verified on the basis of a suitable model of the system through systematic inspection of all states of the model. The attractiveness of model checking comes from the fact that it is completely automatic – i.e., the learning curve for a user is very gentle – and that it offers counterexamples in case a model fails to satisfy a property serving as indispensable debugging information. On top of this, the performance of model-checking tools has long since proved mature as witnessed by a large number of successful industrial applications.

It is my pleasure to recommend the excellent book *Principles of Model Checking* by Christel Baier and Joost-Pieter Katoen as the definitive textbook on model checking, providing both a comprehensive and a comprehensible account of this important topic. The book contains detailed and complete descriptions of first principles of classical Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) model checking. Also, state-of-the art methods for coping with state-space explosion, including symbolic model checking, abstraction and minimization techniques, and partial order reduction, are fully accounted for. The book also covers model checking of real-time and probabilistic systems, important new directions for model checking in which the authors, being two of the most industrious and creative researchers of today, are playing a central role.

The exceptional pedagogical style of the authors provides careful explanations of constructions and proofs, plus numerous examples and exercises of a theoretical, practical and tool-oriented nature. The book will therefore be the ideal choice as a textbook for both graduate and advanced undergraduate students, as well as for self-study, and should definitely be on the bookshelf of any researcher interested in the topic.

Kim Guldstrand Larsen
Professor in Computer Science
Aalborg University, Denmark
May 2007

Preface

It is fair to state, that in this digital era
correct systems for information processing
are more valuable than gold.

(H. Barendregt. The quest for correctness.
In: Images of SMC Research 1996, pages 39–58, 1996.)

This book is on model checking, a prominent formal verification technique for assessing functional properties of information and communication systems. Model checking requires a model of the system under consideration and a desired property and systematically checks whether or not the given model satisfies this property. Typical properties that can be checked are deadlock freedom, invariants, and request-response properties. Model checking is an automated technique to check the absence of errors (i.e., property violations) and alternatively can be considered as an intelligent and effective debugging technique. It is a general approach and is applied in areas like hardware verification and software engineering. Due to unremitting improvements of underlying algorithms and data structures together with hardware technology improvements, model-checking techniques that two decades ago only worked for simple examples are nowadays applicable to more realistic designs. It is fair to say that in the last two decades model checking has developed as a mature and heavily used verification and debugging technique.

Aims and Scope

This book attempts to introduce model checking from first principles, so to speak, and is intended as a textbook for bachelor and master students, as well as an introductory book for researchers working in other areas of computer science or related fields. The reader is introduced to the material by means of an extensive set of examples, most of which are examples running throughout several chapters. The book provides a complete set of basic results together with all detailed proofs. Each chapter is concluded by a summary,

bibliographic notes, and a series of exercises, of both a theoretical and of a practical nature (i.e., experimenting with actual model checkers).

Prerequisites

The concepts of model checking have their roots in mathematical foundations such as propositional logic, automata theory and formal languages, data structures, and graph algorithms. It is expected that readers are familiar with the basics of these topics when starting with our book, although an appendix is provided that summarizes the essentials. Knowledge on complexity theory is required for the theoretical complexity considerations of the various model-checking algorithms.

Content

This book is divided into ten chapters. Chapter 1 motivates and introduces model checking. Chapter 2 presents transition systems as a model for software and hardware systems. Chapter 3 introduces a classification of linear-time properties into safety and liveness, and presents the notion of fairness. Automata-based algorithms for checking (regular) safety and ω-regular properties are presented in Chapter 4. Chapter 5 deals with Linear Temporal Logic (LTL) and shows how the algorithms of Chapter 4 can be used for LTL model checking. Chapter 6 introduces the branching-time temporal logic Computation Tree Logic (CTL), compares this to LTL, and shows how to perform CTL model checking, both explicitly and symbolically. Chapter 7 deals with abstraction mechanisms that are based on trace, bisimulation, and simulation relations. Chapter 8 treats partial-order reduction for LTL and CTL. Chapter 9 is focused on real-time properties and timed automata, and the monograph is concluded with a chapter on the verification of probabilistic models. The appendix summarizes basic results on propositional logic, graphs, language, and complexity theory.

How to Use This Book

A natural plan for an introductory course into model checking that lasts one semester (two lectures a week) comprises Chapters 1 through 6. A follow-up course of about a semester could cover Chapters 7 through 10, after a short refresher on LTL and CTL model checking.

Acknowledgments

This monograph has been developed and extended during the last five years. The following colleagues supported us by using (sometimes very) preliminary versions of this monograph: Luca Aceto (Aalborg, Denmark and Reykjavik, Iceland), Henrik Reif Andersen (Copenhagen, Denmark), Dragan Boshnacki (Eindhoven, The Netherlands), Franck van Breughel (Ottawa, Canada), Josée Desharnais (Quebec, Canada), Susanna Donatelli (Turin, Italy), Stefania Gnesi (Pisa, Italy), Michael R. Hansen (Lyngby, Denmark), Holger Hermanns (Saarbrücken, Germany), Yakov Kesselman (Chicago, USA), Martin Lange (Aarhus, Denmark), Kim G. Larsen (Aalborg, Denmark), Mieke Massink (Pisa, Italy), Mogens Nielsen (Aarhus, Denmark), Albert Nymeyer (Sydney, Australia), Andreas Podelski (Freiburg, Germany), Theo C. Ruys (Twente, The Netherlands), Thomas Schwentick (Dortmund, Germany), Wolfgang Thomas (Aachen, Germany), Julie Vachon (Montreal, Canada), and Glynn Winskel (Cambridge, UK). Many of you provided us with very helpful feedback that helped us to improve the lecture notes.

Henrik Bohnenkamp, Tobias Blechmann, Frank Ciesinski, Marcus Grösser, Tingting Han, Joachim Klein, Sascha Klüppelholz, Miriam Nasfi, Martin Neuhäusser, and Ivan S. Zapreev provided us with many detailed comments, and provided several exercises. Yen Cao is kindly thanked for drawing a part of the figures and Ulrich Schmidt-Görtz for his assistance with the bibliography.

Many people have suggested improvements and pointed out mistakes. We thank everyone for providing us with helpful comments.

Finally, we thank all our students in Aachen, Bonn, Dresden, and Enschede for their feedback and comments.

Christel Baier

Joost-Pieter Katoen

Chapter 1

System Verification

Our reliance on the functioning of ICT systems (Information and Communication Technology) is growing rapidly. These systems are becoming more and more complex and are massively encroaching on daily life via the Internet and all kinds of embedded systems such as smart cards, hand-held computers, mobile phones, and high-end television sets. In 1995 it was estimated that we are confronted with about 25 ICT devices on a daily basis. Services like electronic banking and teleshopping have become reality. The daily cash flow via the Internet is about 10^{12} million US dollars. Roughly 20% of the product development costs of modern transportation devices such as cars, high-speed trains, and airplanes is devoted to information processing systems. ICT systems are universal and omnipresent. They control the stock exchange market, form the heart of telephone switches, are crucial to Internet technology, and are vital for several kinds of medical systems. Our reliance on embedded systems makes their reliable operation of large social importance. Besides offering a good performance in terms like response times and processing capacity, the absence of annoying errors is one of the major quality indications.

It is all about money. We are annoyed when our mobile phone malfunctions, or when our video recorder reacts unexpectedly and wrongly to our issued commands. These software and hardware errors do not threaten our lives, but may have substantial financial consequences for the manufacturer. Correct ICT systems are essential for the survival of a company. Dramatic examples are known. The bug in Intel's Pentium II floating-point division unit in the early nineties caused a loss of about 475 million US dollars to replace faulty processors, and severely damaged Intel's reputation as a reliable chip manufacturer. The software error in a baggage handling system postponed the opening of Denver's airport for 9 months, at a loss of 1.1 million US dollar per day. Twenty-four hours of failure of

Figure 1.1: The Ariane-5 launch on June 4, 1996; it crashed 36 seconds after the launch due to a conversion of a 64-bit floating point into a 16-bit integer value.

the worldwide online ticket reservation system of a large airplane company will cause its bankruptcy because of missed orders.

It is all about safety: errors can be catastrophic too. The fatal defects in the control software of the Ariane-5 missile (Figure 1.1), the Mars Pathfinder, and the airplanes of the Airbus family led to headlines in newspapers all over the world and are notorious by now. Similar software is used for the process control of safety-critical systems such as chemical plants, nuclear power plants, traffic control and alert systems, and storm surge barriers. Clearly, bugs in such software can have disastrous consequences. For example, a software flaw in the control part of the radiation therapy machine Therac-25 caused the death of six cancer patients between 1985 and 1987 as they were exposed to an overdose of radiation.

The increasing reliance of critical applications on information processing leads us to state:

The reliability of ICT systems is a key issue in the system design process.

The magnitude of ICT systems, as well as their complexity, grows apace. ICT systems are no longer standalone, but are typically embedded in a larger context, connecting and interacting with several other components and systems. They thus become much more vulnerable to errors – the number of defects grows exponentially with the number of interacting system components. In particular, phenomena such as concurrency and nondeterminism that are central to modeling interacting systems turn out to be very hard to handle with standard techniques. Their growing complexity, together with the pressure to drastically reduce system development time ("time-to-market"), makes the delivery of low-defect ICT systems an enormously challenging and complex activity.

Hard- and Software Verification

System verification techniques are being applied to the design of ICT systems in a more reliable way. Briefly, system verification is used to establish that the design or product under consideration possesses certain properties. The properties to be validated can be quite elementary, e.g., a system should never be able to reach a situation in which no progress can be made (a deadlock scenario), and are mostly obtained from the *system's specification*. This specification prescribes what the system has to do and what not, and thus constitutes the basis for any verification activity. A defect is found once the system does not fulfill one of the specification's properties. The system is considered to be "correct" whenever it satisfies all properties obtained from its specification. So correctness is always relative to a specification, and is not an absolute property of a system. A schematic view of verification is depicted in Figure 1.2.

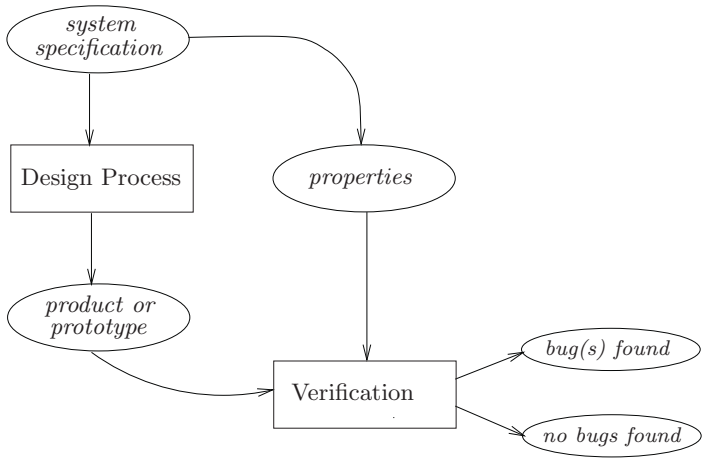


Figure 1.2: Schematic view of an a posteriori system verification.

This book deals with a verification technique called model checking that starts from a formal system specification. Before introducing this technique and discussing the role of formal specifications, we briefly review alternative software and hardware verification techniques.

Software Verification Peer reviewing and testing are the major software verification techniques used in practice.

A *peer review* amounts to a software inspection carried out by a team of software engineers that preferably has not been involved in the development of the software under review. The

uncompiled code is not executed, but analyzed completely statically. Empirical studies indicate that peer review provides an effective technique that catches between 31 % and 93 % of the defects with a median around 60%. While mostly applied in a rather ad hoc manner, more dedicated types of peer review procedures, e.g., those that are focused at specific error-detection goals, are even more effective. Despite its almost complete manual nature, peer review is thus a rather useful technique. It is therefore not surprising that some form of peer review is used in almost 80% of all software engineering projects. Due to its static nature, experience has shown that subtle errors such as concurrency and algorithm defects are hard to catch using peer review.

Software testing constitutes a significant part of any software engineering project. Between 30% and 50% of the total software project costs are devoted to testing. As opposed to peer review, which analyzes code statically without executing it, testing is a dynamic technique that actually runs the software. Testing takes the piece of software under consideration and provides its compiled code with inputs, called tests. Correctness is thus determined by forcing the software to traverse a set of execution paths, sequences of code statements representing a run of the software. Based on the observations during test execution, the actual output of the software is compared to the output as documented in the system specification. Although test generation and test execution can partly be automated, the comparison is usually performed by human beings. The main advantage of testing is that it can be applied to all sorts of software, ranging from application software (e.g., e-business software) to compilers and operating systems. As exhaustive testing of all execution paths is practically infeasible; in practice only a small subset of these paths is treated. Testing can thus never be complete. That is to say, testing can only show the presence of errors, not their absence. Another problem with testing is to determine when to stop. Practically, it is hard, and mostly impossible, to indicate the intensity of testing to reach a certain defect density – the fraction of defects per number of uncommented code lines.

Studies have provided evidence that peer review and testing catch different classes of defects at different stages in the development cycle. They are therefore often used together. To increase the reliability of software, these software verification approaches are complemented with software process improvement techniques, structured design and specification methods (such as the Unified Modeling Language), and the use of version and configuration management control systems. Formal techniques are used, in one form or another, in about 10 % to 15% of all software projects. These techniques are discussed later in this chapter.

Catching software errors: the sooner the better. It is of great importance to locate software bugs. The slogan is: the sooner the better. The costs of repairing a software flaw during maintenance are roughly 500 times higher than a fix in an early design phase (see Figure 1.3). System verification should thus take place early stage in the design process.

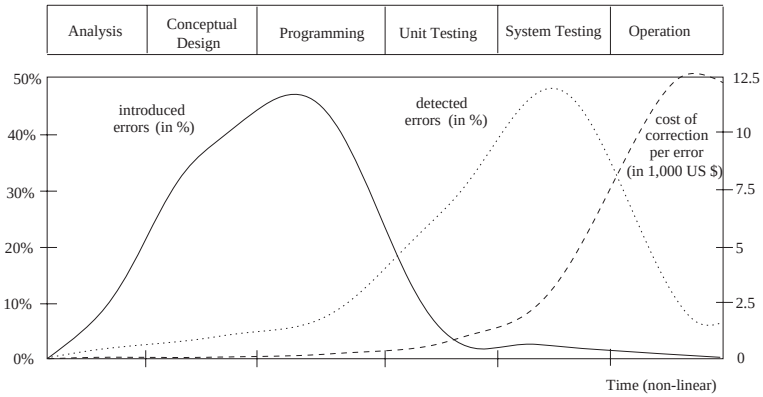


Figure 1.3: Software lifecycle and error introduction, detection, and repair costs [275].

About 50% of all defects are introduced during programming, the phase in which actual coding takes place. Whereas just 15% of all errors are detected in the initial design stages, most errors are found during testing. At the start of unit testing, which is oriented to discovering defects in the individual software modules that make up the system, a defect density of about 20 defects per 1000 lines of (uncommented) code is typical. This has been reduced to about 6 defects per 1000 code lines at the start of system testing, where a collection of such modules that constitutes a real product is tested. On launching a new software release, the typical accepted software defect density is about one defect per 1000 lines of code lines[1].

Errors are typically concentrated in a few software modules – about half of the modules are defect free, and about 80% of the defects arise in a small fraction (about 20%) of the modules – and often occur when interfacing modules. The repair of errors that are detected prior to testing can be done rather economically. The repair cost significantly increases from about $ 1000 (per error repair) in unit testing to a maximum of about $ 12,500 when the defect is demonstrated during system operation only. It is of vital importance to seek techniques that find defects as early as possible in the software design process: the costs to repair them are substantially lower, and their influence on the rest of the design is less substantial.

Hardware Verification Preventing errors in hardware design is vital. Hardware is subject to high fabrication costs; fixing defects after delivery to customers is difficult, and quality expectations are high. Whereas software defects can be repaired by providing

[1] For some products this is much higher, though. Microsoft has acknowledged that Windows 95 contained at least 5000 defects. Despite the fact that users were daily confronted with anomalous behavior, Windows 95 was very successful.

users with patches or updates – nowadays users even tend to anticipate and accept this – hardware bug fixes after delivery to customers are very difficult and mostly require refabrication and redistribution. This has immense economic consequences. The replacement of the faulty Pentium II processors caused Intel a loss of about $ 475 million. Moore's law – the number of logical gates in a circuit doubles every 18 months – has proven to be true in practice and is a major obstacle to producing correct hardware. Empirical studies have indicated that more than 50% of all ASICs (Application-Specific Integrated Circuits) do not work properly after initial design and fabrication. It is not surprising that chip manufacturers invest a lot in getting their designs right. Hardware verification is a well-established part of the design process. The design effort in a typical hardware design amounts to only 27% of the total time spent on the chip; the rest is devoted to error detection and prevention.

Hardware verification techniques. Emulation, simulation, and structural analysis are the major techniques used in hardware verification.

Structural analysis comprises several specific techniques such as synthesis, timing analysis, and equivalence checking that are not described in further detail here.

Emulation is a kind of testing. A reconfigurable generic hardware system (the emulator) is configured such that it behaves like the circuit under consideration and is then extensively tested. As with software testing, emulation amounts to providing a set of stimuli to the circuit and comparing the generated output with the expected output as laid down in the chip specification. To fully test the circuit, all possible input combinations in every possible system state should be examined. This is impractical and the number of tests needs to be reduced significantly, yielding potential undiscovered errors.

With *simulation*, a model of the circuit at hand is constructed and simulated. Models are typically provided using hardware description languages such as Verilog or VHDL that are both standardized by IEEE. Based on stimuli, execution paths of the chip model are examined using a simulator. These stimuli may be provided by a user, or by automated means such as a random generator. A mismatch between the simulator's output and the output described in the specification determines the presence of errors. Simulation is like testing, but is applied to models. It suffers from the same limitations, though: the number of scenarios to be checked in a model to get full confidence goes beyond any reasonable subset of scenarios that can be examined in practice.

Simulation is the most popular hardware verification technique and is used in various design stages, e.g., at register-transfer level, gate and transistor level. Besides these error detection techniques, *hardware testing* is needed to find fabrication faults resulting from layout defects in the fabrication process.

1.1 Model Checking

In software and hardware design of complex systems, more time and effort are spent on verification than on construction. Techniques are sought to reduce and ease the verification efforts while increasing their coverage. Formal methods offer a large potential to obtain an early integration of verification in the design process, to provide more effective verification techniques, and to reduce the verification time.

Let us first briefly discuss the role of formal methods. To put it in a nutshell, formal methods can be considered as "the applied mathematics for modeling and analyzing ICT systems". Their aim is to establish system correctness with mathematical rigor. Their great potential has led to an increasing use by engineers of formal methods for the verification of complex software and hardware systems. Besides, formal methods are one of the "highly recommended" verification techniques for software development of safety-critical systems according to, e.g., the best practices standard of the IEC (International Electrotechnical Commission) and standards of the ESA (European Space Agency). The resulting report of an investigation by the FAA (Federal Aviation Authority) and NASA (National Aeronautics and Space Administration) about the use of formal methods concludes that

> *Formal methods should be part of the education of every computer scientist and software engineer, just as the appropriate branch of applied maths is a necessary part of the education of all other engineers.*

During the last two decades, research in formal methods has led to the development of some very promising verification techniques that facilitate the early detection of defects. These techniques are accompanied by powerful software tools that can be used to automate various verification steps. Investigations have shown that formal verification procedures would have revealed the exposed defects in, e.g., the Ariane-5 missile, Mars Pathfinder, Intel's Pentium II processor, and the Therac-25 therapy radiation machine.

Model-based verification techniques are based on models describing the possible system behavior in a mathematically precise and unambiguous manner. It turns out that – prior to any form of verification – the accurate modeling of systems often leads to the discovery of incompleteness, ambiguities, and inconsistencies in informal system specifications. Such problems are usually only discovered at a much later stage of the design. The system models are accompanied by algorithms that systematically explore all states of the system model. This provides the basis for a whole range of verification techniques ranging from an exhaustive exploration (model checking) to experiments with a restrictive set of scenarios in the model (simulation), or in reality (testing). Due to unremitting improvements of un-

derlying algorithms and data structures, together with the availability of faster computers and larger computer memories, model-based techniques that a decade ago only worked for very simple examples are nowadays applicable to realistic designs. As the startingpoint of these techniques is a model of the system under consideration, we have as a given fact that

> *Any verification using model-based techniques is only as good as the model of the system.*

Model checking is a verification technique that explores all possible system states in a brute-force manner. Similar to a computer chess program that checks possible moves, a model checker, the software tool that performs the model checking, examines all possible system scenarios in a systematic manner. In this way, it can be shown that a given system model truly satisfies a certain property. It is a real challenge to examine the largest possible state spaces that can be treated with current means, i.e., processors and memories. State-of-the-art model checkers can handle state spaces of about 10^8 to 10^9 states with explicit state-space enumeration. Using clever algorithms and tailored data structures, larger state spaces (10^{20} up to even 10^{476} states) can be handled for specific problems. Even the subtle errors that remain undiscovered using emulation, testing and simulation can potentially be revealed using model checking.

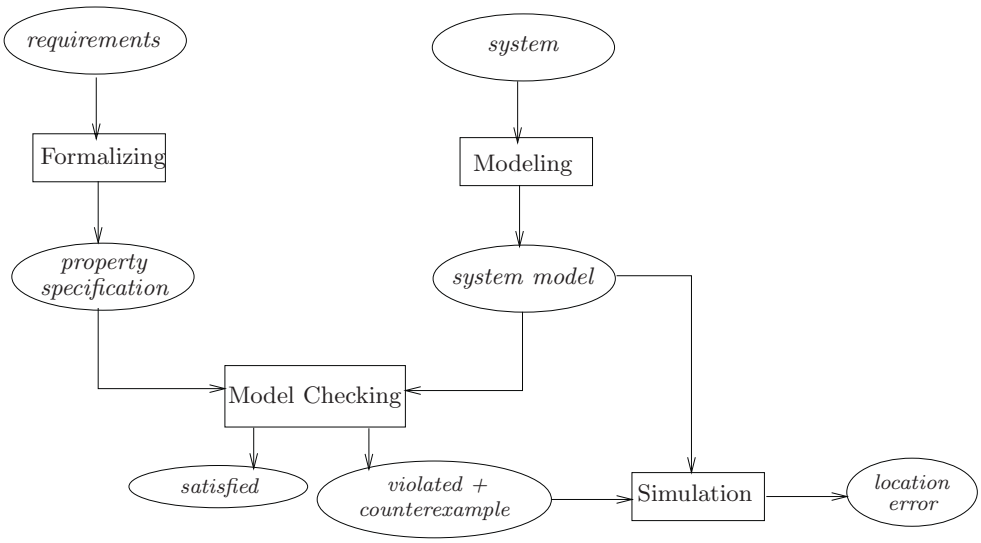


Figure 1.4: Schematic view of the model-checking approach.

Typical properties that can be checked using model checking are of a qualitative nature: Is the generated result OK?, Can the system reach a deadlock situation, e.g., when two

concurrent programs are waiting for each other and thus halting the entire system? But also timing properties can be checked: Can a deadlock occur within 1 hour after a system reset?, or, Is a response always received within 8 minutes? Model checking requires a precise and unambiguous statement of the properties to be examined. As with making an accurate system model, this step often leads to the discovery of several ambiguities and inconsistencies in the informal documentation. For instance, the formalization of all system properties for a subset of the ISDN user part protocol revealed that 55% (!) of the original, informal system requirements were inconsistent.

The system model is usually automatically generated from a model description that is specified in some appropriate dialect of programming languages like C or Java or hardware description languages such as Verilog or VHDL. Note that the property specification prescribes *what* the system should do, and what it should not do, whereas the model description addresses *how* the system behaves. The model checker examines all relevant system states to check whether they satisfy the desired property. If a state is encountered that violates the property under consideration, the model checker provides a counterexample that indicates how the model could reach the undesired state. The counterexample describes an execution path that leads from the initial system state to a state that violates the property being verified. With the help of a simulator, the user can replay the violating scenario, in this way obtaining useful debugging information, and adapt the model (or the property) accordingly (see Figure 1.4).

Model checking has been successfully applied to several ICT systems and their applications. For instance, deadlocks have been detected in online airline reservation systems, modern e-commerce protocols have been verified, and several studies of international IEEE standards for in-house communication of domestic appliances have led to significant improvements of the system specifications. Five previously undiscovered errors were identified in an execution module of the Deep Space 1 spacecraft controller (see Figure 1.5), in one case identifying a major design flaw. A bug identical to one discovered by model checking escaped testing and caused a deadlock during a flight experiment 96 million km from earth. In the Netherlands, model checking has revealed several serious design flaws in the control software of a storm surge barrier that protects the main port of Rotterdam against flooding.

Example 1.1. Concurrency and Atomicity

Most errors, such as the ones exposed in the Deep Space-1 spacecraft, are concerned with classical concurrency errors. Unforeseen interleavings between processes may cause undesired events to happen. This is exemplified by analysing the following concurrent program, in which three processes, Inc, Dec, and Reset, cooperate. They operate on the shared integer variable x with arbitrary initial value that can be accessed (i.e., read), and

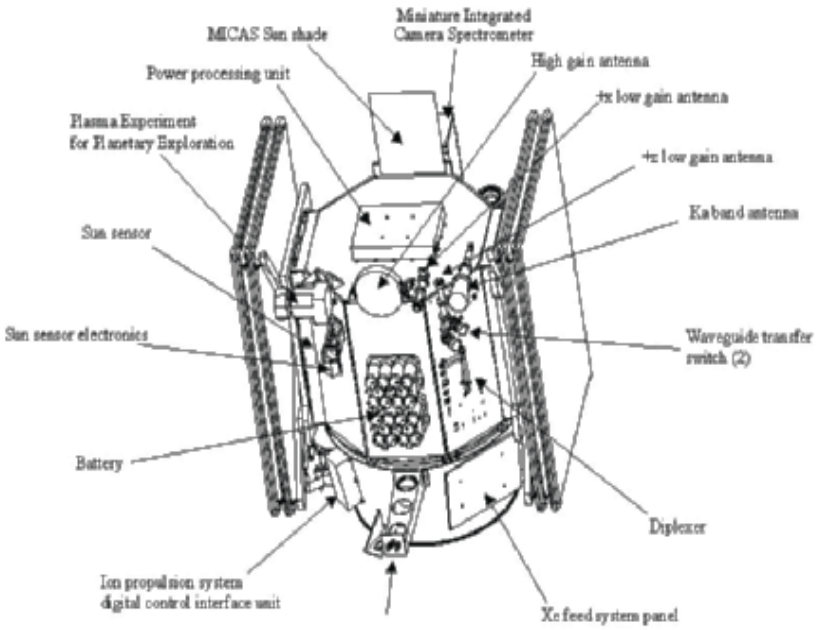


Figure 1.5: Modules of NASA's Deep Space-1 space-craft (launched in October 1998) have been thoroughly examined using model checking.

modified (i.e., written) by each of the individual processes. The processes are

$$\begin{aligned} \textbf{proc}\ \mathsf{Inc} &= \textbf{while}\ \text{true}\ \textbf{do}\ \textbf{if}\ x < 200\ \textbf{then}\ x := x + 1\ \textbf{fi}\ \textbf{od} \\ \textbf{proc}\ \mathsf{Dec} &= \textbf{while}\ \text{true}\ \textbf{do}\ \textbf{if}\ x > 0\ \textbf{then}\ x := x - 1\ \textbf{fi}\ \textbf{od} \\ \textbf{proc}\ \mathsf{Reset} &= \textbf{while}\ \text{true}\ \textbf{do}\ \textbf{if}\ x = 200\ \textbf{then}\ x := 0\ \textbf{fi}\ \textbf{od} \end{aligned}$$

Process Inc increments x if its value is smaller than 200, Dec decrements x if its value is at least 1, and Reset resets x once it has reached the value 200. They all do so repetitively.

Is the value of x always between (and including) 0 and 200? At first sight this seems to be true. A more thorough inspection, though, reveals that this is not the case. Suppose x equals 200. Process Dec tests the value of x, and passes the test, as x exceeds 0. Then, control is taken over by process Reset. It tests the value of x, passes its test, and immediately resets x to zero. Then, control is returned to process Dec and this process decrements x by one, resulting in a negative value for x (viz. -1). Intuitively, we tend to interpret the tests on x and the assignments to x as being executed atomically, i.e., as a single step, whereas in reality this is (mostly) not the case. ■

1.2 Characteristics of Model Checking

This book is devoted to the principles of model checking:

> *Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model.*

The next chapters treat the elementary technical details of model checking. This section describes the process of model checking (how to use it), presents its main advantages and drawbacks, and discusses its role in the system development cycle.

1.2.1 The Model-Checking Process

In applying model checking to a design the following different phases can be distinguished:

- *Modeling* phase:
  - model the system under consideration using the model description language of the model checker at hand;
  - as a first sanity check and quick assessment of the model perform some simulations;
  - formalize the property to be checked using the property specification language.
- *Running* phase: run the model checker to check the validity of the property in the system model.
- *Analysis* phase:
  - property satisfied? → check next property (if any);
  - property violated? →
    1. analyze generated counterexample by simulation;
    2. refine the model, design, or property;
    3. repeat the entire procedure.
  - out of memory? → try to reduce the model and try again.

In addition to these steps, the entire verification should be planned, administered, and organized. This is called *verification organization*. We discuss these phases of model checking in somewhat more detail below.

Modeling The prerequisite inputs to model checking are a model of the system under consideration and a formal characterization of the property to be checked.

Models of systems describe the behavior of systems in an accurate and unambiguous way. They are mostly expressed using *finite-state automata*, consisting of a finite set of states and a set of transitions. States comprise information about the current values of variables, the previously executed statement (e.g., a program counter), and the like. Transitions describe how the system evolves from one state into another. For realistic systems, finite-state automata are described using a model description language such as an appropriate dialect/extension of C, Java, VHDL, or the like. Modeling systems, in particular concurrent ones, at the right abstraction level is rather intricate and is really an art; it is treated in more detail in Chapter 2.

In order to improve the quality of the model, a simulation prior to the model checking can take place. Simulation can be used effectively to get rid of the simpler category of modeling errors. Eliminating these simpler errors before any form of thorough checking takes place may reduce the costly and time-consuming verification effort.

To make a rigorous verification possible, properties should be described in a precise and unambiguous manner. This is typically done using a property specification language. We focus in particular on the use of a *temporal logic* as a property specification language, a form of modal logic that is appropriate to specify relevant properties of ICT systems. In terms of mathematical logic, one checks that the system description is a model of a temporal logic formula. This explains the term "model checking". Temporal logic is basically an extension of traditional propositional logic with operators that refer to the behavior of systems over time. It allows for the specification of a broad range of relevant system properties such as functional correctness (does the system do what it is supposed to do?), reachability (is it possible to end up in a deadlock state?), safety ("something bad never happens"), liveness ("something good will eventually happen"), fairness (does, under certain conditions, an event occur repeatedly?), and real-time properties (is the system acting in time?).

Although the aforementioned steps are often well understood, in practice it may be a serious problem to judge whether the formalized problem statement (model + properties) is an adequate description of the actual verification problem. This is also known as the *validation* problem. The complexity of the involved system, as well as the lack of precision

of the informal specification of the system's functionality, may make it hard to answer this question satisfactorily. Verification and validation should not be confused. Verification amounts to check that the design satisfies the requirements that have been identified, i.e., verification is "check that we are building the thing right". In validation, it is checked whether the formal model is consistent with the informal conception of the design, i.e., validation is "check that we are verifying the right thing".

Running the Model Checker The model checker first has to be initialized by appropriately setting the various options and directives that may be used to carry out the exhaustive verification. Subsequently, the actual model checking takes place. This is basically a solely algorithmic approach in which the validity of the property under consideration is checked in all states of the system model.

Analyzing the Results There are basically three possible outcomes: the specified property is either valid in the given model or not, or the model turns out to be too large to fit within the physical limits of the computer memory.

In case the property is valid, the following property can be checked, or, in case all properties have been checked, the model is concluded to possess all desired properties.

Whenever a property is falsified, the negative result may have different causes. There may be a *modeling error*, i.e., upon studying the error it is discovered that the model does not reflect the design of the system. This implies a correction of the model, and verification has to be restarted with the improved model. This reverification includes the verification of those properties that were checked before on the erroneous model and whose verification may be invalidated by the model correction! If the error analysis shows that there is no undue discrepancy between the design and its model, then either a *design error* has been exposed, or a *property error* has taken place. In case of a design error, the verification is concluded with a negative result, and the design (together with its model) has to be improved. It may be the case that upon studying the exposed error it is discovered that the property does not reflect the informal requirement that had to be validated. This implies a modification of the property, and a new verification of the model has to be carried out. As the model is not changed, no reverification of properties that were checked before has to take place. The design is verified if and only if all properties have been checked with respect to a valid model.

Whenever the model is too large to be handled – state spaces of real-life systems may be many orders of magnitude larger than what can be stored by currently available memories – there are various ways to proceed. A possibility is to apply techniques that try to exploit

implicit regularities in the structure of the model. Examples of these techniques are the representation of state spaces using symbolic techniques such as binary decision diagrams or partial order reduction. Alternatively, rigorous abstractions of the complete system model are used. These abstractions should preserve the (non-)validity of the properties that need to be checked. Often, abstractions can be obtained that are sufficiently small with respect to a single property. In that case, different abstractions need to be made for the model at hand. Another way of dealing with state spaces that are too large is to give up the precision of the verification result. The probabilistic verification approaches explore only part of the state space while making a (often negligible) sacrifice in the verification coverage. The most important state-space reduction strategies are discussed in Chapters 7 through 9 of this monograph.

Verification Organization The entire model-checking process should be well organized, well structured, and well planned. Industrial applications of model checking have provided evidence that the use of version and configuration management is of particular relevance. During the verification process, for instance, different model descriptions are made describing different parts of the system, various versions of the verification models are available (e.g., due to abstraction), and plenty of verification parameters (e.g., model-checking options) and results (diagnostic traces, statistics) are available. This information needs to be documented and maintained very carefully in order to manage a practical model-checking process and to allow the reproduction of the experiments that were carried out.

1.2.2 Strengths and Weaknesses

The strengths of model checking:

- It is a *general* verification approach that is applicable to a wide range of applications such as embedded systems, software engineering, and hardware design.
- It supports *partial* verification, i.e., properties can be checked individually, thus allowing focus on the essential properties first. No complete requirement specification is needed.
- It is not vulnerable to the likelihood that an error is exposed; this contrasts with testing and simulation that are aimed at tracing the most probable defects.
- It provides *diagnostic information* in case a property is invalidated; this is very useful for debugging purposes.

- It is a potential "push-button" technology; the use of model checking requires neither a high degree of user interaction nor a high degree of expertise.
- It enjoys a rapidly increasing *interest by industry*; several hardware companies have started their in-house verification labs, job offers with required skills in model checking frequently appear, and commercial model checkers have become available.
- It can be easily *integrated* in existing development cycles; its learning curve is not very steep, and empirical studies indicate that it may lead to shorter development times.
- It has a *sound and mathematical underpinning*; it is based on theory of graph algorithms, data structures, and logic.

The weaknesses of model checking:

- It is mainly appropriate to *control-intensive* applications and less suited for data-intensive applications as data typically ranges over infinite domains.
- Its applicability is subject to *decidability issues*; for infinite-state systems, or reasoning about abstract data types (which requires undecidable or semi-decidable logics), model checking is in general not effectively computable.
- It verifies a *system model*, and not the actual system (product or prototype) itself; any obtained result is thus as good as the system model. Complementary techniques, such as testing, are needed to find fabrication faults (for hardware) or coding errors (for software).
- It checks only *stated requirements*, i.e., there is no guarantee of completeness. The validity of properties that are not checked cannot be judged.
- It suffers from the *state-space explosion* problem, i.e., the number of states needed to model the system accurately may easily exceed the amount of available computer memory. Despite the development of several very effective methods to combat this problem (see Chapters 7 and 8), models of realistic systems may still be too large to fit in memory.
- Its usage requires some *expertise* in finding appropriate abstractions to obtain smaller system models and to state properties in the logical formalism used.
- It is not guaranteed to yield correct results: as with any tool, a model checker may contain *software defects*.[2]

[2]Parts of the more advanced model-checking procedures have been formally proven correct using theorem provers to circumvent this.

- It does not allow checking *generalizations*: in general, checking systems with an arbitrary number of components, or parameterized systems, cannot be treated. Model checking can, however, suggest results for arbitrary parameters that may be verified using proof assistants.

We believe that one can never achieve absolute guaranteed correctness for systems of realistic size. Despite the above limitations we conclude that

> *Model checking is an effective technique to expose potential design errors.*

Thus, model checking can provide a significant increase in the level of confidence of a system design.

1.3 Bibliographic Notes

Model checking. Model checking originates from the independent work of two pairs in the early eighties: Clarke and Emerson [86] and Queille and Sifakis [347]. The term *model checking* was coined by Clarke and Emerson. The brute-force examination of the entire state space in model checking can be considered as an extension of automated protocol validation techniques by Hajek [182] and West [419, 420]. While these earlier techniques were restricted to checking the absence of deadlocks or livelocks, model checking allows for the examination of broader classes of properties. Introductory papers on model checking can be found in [94, 95, 96, 293, 426]. The limitations of model checking were discussed by Apt and Kozen [17]. More information on model checking is available in the earlier books by Holzmann [205], McMillan [288], and Kurshan [250] and the more recent works by Clarke, Grumberg, and Peled [92], Huth and Ryan [219], Schneider [365], and Bérard et al. [44]. The model-checking trajectory has recently been described by Ruys and Brinksma [360].

Software verification. Empirical data about software engineering is gathered by the Center for Empirically Based Software Engineering (`www.cebase.org`); their collected data about software defects has recently been summarized by Boehm and Basili [53]. The different characterizations of verification ("are we building the thing right?") and validation ("are we building the right thing?") originate from Boehm [52]. An overview of software testing is given by Whittaker [421]; books about software testing are by Myers [308] and Beizer [36]. Testing based on formal specifications has been studied extensively in the area of communication protocols. This has led to an international standard for conformance

testing [222]. The use of software verification techniques by the German software industry has been studied by Liggesmeyer et al. [275]. Books by Storey [381] and Leveson [269] describe techniques for developing safety-critical software and discuss the role of formal verification in this context. Rushby [359] addresses the role of formal methods for developing safety-critical software. The book of Peled [327] gives a detailed account of formal techniques for software reliability that includes testing, model checking, and deductive methods.

Model-checking software. Model-checking communication protocols has become popular through the pioneering work of Holzmann [205, 206]. An interesting project at Bell Labs in which a model-checking team and a traditional design team worked on the design of part of the ISDN user part protocol has been reported by Holzmann [207]. In this large case study, 112 serious design flaws were discovered while checking 145 formal properties in about 10,000 verification runs. Errors found by Clarke et al. [89] in the IEEE Futurebus+ standard (checking a model of more than 10^{30} states) has led to a substantial revision of the protocol by IEEE. Chan et al. [79] used model checking to verify the control software of a traffic control and alert system for airplanes. Recently, Staunstrup et al. [377] have reported the succesful model checking of a train model consisting of 1421 state machines comprising a state space of 10^{476} states. Lowe [278], using model checking, discovered a flaw in the well-known Needham-Schroeder authentication algorithm that remained undetected for over 17 years. The usage of formal methods (that includes model checking) in the software development process of a safety-critical system within a Dutch software house is presented by Tretmans, Wijbrans, and Chaudron [393]. The formal analysis of NASA's Mars Pathfinder and the Deep Space-1 spacecraft are addressed by Havelund, Lowry, and Penix [194], and Holzmann, Najm, and Serhrouchini [210], respectively. The automated generation of abstract models amenable to model checking from programs written in programming languages such as `C`, `C++`, or Java has been pursued, for instance, by Godefroid [170], Dwyer, Hatcliff, and coworkers [193], at Microsoft Research by Ball, Podelski, and Rajamani [33] and at NASA Research by Havelund and Pressburger [195].

Model-checking hardware. Applying model checking to hardware originates from Browne et al. [66] analyzing some moderate-size self-timed sequential circuits. Successful applications of (symbolic) model checking to large hardware systems have been first reported by Burch et al. [75] in the early nineties. They analyzed a synchronous pipeline circuit of approximately 10^{20} states. Overviews of formal hardware verification techniques can be found in works by Gupta [179], and the books by Yoeli [428] and Kropf [246]. The need for formal verification techniques for hardware verification has been advocated by, among others, Sangiovanni-Vincentelli, McGeer, and Saldanha [362]. The integration of model-checking techniques for error finding in the hardware development process at IBM has been recently described by Schlipf et al. [364] and Abarbanel-Vinov et al. [2]. They conclude that model checking is a powerful extension of the traditional verification pro-

cess, and consider it as complementary to simulation/emulation. The design of a memory bus adapter at IBM showed, e.g., that 24% of all defects were found with model checking, while 40% of these errors would most likely not have been found by simulation.

Chapter 2

Modelling Concurrent Systems

A prerequisite for model checking is a model of the system under consideration. This chapter introduces transition systems, a (by now) standard class of models to represent hardware and software systems. Different aspects for modeling concurrent systems are treated, ranging from the simple case in which processes run completely autonomously to the more realistic setting where processes communicate in some way. The chapter is concluded by considering the problem of state-space explosion.

2.1 Transition Systems

Transition systems are often used in computer science as models to describe the behavior of systems. They are basically directed graphs where nodes represent *states*, and edges model *transitions*, i.e., state changes. A state describes some information about a system at a certain moment of its behavior. For instance, a state of a traffic light indicates the current color of the light. Similarly, a state of a sequential computer program indicates the current values of all program variables together with the current value of the program counter that indicates the next program statement to be executed. In a synchronous hardware circuit, a state typically represents the current value of the registers together with the values of the input bits. Transitions specify how the system can evolve from one state to another. In the case of the traffic light a transition may indicate a switch from one color to another, whereas for the sequential program a transition typically corresponds to the execution of a statement and may involve the change of some variables and the program counter. In the case of the synchronous hardware circuit, a transition models the change of the registers and output bits on a new set of inputs.

In the literature, many different types of transition systems have been proposed. We use transition systems with *action names* for the transitions (state changes) and *atomic propositions* for the states. Action names will be used for describing communication mechanisms between processes. We use letters at the beginning of the Greek alphabet (such as α, β, and so on) to denote actions. Atomic propositions are used to formalize temporal characteristics. Atomic propositions intuitively express simple known facts about the states of the system under consideration. They are denoted by arabic letters from the beginning of the alphabet, such as a, b, c, and so on. Examples of atomic propositions are "x equals 0", or "x is smaller than 200" for some given integer variable x. Other examples are "there is more than a liter of fluid in the tank" or "there are no customers in the shop".

Definition 2.1. Transition System (TS)

A *transition system* TS is a tuple $(S, Act, \rightarrow, I, AP, L)$ where

- S is a set of states,
- Act is a set of actions,
- $\longrightarrow \subseteq S \times Act \times S$ is a transition relation,
- $I \subseteq S$ is a set of initial states,
- AP is a set of atomic propositions, and
- $L : S \rightarrow 2^{AP}$ is a labeling function.

TS is called *finite* if S, Act, and AP are finite. ■

For convenience, we write $s \xrightarrow{\alpha} s'$ instead of $(s, \alpha, s') \in \longrightarrow$. The intuitive behavior of a transition system can be described as follows. The transition system starts in some initial state $s_0 \in I$ and evolves according to the transition relation $\longrightarrow$. That is, if s is the current state, then a transition $s \xrightarrow{\alpha} s'$ originating from s is selected *nondeterministically* and taken, i.e., the action α is performed and the transition system evolves from state s into the state s'. This selection procedure is repeated in state s' and finishes once a state is encountered that has no outgoing transitions. (Note that I may be empty; in that case, the transition system has no behavior at all as no initial state can be selected.) It is important to realize that in case a state has more than one outgoing transition, the "next" transition is chosen in a purely nondeterministic fashion. That is, the outcome of this selection process is not known a priori, and, hence, no statement can be made about

the likelihood with which a certain transition is selected. Similarly, when the set of initial states consists of more than one state, the start state is selected nondeterministically.

The labeling function L relates a set $L(s) \in 2^{AP}$ of atomic propositions to any state s.[1] $L(s)$ intuitively stands for exactly those atomic propositions $a \in AP$ which are satisfied by state s. Given that Φ is a propositional logic formula, then s satisfies the formula Φ if the evaluation induced by $L(s)$ makes the formula Φ true; that is:

$$s \models \Phi \quad \text{iff} \quad L(s) \models \Phi.$$

(Basic principles of propositional logic are explained in Appendix A.3, see page 915 ff.)

Example 2.2. Beverage Vending Machine

We consider an (somewhat foolish) example, which has been established as standard in the field of process calculi. The transition system in Figure 2.1 models a preliminary design of a beverage vending machine. The machine can either deliver beer or soda. States are represented by ovals and transitions by labeled edges. State names are depicted inside the ovals. Initial states are indicated by having an incoming arrow without source.

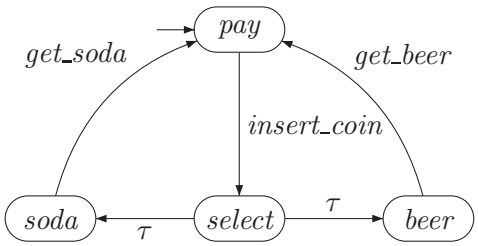


Figure 2.1: A transition system of a simple beverage vending machine.

The state space is $S = \{ pay, select, soda, beer \}$. The set of initial states consists of only one state, i.e., $I = \{ pay \}$. The (user) action *insert_coin* denotes the insertion of a coin, while the (machine) actions *get_soda* and *get_beer* denote the delivery of soda and beer, respectively. Transitions of which the action label is not of further interest here, e.g., as it denotes some internal activity of the beverage machine, are all denoted by the distinguished action symbol τ. We have:

$$Act = \{ insert\_coin, get\_soda, get\_beer, \tau \}.$$

Some example transitions are:

$$pay \xrightarrow{insert\_coin} select \quad \text{and} \quad beer \xrightarrow{get\_beer} pay.$$

[1] Recall that 2^{AP} denotes the power set of AP.

It is worthwhile to note that after the insertion of a coin, the vending machine nondeterministically can choose to provide either beer or soda.

The atomic propositions in the transition system depend on the properties under consideration. A simple choice is to let the state names act as atomic propositions, i.e., $L(s) = \{ s \}$ for any state s. If, however, the only relevant properties do not refer to the selected beverage, as in the property

"The vending machine only delivers a drink after providing a coin",

it suffices to use the two-element set of propositions $AP = \{ \mathit{paid}, \mathit{drink} \}$ with labeling function:

$$L(\mathit{pay}) = \varnothing, \quad L(\mathit{soda}) = L(\mathit{beer}) = \{ \mathit{paid}, \mathit{drink} \}, \quad L(\mathit{select}) = \{ \mathit{paid} \}.$$

Here, the proposition *paid* characterizes exactly those states in which the user has already paid but not yet obtained a beverage. ■

The previous example illustrates a certain arbitrariness concerning the choice of atomic propositions and action names. Even if the formal definition of a transition system requires determining the set of actions Act and the set of propositions AP, the components Act and AP are casually dealt with in the following. Actions are only necessary for modeling communication mechanisms as we will see later on. In cases where action names are irrelevant, e.g., because the transition stands for an internal process activity, we use a special symbol τ or, in cases where action names are not relevant, even omit the action label. The set of propositions AP is always chosen depending on the characteristics of interest. In depicting transition systems, the set of propositions AP often is not explicitly indicated and it is assumed that $AP \subseteq S$ with labeling function $L(s) = \{ s \} \cap AP$.

Crucial for modeling hard- or software systems by transition systems is the nondeterminism, which in this context is by far more than a theoretical concept. Later in this chapter (Section 2.2), we will explain in detail how transition systems can serve as a formal model for parallel systems. We mention here only that nondeterministic choices serve to model the parallel execution of independent activities by *interleaving* and to model the *conflict situations* that arise, e.g., if two processes aim to access a shared resource. Essentially, interleaving means the nondeterministic choice of the order in which order the actions of the processes that run in parallel are executed. Besides parallelism, the nondeterminism is also important for *abstraction* purposes, for *underspecification*, and to model the interface with an unknown or unpredictable *environment* (e.g., a human user). An example of the last is provided by the beverage vending machine where the user resolves the nondeterministic choice between the two τ-transitions in state "select" by choosing one of the two available drinks. The notion "underspecification" refers to early design phases where

a coarse model for a system is provided that represents several options for the possible behaviors by nondeterminism. The rough idea is that in further refinement steps the designer realizes one of the nondeterministic alternatives, but skips the others. In this sense, nondeterminism in a transition system can represent implementation freedom.

Definition 2.3. Direct Predecessors and Successors

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system. For $s \in S$ and $\alpha \in Act$, the set of direct *α-successors* of s is defined as:

$$Post(s, \alpha) \;=\; \Big\{\, s' \in S \;\mid\; s \xrightarrow{\alpha} s' \,\Big\}, \qquad Post(s) \;=\; \bigcup_{\alpha \in Act} Post(s, \alpha).$$

The set of *α-predecessors* of s is defined by:

$$Pre(s, \alpha) \;=\; \Big\{\, s' \in S \;\mid\; s' \xrightarrow{\alpha} s \,\Big\}, \qquad Pre(s) \;=\; \bigcup_{\alpha \in Act} Pre(s, \alpha).$$

■

Each state $s' \in Post(s, \alpha)$ is called a direct α-successor of s. Accordingly, each state $s' \in Post(s)$ is called a direct successor of s. The notations for the sets of direct successors are expanded to subsets of S in the obvious way (i.e., pointwise extension): for $C \subseteq S$, let

$$Post(C, \alpha) \;=\; \bigcup_{s \in C} Post(s, \alpha), \quad Post(C) \;=\; \bigcup_{s \in C} Post(s).$$

The notations $Pre(C, \alpha)$ and $Pre(C)$ are defined in an analogous way:

$$Pre(C, \alpha) \;=\; \bigcup_{s \in C} Pre(s, \alpha), \quad Pre(C) \;=\; \bigcup_{s \in C} Pre(s).$$

Terminal states of a transition system TS are states without any outgoing transitions. Once the system described by TS reaches a terminal state, the complete system comes to a halt.

Definition 2.4. Terminal State

State s in transition system TS is called *terminal* if and only if $Post(s) = \varnothing$. ■

For a transition system modeling a sequential computer program, terminal states occur as a natural phenomenon representing the termination of the program. Later on, we will

see that for transition systems modeling parallel systems, such terminal states are usually considered to be undesired (see Section 3.1, page 89 ff.).

We mentioned above that nondeterminism is crucial for modeling computer systems. However, it is often useful to consider transition systems where the "observable" behavior is deterministic, according to some notion of observables. There are two general approaches to formalize the visible behavior of a transition system: one relies on the actions, the other on the labels of the states. While the action-based approach assumes that only the executed actions are observable from outside, the state-based approach ignores the actions and relies on the atomic propositions that hold in the current state to be visible. Transition systems that are deterministic in the action-based view have at most one outgoing transition labeled with action α per state, while determinism from the view of state labels means that for any state label $A \in 2^{AP}$ and any state there is at most one outgoing transition leading to a state with label A. In both cases, it is required that there be at most one initial state.

Definition 2.5. Deterministic Transition System

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system.

1. TS is called *action-deterministic* if $|\, I \,| \leqslant 1$ and $|\, Post(s, \alpha) \,| \leqslant 1$ for all states s and actions α.
2. TS is called *AP-deterministic* if $|\, I \,| \leqslant 1$ and $|Post(s) \cap \{\, s' \in S \mid L(s') = A \,\}| \leqslant 1$ for all states s and $A \in 2^{AP}$.

■

2.1.1 Executions

So far, the behavior of a transition system has been described at an intuitive level. This will now be formalized using the notion of *executions* (also called *runs*). An execution of a transition system results from the resolution of the possible nondeterminism in the system. An execution thus describes a possible behavior of the transition system. Formally:

Definition 2.6. Execution Fragment

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system. A *finite* execution fragment ϱ of TS is an alternating sequence of states and actions ending with a state

$$\varrho \;=\; s_0\, \alpha_1\, s_1\, \alpha_2 \ldots \alpha_n\, s_n \text{ such that } s_i \xrightarrow{\alpha_{i+1}} s_{i+1} \text{ for all } 0 \leqslant i < n,$$

where $n \geqslant 0$. We refer to n as the length of the execution fragment ϱ. An *infinite* execution fragment ρ of TS is an infinite, alternating sequence of states and actions:

$$\rho \;=\; s_0\,\alpha_1\,s_1\,\alpha_2\,s_2\,\alpha_3\ldots \text{ such that } s_i \xrightarrow{\alpha_{i+1}} s_{i+1} \text{ for all } 0 \leqslant i.$$

■

Note that the sequence s with $s \in S$ is a legal finite execution fragment of length n=0. Each prefix of odd length of an infinite execution fragment is a finite execution fragment. From now on, the term *execution fragment* will be used to denote either a finite or an infinite execution fragment. Execution fragments $\varrho = s_0\,\alpha_1\,\ldots\,\alpha_n\,s_n$ and $\rho = s_0\,\alpha_1\,s_1\,\alpha_2\,\ldots$ will be written respectively as

$$\varrho \;=\; s_0 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} s_n \qquad \text{and} \qquad \rho \;=\; s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots\ .$$

An execution fragment is called maximal when it cannot be prolonged:

Definition 2.7. Maximal and Initial Execution Fragment

A *maximal* execution fragment is either a finite execution fragment that ends in a terminal state, or an infinite execution fragment. An execution fragment is called *initial* if it starts in an initial state, i.e., if $s_0 \in I$. ■

Example 2.8. Executions of the Beverage Vending Machine

Some examples of execution fragments of the beverage vending machine described in Example 2.2 (page 21) are as follows. For simplicity, the action names are abbreviated, e.g., *sget* is a shorthand for *get_soda* and *coin* for *insert_coin*.

$$\rho_1 \;=\; \mathit{pay} \xrightarrow{\mathit{coin}} \mathit{select} \xrightarrow{\tau} \mathit{soda} \xrightarrow{\mathit{sget}} \mathit{pay} \xrightarrow{\mathit{coin}} \mathit{select} \xrightarrow{\tau} \mathit{soda} \xrightarrow{\mathit{sget}} \ldots$$

$$\rho_2 \;=\; \mathit{select} \xrightarrow{\tau} \mathit{soda} \xrightarrow{\mathit{sget}} \mathit{pay} \xrightarrow{\mathit{coin}} \mathit{select} \xrightarrow{\tau} \mathit{beer} \xrightarrow{\mathit{bget}} \ldots$$

$$\varrho \;=\; \mathit{pay} \xrightarrow{\mathit{coin}} \mathit{select} \xrightarrow{\tau} \mathit{soda} \xrightarrow{\mathit{sget}} \mathit{pay} \xrightarrow{\mathit{coin}} \mathit{select} \xrightarrow{\tau} \mathit{soda}\ .$$

Execution fragments ρ_1 and ϱ are initial, but ρ_2 is not. ϱ is not maximal as it does not end in a terminal state. Assuming that ρ_1 and ρ_2 are infinite, they are maximal. ■

Definition 2.9. Execution

An *execution* of transition system TS is an initial, maximal execution fragment. ■

In Example 2.8, ρ_1 is an execution, while ρ_2 and ϱ are not. Note that ρ_2 is maximal but not initial, while ϱ is initial but not maximal.

A state s is called reachable if there is some execution fragment that ends in s and that starts in some initial state.

Definition 2.10. Reachable States

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system. A state $s \in S$ is called *reachable* in TS if there exists an initial, finite execution fragment

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n = s .$$

$Reach(TS)$ denotes the set of all reachable states in TS. ■

2.1.2 Modeling Hardware and Software Systems

This section illustrates the use of transition systems by elaborating on the modeling of (synchronous) hardware circuits and sequential data-dependent systems – a kind of simple sequential computer programs. For both cases, the basic concept is that states represent possible storage configurations (i.e., evaluations of relevant "variables"), and state changes (i.e., transitions) represent changes of "variables". Here, the term "variable" has to be understood in the broadest sense. For computer programs a variable can be a control variable (like a program counter) or a program variable. For circuits a variable can, e.g, stand for either a register or an input bit.

Sequential Hardware Circuits

Before presenting a general recipe for modeling sequential hardware circuits as transition systems we consider a simple example to clarify the basic concepts.

Example 2.11. A Simple Sequential Hardware Circuit

Consider the circuit diagram of the sequential circuit with input variable x, output variable y, and register r (see left part of Figure 2.2). The control function for output variable y is given by

$$\lambda_y = \neg(x \oplus r)$$

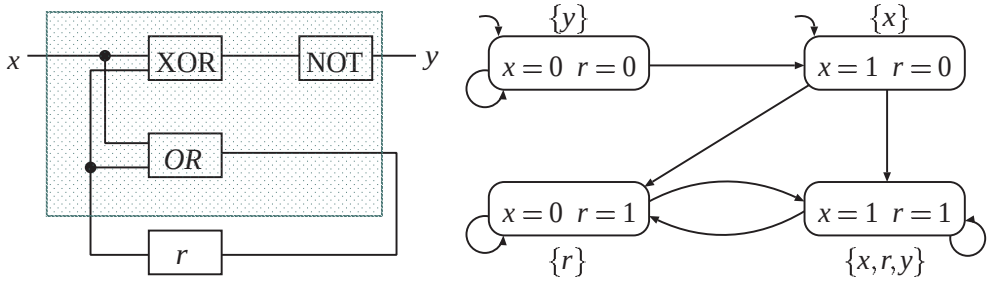


Figure 2.2: Transition system representation of a simple hardware circuit.

where $\oplus$ stands for exclusive or (XOR, or parity function). The register evaluation changes according to the circuit function

$$\delta_r \;=\; x \vee r \;.$$

Note that once the register evaluation is $[r = 1]$, r keeps that value. Under the initial register evaluation $[r = 0]$, the circuit behavior is modeled by the transition system TS with state space

$$S \;=\; \mathit{Eval}(x, r)$$

where $\mathit{Eval}(x, r)$ stands for the set of evaluations of input variable x and register variable r. The initial states of TS are $I = \{\, \langle x = 0, r = 0\rangle, \langle x = 1, r = 0\rangle \,\}$. Note that there are two initial states as we do not make any assumption about the initial value of the input bit x.

The set of actions is irrelevant and omitted here. The transitions result directly from the functions λ_y and δ_r. For instance, $\langle x = 0, r = 1\rangle \rightarrow \langle x = 0, r = 1\rangle$ if the next input bit equals 0, and $\langle x = 0, r = 1\rangle \rightarrow \langle x = 1, r = 1\rangle$ if the next input bit is 1.

It remains to consider the labeling L. Using the set of atomic propositions $AP = \{\, x, y, r \,\}$, then, e.g., the state $\langle x = 0, r = 1\rangle$ is labeled with $\{\, r \,\}$. It is not labeled with y since the circuit function $\neg(x \oplus r)$ results in the value 0 for this state. For state $\langle x = 1, r = 1\rangle$ we obtain $L(\langle x = 1, r = 1\rangle) = \{\, x, r, y \,\}$, as λ_y yields the value 1. Accordingly, we obtain: $L(\langle x = 0, r = 0\rangle) = \{\, y \,\}$, and $L(\langle x = 1, r = 0\rangle) = \{\, x \,\}$. The resulting transition system (with this labeling) is depicted in the right part of Figure 2.2.

Alternatively, using the set of propositions $AP' = \{\, x, y \,\}$ – the register evaluations are assumed to be "invisible" – one obtains:

$$\begin{array}{lcllcl} L'(\langle x = 0, r = 0\rangle) & = & \{\, y \,\} & \qquad L'(\langle x = 0, r = 1\rangle) & = & \varnothing \\ L'(\langle x = 1, r = 0\rangle) & = & \{\, x \,\} & \qquad L'(\langle x = 1, r = 1\rangle) & = & \{\, x, y \,\} \end{array}$$

The propositions in AP' suffice to formalize, e.g., the property "the output bit y is set infinitely often". Properties that refer to the register r are not expressible. ■

The approach taken in this example can be generalized toward arbitrary sequential hardware circuits (without "don't cares") with n input bits $x_1, \ldots, x_n$, m output bits $y_1, \ldots, y_m$, and k registers $r_1, \ldots, r_k$ as follows. The states of the transition system represent the evaluations of the $n+k$ input and register bits $x_1, \ldots, x_n, r_1, \ldots, r_k$. The evaluation of output bits depends on the evaluations of input bits and registers and can be derived from the states. Transitions represent the behavior, whereas it is assumed that the values of input bits are *nondeterministically* provided (by the circuit environment). Furthermore, we assume a given initial register evaluation

$$[r_1 = c_{0,1}, \ldots, r_k = c_{0,k}]$$

where $c_{0,i}$ denotes the initial value of register i for $0 < i \leqslant k$. Alternatively, a set of possible initial register evaluations may be given.

The transition system $TS = (S, Act, \rightarrow, I, AP, L)$ modeling this sequential hardware circuit has the following components. The state space S is determined by

$$S \;=\; \mathit{Eval}(x_1, \ldots, x_n, r_1, \ldots, r_k).$$

Here, $\mathit{Eval}(x_1, \ldots, x_n, r_1, \ldots, r_k)$ stands for the set of evaluations of input variables x_i and registers r_j and can be identified with the set $\{0,1\}^{n+k}$.[2] Initial states are of the form $(\ldots, c_{0,1}, \ldots, c_{0,k})$ where the k registers are evaluated with their initial value. The first n components prescribing the values of input bits are arbitrary. Thus, the set of initial states is

$$I \;=\; \Big\{(a_1, \ldots, a_n, c_{0,1}, \ldots, c_{0,k}) \mid a_1, \ldots, a_n \in \{0,1\}\Big\}.$$

The set Act of actions is irrelevant, and we choose $Act = \{\tau\}$. For simplicity, let the set of atomic propositions be

$$AP \;=\; \{x_1, \ldots, x_n, y_1, \ldots, y_m, r_1, \ldots, r_k\}\ .$$

(In practice, this could be defined as any subset of this AP). Thus, any register, any input bit, and any output bit can be used as an atomic proposition. The labeling function assigns to any state $s \in \mathit{Eval}(x_1, \ldots, x_n, r_1, \ldots, r_k)$ exactly those atomic propositions x_i, r_j which are evaluated to 1 under s. If for state s, output bit y_i is evaluated to 1, then (and only then) the atomic proposition y_i is part of $L(s)$. Thus,

$$\begin{aligned} L(a_1, \ldots, a_n, c_1, \ldots, c_k) \;=\; & \{x_i \mid a_i = 1\} \cup \{r_j \mid c_j = 1\} \\ & \cup \{y_i \mid s \models \lambda_{y_i}(a_1, \ldots, a_n, c_1, \ldots, c_k) = 1\} \end{aligned}$$

[2] An evaluation $s \in \mathit{Eval}(\cdot)$ is a mapping which assigns a value $s(x_i) \in \{0,1\}$ to any input bit x_i. Similarly, every register r_j is mapped onto a value $s(r_j) \in \{0,1\}$. To simplify matters, we assume every element $s \in S$ to be a bit-tuple of length $n+k$. The ith bit is set if and only if x_i is evaluated to 1 $(0 < i \leqslant n)$. Accordingly, the $n+j$th bit indicates the evaluation of r_j $(0 < j \leqslant k)$.

where $\lambda_{y_i} : S \to \{0,1\}$ is the switching function corresponding to output bit y_i that results from the gates of the circuit.

Transitions exactly represent the behavior. In the following, let δ_{r_j} denote the transition function for register r_j resulting from the circuit diagram. Then:

$$(\underbrace{a_1, \ldots, a_n}_{\text{input evaluation}} , \underbrace{c_1, \ldots, c_k}_{\text{register evaluation}}) \xrightarrow{\tau} (a'_1, \ldots, a'_n, c'_1, \ldots, c'_k)$$

if and only if $c'_j = \delta_{r_j}(a_1, \ldots, a_n, c_1, \ldots, c_k)$. Assuming that the evaluation of input bits changes nondeterministically, no restrictions on the bits $a'_1, \ldots, a'_n$ are imposed.

It is left to the reader to check that applying this recipe to the example circuit in the left part of Figure 2.2 indeed results in the transition system depicted in the right part of that figure.

Data-Dependent Systems

The executable actions of a data-dependent system typically result from conditional branching, as in

$$\textbf{if } x\%2 = 1 \textbf{ then } x := x + 1 \textbf{ else } x := 2{\cdot}x \textbf{ fi}.$$

In principle, when modeling this program fragment as a transition system, the conditions of transitions could be omitted and conditional branchings could be replaced by nondeterminism; but, generally speaking, this results in a very abstract transition system for which only a few relevant properties can be verified. Alternatively, *conditional transitions* can be used and the resulting graph (labeled with conditions) can be unfolded into a transition system that subsequently can be subject to verification. This unfolding approach is detailed out below. We first illustrate this by means of an example.

Example 2.12. Beverage Vending Machine Revisited

Consider an extension of the beverage vending machine described earlier in Example 2.2 (page 21) which counts the number of soda and beer bottles and returns inserted coins if the vending machine is empty. For the sake of simplicity, the vending machine is represented by the two locations *start* and *select*. The following conditional transitions

$$start \xrightarrow{true\ :\ coin} select \qquad \text{and} \qquad start \xrightarrow{true\ :\ refill} start$$

model the insertion of a coin and refilling the vending machine. Labels of conditional transitions are of the form $g : \alpha$ where g is a Boolean condition (called guard), and α is an action that is possible once g holds. As the condition for both conditional transitions

above always holds, the action *coin* is always enabled in the starting location. To keep things simple, we assume that by *refill* both storages are entirely refilled. Conditional transitions

$$\mathit{select} \xleftarrow{\mathit{nsoda} > 0 \,:\, \mathit{sget}} \mathit{start} \qquad \text{and} \qquad \mathit{select} \xleftarrow{\mathit{nbeer} > 0 \,:\, \mathit{bget}} \mathit{start}$$

model that soda (or beer) can be obtained if there is some soda (or beer) left in the vending machine. The variables *nsoda* and *nbeer* record the number of soda and beer bottles in the machine, respectively. Finally, the vending machine automatically switches to the initial *start* location while returning the inserted coin once there are no bottles left:

$$\mathit{select} \xleftarrow{\mathit{nsoda} = 0 \,\wedge\, \mathit{nbeer} = 0 \colon \mathit{ret\_coin}} \mathit{start}$$

Let the maximum capacity of both bottle repositories be *max*. The insertion of a coin (by action *coin*) leaves the number of bottles unchanged. The same applies when a coin is returned (by action *ret_coin*). The effect of the other actions is as follows:

Action	Effect
refill	$\mathit{nsoda} := \mathit{max};\ \mathit{nbeer} := \mathit{max}$
sget	$\mathit{nsoda} := \mathit{nsoda} - 1$
bget	$\mathit{nbeer} := \mathit{nbeer} - 1$

The graph consisting of locations as nodes and conditional transitions as edges is not a transition system, since the edges are provided with conditions. A transition system, however, can be obtained by "unfolding" this graph. For instance, Figure 2.3 on page 31 depicts this unfolded transition system when *max* equals 2. The states of the transition system keep track of the current location in the graph described above and of the number of soda- and beer bottles in the vending machine (as indicated by the gray and black dots, respectively, inside the nodes of the graph). ■

The ideas outlined in the previous example are formalized by using so-called *program graphs* over a set *Var* of typed variables such as *nsoda* and *nbeer* in the example. Essentially, this means that a standardized type (e.g., `boolean`, `integer`, or `char`) is associated with each variable. The type of variable x is called the domain $\mathit{dom}(x)$ of x. Let $\mathit{Eval}(\mathit{Var})$ denote the set of (variable) evaluations that assign values to variables. $\mathit{Cond}(\mathit{Var})$ is the set of Boolean conditions over *Var*, i.e., propositional logic formulae whose propositional symbols are of the form "$\overline{x} \in \overline{D}$" where $\overline{x} = (x_1, \ldots, x_n)$ is a tuple consisting of pairwise distinct variables in *Var* and $\overline{D}$ is a subset of $\mathit{dom}(x_1) \times \ldots \times \mathit{dom}(x_n)$. The proposition

$$(-3 < x - x' \leqslant 5) \ \wedge\ (x \leqslant 2{\cdot}x') \ \wedge\ (y = \mathit{green}),$$

for instance, is a legal Boolean condition for integer variables x and x', and y a variable with, e.g., $\mathit{dom}(y) = \{\, \mathit{red}, \mathit{green} \,\}$. Here and in the sequel, we often use simplified notations

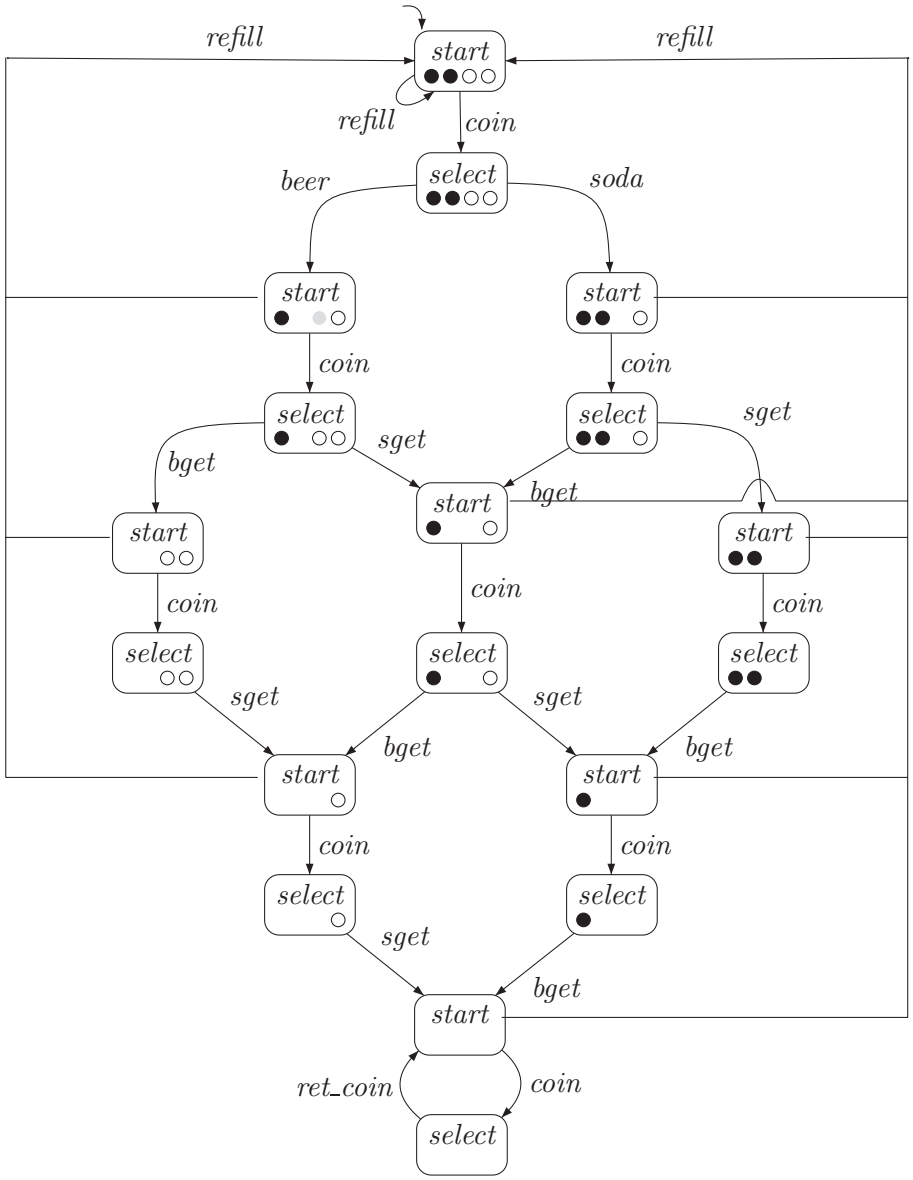


Figure 2.3: Transition system modeling the extended beverage vending machine.

for the propositional symbols such as "$3 < x - x' \leqslant 5$" instead of "$(x, x') \in \{\, (n, m) \in \mathbb{N}^2 \mid 3 < n - m \leqslant 5 \,\}$".

Initially, we do not restrict the domains. $dom(x)$ can be an arbitrary, possibly infinite, set. Even if in real computer systems all domains are finite (e.g., the type `integer` only includes integers n of a finite domain, like $-2^{16} < n < 2^{16}$), then the logical or algorithmic structure of a program is often based on infinite domains. The decision which restrictions on domains are useful for implementation, e.g., how many bits should be provided for representation of variables of type `integer` is delayed until a later design stage and is ignored here.

A *program graph* over a set of typed variables is a digraph whose edges are labeled with conditions on these variables and actions. The effect of the actions is formalized by means of a mapping

$$\mathit{Effect} : \mathit{Act} \times \mathit{Eval}(\mathit{Var}) \to \mathit{Eval}(\mathit{Var})$$

which indicates how the evaluation η of variables is changed by performing an action. If, e.g., α denotes the action $x := y{+}5$, where x and y are integer variables, and η is the evaluation with $\eta(x) = 17$ and $\eta(y) = -2$, then

$$\mathit{Effect}(\alpha, \eta)(x) \;=\; \eta(y) + 5 \;=\; -2 + 5 = 3, \quad \text{and} \quad \mathit{Effect}(\alpha, \eta)(y) \;= \eta(y) \;=\; -2.$$

$\mathit{Effect}(\alpha, \eta)$ is thus the evaluation that assigns 3 to x and -2 to y. The nodes of a program graph are called *locations* and have a control function since they specify which of the conditional transitions are possible.

Definition 2.13. Program Graph (PG)

A *program graph* PG over set Var of typed variables is a tuple $(\mathit{Loc}, \mathit{Act}, \mathit{Effect}, \hookrightarrow, \mathit{Loc}_0, g_0)$ where

- Loc is a set of locations and Act is a set of actions,
- $\mathit{Effect} : \mathit{Act} \times \mathit{Eval}(\mathit{Var}) \to \mathit{Eval}(\mathit{Var})$ is the effect function,
- $\hookrightarrow \subseteq\ \mathit{Loc} \times \mathit{Cond}(\mathit{Var}) \times \mathit{Act} \times \mathit{Loc}$ is the conditional transition relation,
- $\mathit{Loc}_0 \subseteq \mathit{Loc}$ is a set of initial locations,
- $g_0 \in \mathit{Cond}(\mathit{Var})$ is the initial condition.

■

The notation $\ell \overset{g:\alpha}{\hookrightarrow} \ell'$ is used as shorthand for $(\ell, g, \alpha, \ell') \in \hookrightarrow$. The condition g is also called the *guard* of the conditional transition $\ell \overset{g:\alpha}{\hookrightarrow} \ell'$. If the guard is a tautology (e.g., $g = \text{true}$ or $g = (x < 1) \vee (x \geqslant 1)$), then we simply write $\ell \overset{\alpha}{\hookrightarrow} \ell'$.

The behavior in location $\ell \in \mathit{Loc}$ depends on the current variable evaluation η. A nondeterministic choice is made between all transitions $\ell \overset{g:\alpha}{\hookrightarrow} \ell'$ which satisfy condition g in evaluation η (i.e., $\eta \models g$). The execution of action α changes the evaluation of variables according to $\mathit{Effect}(\alpha, \cdot)$. Subsequently, the system changes into location ℓ'. If no such transition is possible, the system stops.

Example 2.14. Beverage Vending Machine

The graph described in Example 2.12 (page 29) is a program graph. The set of variables is

$$\mathit{Var} \;=\; \{\, \mathit{nsoda},\quad \mathit{nbeer} \,\}$$

where both variables have the domain $\{\, 0, 1, \ldots, \mathit{max} \,\}$. The set Loc of locations equals $\{\, \mathit{start}, \mathit{select} \,\}$ with $\mathit{Loc}_0 = \{\, \mathit{start} \,\}$, and

$$\mathit{Act} = \{\, \mathit{bget}, \mathit{sget}, \mathit{coin}, \mathit{ret\_coin}, \mathit{refill} \,\} \;.$$

The effect of the actions is defined by:

$$\begin{array}{lcl}
\mathit{Effect}(\mathit{coin}, \eta) & = & \eta \\
\mathit{Effect}(\mathit{ret_coin}, \eta) & = & \eta \\
\mathit{Effect}(\mathit{sget}, \eta) & = & \eta[\mathit{nsoda} := \mathit{nsoda}{-}1] \\
\mathit{Effect}(\mathit{bget}, \eta) & = & \eta[\mathit{nbeer} := \mathit{nbeer}{-}1] \\
\mathit{Effect}(\mathit{refill}, \eta) & = & [\mathit{nsoda} := \mathit{max}, \mathit{nbeer} := \mathit{max}]
\end{array}$$

Here, $\eta[\mathit{nsoda} := \mathit{nsoda}{-}1]$ is a shorthand for evaluation η' with $\eta'(\mathit{nsoda}) = \eta(\mathit{nsoda}){-}1$ and $\eta'(x) = \eta(x)$ for all variables different from *nsoda*. The initial condition g_0 states that initially both storages are entirely filled, i.e., $g_0 = (\mathit{nsoda} = \mathit{max} \wedge \mathit{nbeer} = \mathit{max})$. ■

Each program graph can be interpreted as a transition system. The underlying transition system of a program graph results from *unfolding*. Its states consist of a control component, i.e., a location ℓ of the program graph, together with an evaluation η of the variables. States are thus pairs of the form $\langle \ell, \eta \rangle$. Initial states are initial locations that satisfy the initial condition g_0. To formulate properties of the system described by a program graph, the set AP of propositions is comprised of locations $\ell \in \mathit{Loc}$ (to be able to state at which control location the system currently is), and Boolean conditions for the variables. For example, a proposition like

$$(x \leqslant 5) \;\wedge\; (y \text{ is even}) \;\wedge\; (\ell \in \{\, 1, 2 \,\})$$

can be formulated with x, y being integer variables and with locations being naturals. The labeling of states is such that $\langle \ell, v \rangle$ is labeled with ℓ and with all conditions (over *Var*) that hold in η. The transition relation is determined as follows. Whenever $\ell \xrightarrow{g:\alpha} \ell'$ is a conditional transition in the program graph, and the guard g holds in the current evaluation η, then there is a transition from state $\langle \ell, \eta \rangle$ to state $\langle \ell', \mathit{Effect}(\alpha, \eta) \rangle$. Note that the transition is not guarded. Formally:

Definition 2.15. Transition System Semantics of a Program Graph

The transition system $TS(PG)$ of program graph

$$PG = (\mathit{Loc}, \mathit{Act}, \mathit{Effect}, \hookrightarrow, \mathit{Loc}_0, g_0)$$

over set *Var* of variables is the tuple $(S, \mathit{Act}, \longrightarrow, I, AP, L)$ where

- $S = \mathit{Loc} \times \mathit{Eval}(\mathit{Var})$
- $\longrightarrow \subseteq S \times \mathit{Act} \times S$ is defined by the following rule (see remark below):

$$\frac{\ell \overset{g:\alpha}{\hookrightarrow} \ell' \quad \wedge \quad \eta \models g}{\langle \ell, \eta \rangle \xrightarrow{\alpha} \langle \ell', \mathit{Effect}(\alpha, \eta) \rangle}$$

- $I = \{\langle \ell, \eta \rangle \mid \ell \in \mathit{Loc}_0, \eta \models g_0\}$
- $AP = \mathit{Loc} \cup \mathit{Cond}(\mathit{Var})$
- $L(\langle \ell, \eta \rangle) = \{\ell\} \cup \{g \in \mathit{Cond}(\mathit{Var}) \mid \eta \models g\}$.

■

The definition of $TS(PG)$ determines a very large set of propositions AP. But generally, only a small part of AP is necessary to formulate the relevant system properties. In the following, we exploit the degrees of freedom in choosing the set of propositions of $TS(PG)$ and only use the atomic propositions needed in the context at hand.

Remark 2.16. Structured Operational Semantics

In Definition 2.15, the transition relation is defined using the so-called SOS-notation (Structured Operational Semantics). This notation will be frequently used in the remainder of this monograph. The notation

$$\frac{\text{premise}}{\text{conclusion}}$$

should be read as follows. If the proposition above the "solid line" (i.e., the premise) holds, then the proposition under the fraction bar (i.e., the conclusion) holds as well. Such "if ..., then ..." propositions are also called *inference rules* or simply *rules*. If the premise is a tautology, it may be omitted (as well as the "solid line"). In the latter case, the rule is also called an *axiom*.

Phrases like "The relation $\rightarrow$ is defined by the following (axioms and) rules" have the meaning of an inductive definition where the relation $\rightarrow$ is defined as the *smallest* relation satisfying the indicated axioms and rules. ■

2.2 Parallelism and Communication

In the previous section, we have introduced the notion of transition systems and have shown how sequential hardware circuits and data-dependent systems (like simple sequential computer programs) can be effectively modeled as transition systems. In reality, however, most hard- and software systems are not sequential but parallel in nature. This section describes several mechanisms to provide operational models for *parallel* systems by means of transition systems. These mechanisms range from simple mechanisms where no communication between the participating transitions systems takes place, to more advanced (and realistic) schemes in which messages can be transferred, either synchronously (i.e., by means of "handshaking") or asynchronously (i.e., by buffers with a positive capacity). Let us assume that the operational (stepwise) behavior of the processes that run in parallel are given by transition systems $TS_1, \ldots, TS_n$. The goal is to define an operator $\|$, such that:

$$TS \;=\; TS_1 \parallel TS_2 \parallel \;\ldots\; \parallel TS_n$$

is a transition system that specifies the behavior of the parallel composition of transition systems TS_1 through TS_n. Here, it is assumed that $\|$ is a commutative and associative operator. The nature of the operator $\|$ will, of course, depend on the kind of communication that is supported. We will for instance see that some notions of parallel composition do not yield an associative operator. In the remainder of this section, several variants of $\|$ will be considered and illustrated by means of examples. Note that the above scheme may be repeated for TS_i, i.e., TS_i may again be a transition system that is composed of several transition systems:

$$TS_i \;=\; TS_{i,1} \parallel TS_{i,1} \parallel \;\ldots\; \parallel TS_{i,n_i} \;.$$

By using parallel composition in this hierarchical way, complex systems can be described in a rather structured way.

2.2.1 Concurrency and Interleaving

A widely adopted paradigm for parallel systems is that of *interleaving*. In this model, one abstracts from the fact that a system is actually composed of a set of (partly) independent components. That is to say, the global system state – composed of the current individual states of the components – plays a key role in interleaving. Actions of an independent component are merged (also called weaved), or "interleaved", with actions from other components. Thus, concurrency is represented by (pure) interleaving, that is, the nondeterministic choice between activities of the simultaneously acting processes (or components). This perspective is based on the view that only one processor is available on which the actions of the processes are interlocked. The "one-processor view" is only a modeling concept and also applies if the processes run on different processors. Thereby, (at first) no assumptions are made about the order in which the different processes are executed. If there are, e.g., two nonterminating processes P and Q, say, acting completely independent of each other, then

$$\begin{array}{cccccccccc} P & Q & P & Q & P & Q & Q & Q & P & \ldots \\ P & P & Q & P & P & Q & P & P & Q & \ldots \\ P & Q & P & P & Q & P & P & P & Q & \ldots \end{array}$$

are three possible sequences in which the steps (i.e., execution of actions) of P and Q can be interlocked. (In Chapter 3, certain restrictions will be discussed to ensure that each participating processor is treated in a somewhat "fair" manner. In particular, execution sequences like $P, P, P, \ldots$, where Q is completely ignored, are ruled out. Unless stated otherwise, we accept all possible interleavings, including the unfair ones.)

The interleaving representation of concurrency is subject to the idea that there is a scheduler which interlocks the steps of concurrently executing processes according to an a priori unknown strategy. This type of representation completely abstracts from the speed of the participating processes and thus models *any* possible realization by a single-processor machine or by several processors with arbitrary speeds.

Example 2.17. Two Independent Traffic Lights

Consider the transition systems of two traffic lights for nonintersecting (i.e., parallel) roads. It is assumed that the traffic lights switch completely independent of each other. For example, the traffic lights may be controlled by pedestrians who would like to cross the road. Each traffic light is modeled as a simple transition system with two states, one state

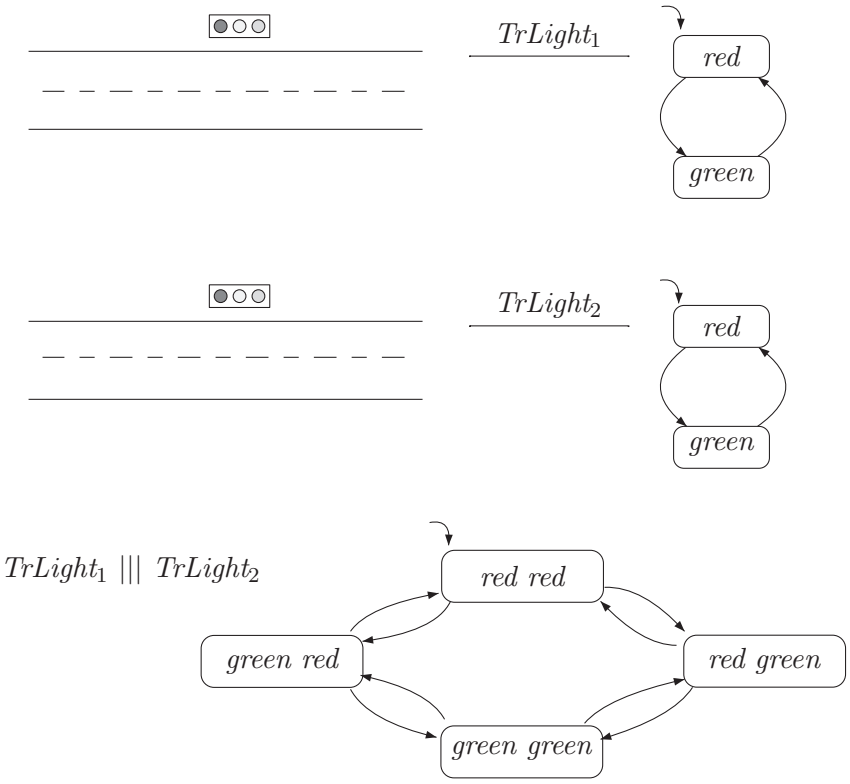


Figure 2.4: Example of interleaving operator for transition systems.

modeling a red light, the other one modeling a green light (see upper part of Figure 2.4). The transition system of the parallel composition of both traffic lights is sketched at the bottom of Figure 2.4 where $|||$ denotes the interleaving operator. In principle, any form of interlocking of the "actions" of the two traffic lights is possible. For instance, in the initial state where both traffic lights are red, there is a non-deterministic choice between which of the lights turns green. Note that this nondeterminism is descriptive, and does not model a scheduling problem between the traffic lights (although it may seem so). ■

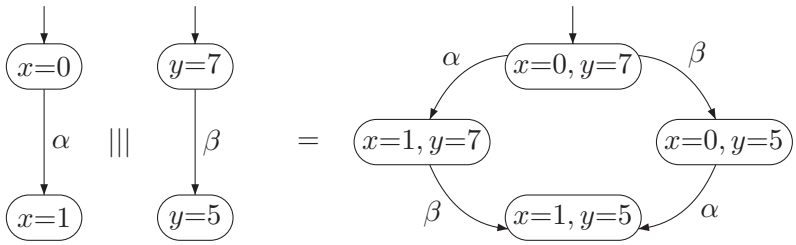
An important justification for interleaving is the fact that the effect of concurrently executed, independent actions α and β, say, is identical to the effect when α and β are successively executed in arbitrary order. This can symbolically be stated as

$$\mathit{Effect}(\alpha \,|||\, \beta, \eta) \;=\; \mathit{Effect}((\alpha\,;\,\beta) + (\beta\,;\,\alpha), \eta)$$

where the operator semicolon ; stands for sequential execution, + stands for nondeterministic choice, and $|||$ for the concurrent execution of independent activities. This fact can be easily understood when the effect is considered from two independent value assignments

$$\underbrace{x := x + 1}_{=\alpha} \;|||\; \underbrace{y := y - 2}_{=\beta}.$$

When initially $x = 0$ and $y = 7$, then x has the value 1 and y the value 5 after executing α and β, independent of whether the assignments occur concurrently (i.e., simultaneously) or in some arbitrary successive order. This is depicted in terms of transition systems as follows:



Note that the independence of actions is crucial. For dependent actions, the order of actions is typically essential: e.g., the final value of variable x in the parallel program $x := x{+}1 \,|||\, x := 2{\cdot}x$ (with initial value $x{=}0$, say) depends on the order in which the assignments $x := x{+}1$ and $x := 2{\cdot}x$ take place.

We are now in a position to formally define the interleaving (denoted $|||$) of transition systems. The transition system $TS_1 \,|||\, TS_2$ represents a parallel system resulting from the weaving (or merging) of the actions of the components as described by TS_1 and TS_2. It is assumed that no communication and no contentions (on shared variables) occur at all. The ("global") states of $TS_1 \,|||\, TS_2$ are pairs $\langle s_1, s_2\rangle$ consisting of "local" states s_i of the components TS_i. The outgoing transitions of the global state $\langle s_1, s_2\rangle$ consist of the outgoing transitions of s_1 together with those of s_2. Accordingly, whenever the composed system is in state $\langle s_1, s_2\rangle$, a nondeterministic choice is made between all outgoing transitions of local state s_1 and those of local state s_2.

Definition 2.18. Interleaving of Transition Systems

Let $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP_i, L_i)$ $i{=}1,2$, be two transition systems. The transition system $TS_1 \,|||\, TS_2$ is defined by:

$$TS_1 \,|||\, TS_2 \;=\; (S_1 \times S_2, Act_1 \cup Act_2, \rightarrow, I_1 \times I_2, AP_1 \cup AP_2, L)$$

where the transition relation $\rightarrow$ is defined by the following rules:

$$\frac{s_1 \xrightarrow{\alpha}_1 s_1'}{\langle s_1, s_2\rangle \xrightarrow{\alpha} \langle s_1', s_2\rangle} \quad \text{and} \quad \frac{s_2 \xrightarrow{\alpha}_2 s_2'}{\langle s_1, s_2\rangle \xrightarrow{\alpha} \langle s_1, s_2'\rangle}$$

and the labeling function is defined by $L(\langle s_1, s_2\rangle) = L(s_1) \cup L(s_2)$. ■

Example 2.19.

Consider the two independent traffic lights described in Example 2.17 (page 36). The depicted transition system is actually the transition system

$$TS \;=\; TrLight_1 \,|||\, TrLight_2$$

that originates from interleaving. ■

For program graphs PG_1 (on Var_1) and PG_2 (on Var_2) without shared variables (i.e., $Var_1 \cap Var_2 = \varnothing$), the interleaving operator, which is applied to the appropriate transition systems, yields a transition system

$$TS(PG_1) \;|||\; TS(PG_2)$$

that describes the behavior of the simultaneous execution of PG_1 and PG_2.

2.2.2 Communication via Shared Variables

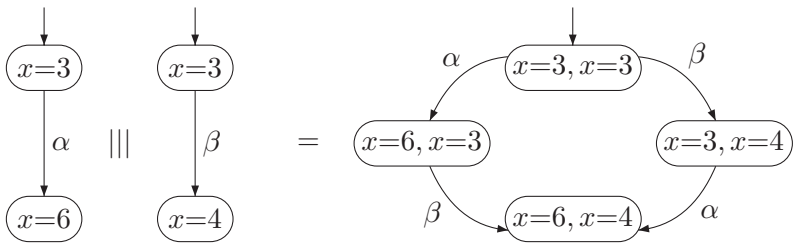
The interleaving operator $|||$ can be used to model *asynchronous* concurrency in which the subprocesses act completely independent of each other, i.e., without any form of message passing or contentions on shared variables. The interleaving operator for transition systems is, however, too simplistic for most parallel systems with concurrent or communicating components. An example of a system whose components have variables in common—shared variables so to speak—will make this clear.

Example 2.20. The Interleaving Operator for Concurrent Processes

Regard the program graph for the instructions α and β of the parallel program

$$\underbrace{x := 2{\cdot}x}_{\text{action } \alpha} \;|||\; \underbrace{x := x+1}_{\text{action } \beta}$$

where we assume that initially $x = 3$. (To simplify the picture, the locations have been skipped.) The transition system $TS(PG_1) \,|||\, TS(PG_2)$ contains, e.g., the inconsistent state $\langle x{=}6, x{=}4\rangle$ and, thus, does not reflect the intuitive behavior of the parallel execution of α and β:



The problem in this example is that the actions α and β access the shared variable x and therefore are competing. The interleaving operator for transition systems, however, "blindly" constructs the Cartesian product of the individual state spaces without considering these potential conflicts. Accordingly, it is not identified that the local states x=6 and x=4 describe exclusive events. ■

In order to deal with parallel programs with shared variables, an interleaving operator will be defined on the level of program graphs (instead of directly on transition systems). The interleaving of program graphs PG_1 and PG_2 is denoted $PG_1 \;|||\; PG_2$. The underlying transition system of the resulting program graph $PG_1 \;|||\; PG_2$, i.e., $TS(PG_1 \,|||\, PG_2)$ (see Definition 2.15, page 34) faithfully describes a parallel system whose components communicate via shared variables. Note that, in general, $TS(PG_1 \,|||\, PG_2) \neq TS(PG_1) \,|||\, TS(PG_2)$.

Definition 2.21. Interleaving of Program Graphs

Let $PG_i = (Loc_i, Act_i, Effect_i, \hookrightarrow_i, Loc_{0,i}, g_{0,i})$, for i=1, 2 be two program graphs over the variables Var_i. Program graph $PG_1 \,|||\, PG_2$ over $Var_1 \cup Var_2$ is defined by

$$PG_1 \,|||\, PG_2 \;=\; (Loc_1 \times Loc_2, Act_1 \uplus Act_2, Effect, \hookrightarrow, Loc_{0,1} \times Loc_{0,2}, g_{0,1} \wedge g_{0,2})$$

where $\hookrightarrow$ is defined by the rules:

$$\frac{\ell_1 \overset{g:\alpha}{\hookrightarrow}_1 \ell_1'}{\langle \ell_1, \ell_2 \rangle \overset{g:\alpha}{\hookrightarrow} \langle \ell_1', \ell_2 \rangle} \quad \text{and} \quad \frac{\ell_2 \overset{g:\alpha}{\hookrightarrow}_2 \ell_2'}{\langle \ell_1, \ell_2 \rangle \overset{g:\alpha}{\hookrightarrow} \langle \ell_1, \ell_2' \rangle}$$

and $Effect(\alpha, \eta) \;=\; Effect_i(\alpha, \eta)$ if $\alpha \in Act_i$. ■

The program graphs PG_1 and PG_2 have the variables $Var_1 \cap Var_2$ in common. These are the shared (sometimes also called "global") variables. The variables in $Var_1 \setminus Var_2$ are the local variables of PG_1, and similarly, those in $Var_2 \setminus Var_1$ are the local variables of PG_2.

Example 2.22. Interleaving of Program Graphs

Consider the program graphs PG_1 and PG_2 that correspond to the assignments $x := x{+}1$ and $x := 2{\cdot}x$, respectively. The program graph $PG_1 \,|||\, PG_2$ is depicted in the bottom left of Figure 2.5. Its underlying transition system $TS(PG_1 \,|||\, PG_2)$ is depicted in the bottom right of that figure where it is assumed that initially x equals 3. Note that the nondeterminism in the initial state of the transition system does not represent concurrency but just the possible resolution of the contention between the statements $x := 2{\cdot}x$ and $x := x{+}1$ that both modify the shared variable x. ■

The distinction between local and shared variables has also an impact on the actions of the composed program graph $PG_1 \,|||\, PG_2$. Actions that access (i.e., inspect or modify) shared

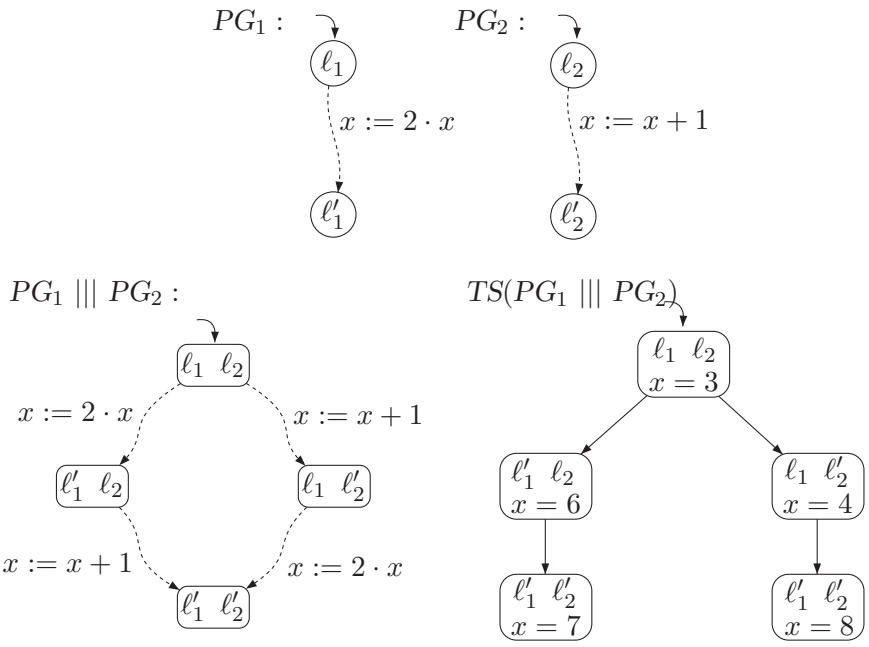


Figure 2.5: Interleaving of two example program graphs.

variables may be considered as "critical"; otherwise, they are viewed to be noncritical. (For the sake of simplicity, we are a bit conservative here and consider the inspection of shared variables as critical.) The difference between the critical and noncritical actions becomes clear when interpreting the (possible) nondeterminism in the transition system $TS(PG_1 \,|||\, PG_2)$. nondeterminism in a state of this transition system may stand either for

(i) an "internal" nondeterministic choice within program graph PG_1 or PG_2,

(ii) the interleaving of noncritical actions of PG_1 and PG_2, or

(iii) the resolution of a contention between critical actions of PG_1 and PG_2 (concurrency).

In particular, a noncritical action of PG_1 can be executed in parallel with critical or noncritical actions of PG_2 as it will only affect its local variables. By symmetry, the same applies to the noncritical actions of PG_2. Critical actions of PG_1 and PG_2, however, cannot be executed simultaneously as the value of the shared variables depends on the order of executing these actions (see Example 2.20). Instead, any global state where critical actions of PG_1 and PG_2 are enabled describe a concurrency situation that has to be resolved by an appropriate scheduling strategy. (Simultaneous reading of shared variables could be allowed, however.)

Remark 2.23. On Atomicity

For modeling a parallel system by means of the interleaving operator for program graphs it is decisive that the actions $\alpha \in Act$ are indivisible. The transition system representation only expresses the effect of the completely executed action α. If there is, e.g., an action α with its effect being described by the statement sequence

$$x := x + 1;\ y := 2x + 1;\ \textbf{if}\ x < 12\ \textbf{then}\ z := (x - z)^2 * y\ \textbf{fi},$$

then an implementation is assumed which does *not* interlock the basic substatements $x := x+1$, $y := 2x+1$, the comparison "$x < 12$", and, possibly, the assignment $z := (x-z)^2 * y$ with other concurrent processes. In this case,

$$\begin{array}{lcll} \mathit{Effect}(\alpha,\eta)(x) & = & \eta(x) + 1 & \\ \mathit{Effect}(\alpha,\eta)(y) & = & 2(\eta(x) + 1) + 1 & \\ \mathit{Effect}(\alpha,\eta)(z) & = & \begin{cases} (\eta(x) + 1 - \eta(z))^2 * 2(\eta(x) + 1) + 1 & \text{if } \eta(x) + 1 < 12 \\ \eta(z) & \text{otherwise} \end{cases} & \end{array}$$

Hence, statement sequences of a process can be declared *atomic* by program graphs when put as a single label to an edge. In program texts such multiple assignments are surrounded by brackets $\langle \ldots \rangle$. ■

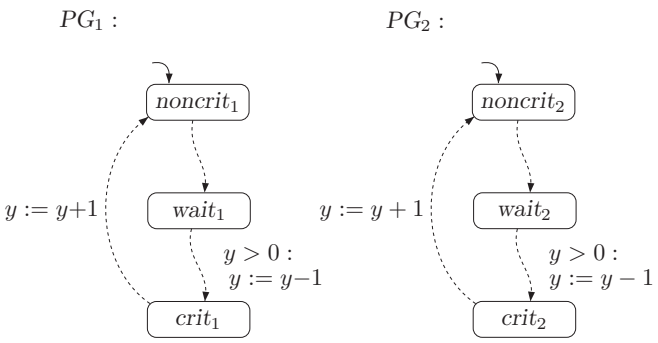


Figure 2.6: Individual program graphs for semaphore-based mutual exclusion.

Example 2.24. Mutual Exclusion with Semaphores

Consider two simplified processes $P_i, i{=}1, 2$ of the form:

```
P_i  loop forever
     ⋮                    (* noncritical actions *)
     request
     critical section
     release
     ⋮                    (* noncritical actions *)
     end loop
```

Processes P_1 and P_2 are represented by the program graphs PG_1 and PG_2, respectively, that share the binary semaphore y. $y{=}0$ indicates that the semaphore—the lock to get access to the critical section—is currently possessed by one of the processes. When $y{=}1$, the semaphore is free. The program graphs PG_1 and PG_2 are depicted in Figure 2.6.

For the sake of simplicity, local variables and shared variables different from y are not considered. Also, the activities inside and outside the critical sections are omitted. The locations of PG_i are $noncrit_i$ (representing the noncritical actions), $wait_i$ (modeling the situation in which P_i waits to enter its critical section), and $crit_i$ (modeling the critical section). The program graph $PG_1 \,|||\, PG_2$ consists of nine locations, including the (undesired) location $\langle crit_1, crit_2 \rangle$ that models the situation where both P_1 and P_2 are in their critical section, see Figure 2.7.

When unfolding $PG_1 \,|||\, PG_2$ into the transition system $TS_{Sem} = TS(PG_1 \,|||\, PG_2)$ (see Figure 2.8 on page 45), it can be easily checked that from the 18 global states in TS_{Sem}

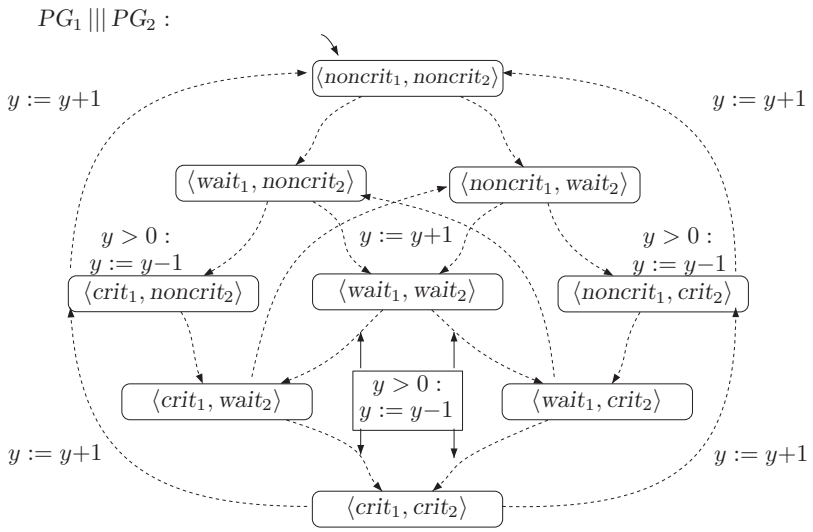


Figure 2.7: $PG_1 \,|||\, PG_2$ for semaphore-based mutual exclusion.

only the following eight states are reachable:

$$\begin{array}{ll} \langle noncrit_1, noncrit_2, y = 1 \rangle & \langle noncrit_1, wait_2, y = 1 \rangle \\ \langle wait_1, noncrit_2, y = 1 \rangle & \langle wait_1, wait_2, y = 1 \rangle \\ \langle noncrit_1, crit_2, y = 0 \rangle & \langle crit_1, noncrit_2, y = 0 \rangle \\ \langle wait_1, crit_2, y = 0 \rangle & \langle crit_1, wait_2, y = 0 \rangle \end{array}$$

States $\langle noncrit_1, noncrit_2, y = 1 \rangle$, and $\langle noncrit_1, crit_2, y = 0 \rangle$ stand for examples of situations where both P_1 and P_2 are able to concurrently execute actions. Note that in Figure 2.8 n stands for *noncrit*, w for *wait*, and c for *crit*. The nondeterminism in these states thus stand for interleaving of noncritical actions. State $\langle crit_1, wait_2, y = 0 \rangle$, e.g., represents a situation where only PG_1 is active, whereas PG_2 is waiting.

From the fact that the global state $\langle crit_1, crit_2, y = \ldots \rangle$ is unreachable in TS_{Sem}, it follows that processes P_1 and P_2 cannot be simultaneously in their critical section. The parallel system thus satisfies the so-called *mutual exclusion* property. ■

In the previous example, the nondeterministic choice in state $\langle wait_1, wait_2, y = 1 \rangle$ represents a contention between allowing either P_1 or P_2 to enter its critical section. The resolution of this scheduling problem—which process is allowed to enter its critical section next?—is left open, however. In fact, the parallel program of the previous example is "abstract" and does not provide any details on how to resolve this contention. At later design stages, for example, when implementing the semaphore y by means of a queue of waiting processes (or the like), a decision has to be made on how to schedule the processes that are enqueued for acquiring the semaphore. At that stage, a last-in first-out (LIFO),

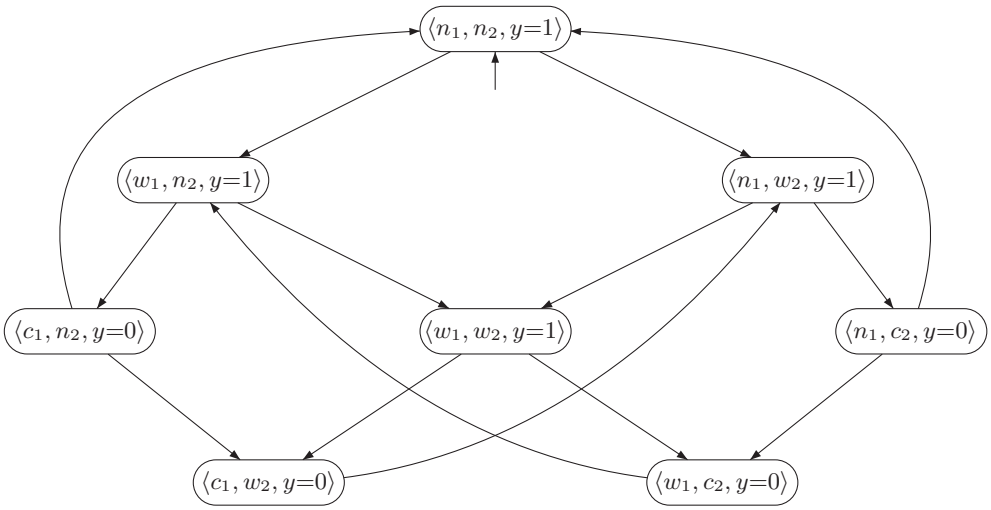


Figure 2.8: Mutual exclusion with semaphore (transition system representation).

first-in first-out (FIFO), or some other scheduling discipline can be chosen. Alternatively, another (more concrete) mutual exclusion algorithm could be selected that resolves this scheduling issue explicitly. A prominent example of such algorithm has been provided in 1981 by Peterson [332].

Example 2.25. Peterson's Mutual Exclusion Algorithm

Consider the processes P_1 and P_2 with the shared variables b_1, b_2, and x. b_1 and b_2 are Boolean variables, while x can take either the value 1 or 2, i.e., $dom(x) = \{1, 2\}$. The scheduling strategy is realized using x as follows. If both processes want to enter the critical section (i.e., they are in location $wait_i$), the value of variable x decides which of the two processes may enter its critical section: if $x = i$, then P_i may enter its critical section (for $i = 1, 2$). On entering location $wait_1$, process P_1 performs $x := 2$, thus giving privilege to process P_2 to enter the critical section. The value of x thus indicates which process has its turn to enter the critical section. Symmetrically, P_2 sets x to 1 when starting to wait. The variables b_i provide information about the current location of P_i. More precisely,

$$b_i \;=\; wait_i \vee crit_i \;.$$

b_i is set when P_i starts to wait. In pseudocode, P_1 performs as follows (the code for process P_2 is similar):

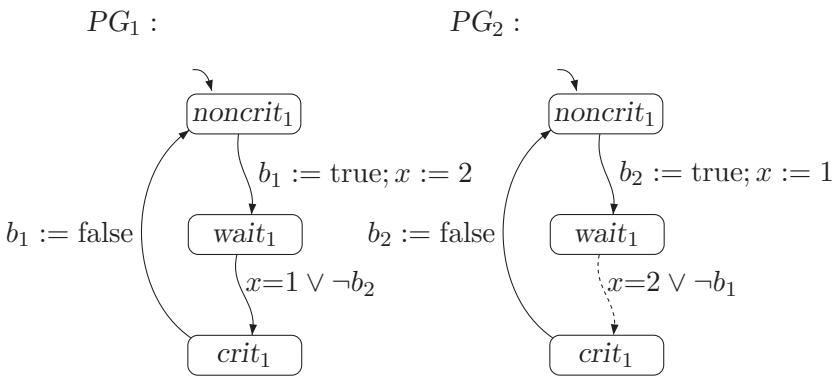


Figure 2.9: Program graphs for Peterson's mutual exclusion algorithm.

P_1 **loop forever**
⋮ (* noncritical actions *)
$\langle b_1 := \text{true};\ x := 2 \rangle$; (* request *)
wait until $(x = 1 \ \vee \ \neg b_2)$
do critical section **od**
$b_1 := \text{false}$ (* release *)
⋮ (* noncritical actions *)
end loop

Process P_i is represented by program graph PG_i over $Var = \{ x, b_1, b_2 \}$ with locations $noncrit_i$, $wait_i$, and $crit_i$, see Figure 2.9 above. The reachable part of the underlying transition system $TS_{Pet} = TS(PG_1 \,|||\, PG_2)$ has the form as indicated in Figure 2.10 (page 47), where for convenience n_i, w_i, c_i are used for $noncrit_i$, $wait_i$, and $crit_i$, respectively. The last digit of the depicted states indicates the evaluation of variable x. For convenience, the values for b_i are not indicated. Its evaluation can directly be deduced from the location of PG_i. Further, $b_1 = b_2 = \text{false}$ is assumed as the initial condition.

Each state in TS_{Pet} has the form $\langle loc_1, loc_2, x, b_1, b_2 \rangle$. As PG_i has three possible locations and b_i and x each can take two different values, the total number of states of TS_{Pet} is 72. Only ten of these states are reachable. Since there is no reachable state with P_1 and P_2 being in their critical section, it can be concluded that Peterson's algorithm satisfies the mutual exclusion property.

In the above program, the multiple assignments $b_1 := \text{true}; x := 2$ and $b_2 := \text{true}; x := 1$ are considered as indivisible (i.e., atomic) actions. This is indicated by the brackets $\langle$

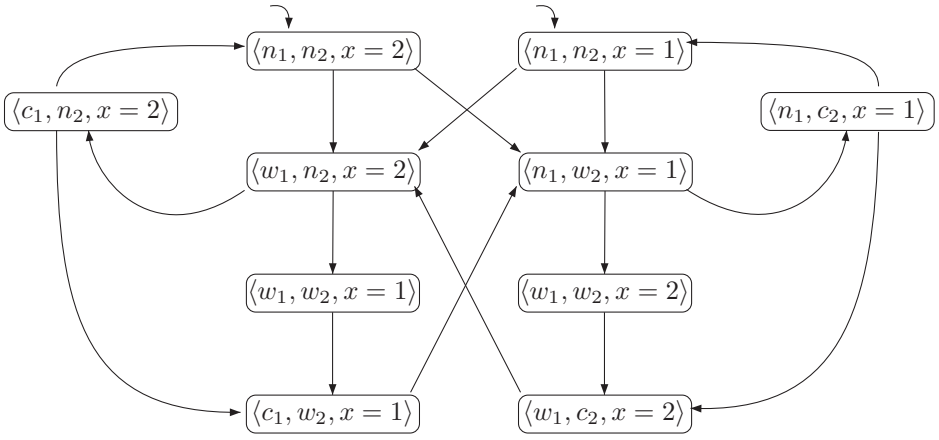


Figure 2.10: Transition system of Peterson's mutual exclusion algorithm.

and $\rangle$ in the program text, and is also indicated in the program graphs PG_1 and PG_2. We like to emphasize that this is not essential, and has only been done to simplify the transition system TS_{Pet}. Mutual exclusion is also ensured when both processes perform the assignments $b_i := \text{true}$ and $x := \ldots$ in this order but in a nonatomic way. Note that, for instance, the order "first $x := \ldots$, then $b_i := \text{true}$" does not guarantee mutual exclusion. This can be seen as follows. Assume that the location inbetween the assignments $x := \ldots$ and $b_i := \text{true}$ in program graph P_i is called req_i. The state sequence

$$\begin{array}{rcccc}
\langle noncrit_1, & noncrit_2, & x = 1, & b_1 = \text{false}, & b_2 = \text{false}\rangle \\
\langle noncrit_1, & req_2, & x = 1, & b_1 = \text{false}, & b_2 = \text{false}\rangle \\
\langle req_1, & req_2, & x = 2, & b_1 = \text{false}, & b_2 = \text{false}\rangle \\
\langle wait_1, & req_2, & x = 2, & b_1 = \text{true}, & b_2 = \text{false}\rangle \\
\langle crit_1, & req_2, & x = 2, & b_1 = \text{true}, & b_2 = \text{false}\rangle \\
\langle crit_1, & wait_2, & x = 2, & b_1 = \text{true}, & b_2 = \text{true}\rangle \\
\langle crit_1, & crit_2, & x = 2, & b_1 = \text{true}, & b_2 = \text{true}\rangle
\end{array}$$

is an initial execution fragment where P_1 enters its critical section first (as $b_2 = \text{false}$) after which P_2 enters its critical section (as $x = 2$). As a result, both processes are simultaneously in their critical section and mutual exclusion is violated. ■

2.2.3 Handshaking

So far, two mechanisms for parallel processes have been considered: interleaving and shared-variable programs. In interleaving, processes evolve completely autonomously

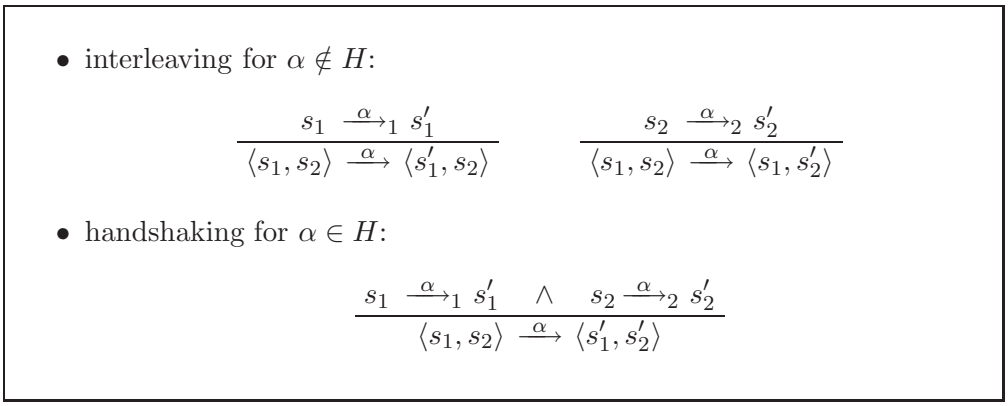


Figure 2.11: Rules for handshaking.

whereas according to the latter type processes "communicate" via shared variables. In this subsection, we consider a mechanism by which concurrent processes interact via handshaking. The term "handshaking" means that concurrent processes that want to interact have to do this in a *synchronous* fashion. That is to say, processes can interact only if they are both participating in this interaction at the same time—they "shake hands".

Information that is exchanged during handshaking can be of various nature, ranging from the value of a simple integer, to complex data structures such as arrays or records. In the sequel, we do not dwell upon the content of the exchanged messages. Instead, an abstract view is adopted and only communication (also called synchronization) actions are considered that represent the occurrence of a handshake and not the content.

To do so, a set H of *handshake actions* is distinguished with $\tau \notin H$. Only if both participating processes are ready to execute the same handshake action, can message passing take place. All actions outside H (i.e., actions in $Act \setminus H$) are independent and therefore can be executed autonomously in an interleaved fashion.

Definition 2.26. Handshaking (Synchronous Message Passing)

Let $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP_i, L_i)$, i=1, 2 be transition systems and $H \subseteq Act_1 \cap Act_2$ with $\tau \notin H$. The transition system $TS_1 \parallel_H TS_2$ is defined as follows:

$$TS_1 \parallel_H TS_2 \;=\; (S_1 \times S_2, Act_1 \cup Act_2, \rightarrow, I_1 \times I_2, AP_1 \cup AP_2, L)$$

where $L(\langle s_1, s_2\rangle) \;=\; L_1(s_1) \cup L_2(s_2)$, and where the transition relation $\rightarrow$ is defined by the rules shown in Figure 2.11. ■

Notation: $TS_1 \parallel TS_2$ abbreviates $TS_1 \parallel_H TS_2$ for $H = Act_1 \cap Act_2$.

Remark 2.27. Empty Set of Handshake Actions

When the set H of handshake actions is empty, all actions of the participating processes can take place autonomously, i.e., in this special case, handshaking reduces to interleaving

$$TS_1 \parallel_{\varnothing} TS_2 \quad = \quad TS_1 \;|||\; TS_2.$$

■

The operator $\parallel_H$ defines the handshaking between two transition systems. Handshaking is commutative, but not associative in general. That is, in general we have

$$TS_1 \parallel_H (TS_2 \parallel_{H'} TS_3) \;\neq\; (TS_1 \parallel_H TS_2) \parallel_{H'} TS_3 \quad \text{for } H \neq H'.$$

However, for a fixed set H of handshake actions over which all processes synchronize, the operator $\parallel_H$ is associative. Let

$$TS \;=\; TS_1 \parallel_H TS_2 \parallel_H \;\ldots\; \parallel_H TS_n,$$

denote the parallel composition of transition systems TS_1 through TS_n where $H \subseteq Act_1 \cap \ldots \cap Act_n$ is a subset of the set of actions Act_i of all transition systems. This form of *multiway* handshaking is appropriate to model broadcasting, a communication form in which a process can transmit a datum to several other processes simultaneously.

In many cases, processes communicate in a pairwise fashion over their common actions. Let $TS_1 \| \ldots \| TS_n$ denote the parallel composition of TS_1 through TS_n (with $n > 0$) where TS_i and TS_j $(0 < i \neq j \leqslant n)$ synchronize over the set of actions $H_{i,j} = Act_i \cap Act_j$ such that $H_{i,j} \cap Act_k = \varnothing$ for $k \notin \{ i, j \}$. It is assumed that $\tau \notin H_{i,j}$. The formal definition of $TS_1 \| \ldots \| TS_n$ is analogous to Definition 2.26. The state space of $TS_1 \| \ldots \| TS_n$ results from the Cartesian product of the state spaces of TS_i. The transition relation $\rightarrow$ is defined by the following rules:

- for $\alpha \in Act_i \setminus (\bigcup_{\substack{0<j\leqslant n \\ i\neq j}} H_{i,j})$ and $0 < i \leqslant n$:

$$\frac{s_i \xrightarrow{\alpha}_i s_i'}{\langle s_1, \ldots, s_i, \ldots, s_n \rangle \xrightarrow{\alpha} \langle s_1, \ldots, s_i', \ldots s_n \rangle}$$

- for $\alpha \in H_{i,j}$ and $0 < i < j \leqslant n$:

$$\frac{s_i \xrightarrow{\alpha}_i s_i' \quad \wedge \quad s_j \xrightarrow{\alpha}_j s_j'}{\langle s_1, \ldots, s_i, \ldots, s_j, \ldots, s_n \rangle \xrightarrow{\alpha} \langle s_1, \ldots, s_i', \ldots, s_j', \ldots, s_n \rangle}$$

According to the first rule, components can execute actions that are not subject to handshaking in a completely autonomous manner as in interleaving. The second rule states that processes TS_i and TS_j $(i \neq j)$ have to perform every handshaking action in $Act_i \cap Act_j$ together. These rules are in fact just generalizations of those given in Figure 2.11.

Example 2.28. Mutual Exclusion by Means of an Arbiter

An alternative solution to the mutual exclusion problem between processes P_1 and P_2 (as before) is to model the binary semaphore that regulates access to the critical section by a separate parallel process that interacts with P_1 and P_2 by means of handshaking. For simplicity, we ignore the waiting phase and assume that P_i simply alternates infinitely often between noncritical and critical sections. Assume (much simplified) transition system representations TS_1 and TS_2 with just two states: $crit_i$ and $noncrit_i$. The new process, named *Arbiter*, mimics a binary semaphore (see Figure 2.12). P_1 and P_2 communicate with the *Arbiter* via handshaking over the set $H = \{ request, rel \}$. Accordingly, the actions *request* (requesting to access the critical section) and *rel* (to leave the critical section) have to be executed synchronously with the *Arbiter*. The complete system

$$TS_{Arb} \;=\; (TS_1 \;|||\; TS_2) \parallel Arbiter$$

guarantees mutual exclusion since there are no states of TS_{Arb} where both P_1 and P_2 are in their critical section (see bottom part of Figure 2.12). Note that in the initial state of $TS_1 ||| TS_2$, the *Arbiter* determines which process will enter the critical section next. ■

Example 2.29. Booking System

Consider a (strongly simplified) booking system at a cashier of a supermarket. The system consists of three processes: the bar code reader *BCR*, the actual booking program *BP*, and the printer *Printer*. The bar code reader reads a bar code and communicates the data of the just scanned product to the booking program. On receiving such data, the booking program transmits the price of the article to the printer that prints the article Id together with the price on the receipt. The interactions between the bar code reader and the booking program, and between the booking program and the printer is performed by handshaking. Each process consist of just two states, named 0 and 1 (see Figure 2.13 for the transitions systems of *BCR*, *BP*, and *Printer*).

The complete system is given by:

$$BCR \parallel BP \parallel Printer.$$

The transition system of the overall system is depicted in Figure 2.14 on page 52. The initial global state of this system is $\langle 0, 0, 0 \rangle$, or in short, 000. In global state 010, e.g., the

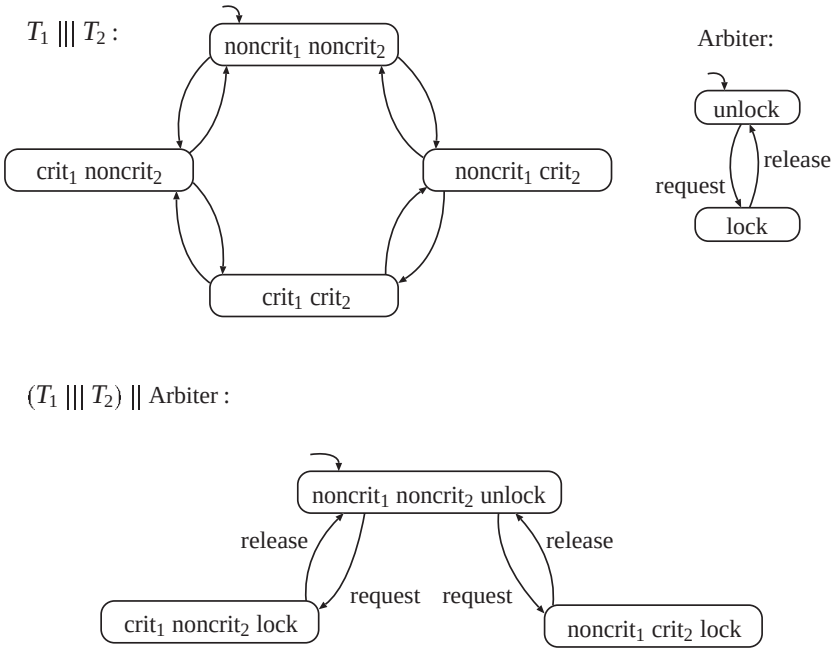


Figure 2.12: Mutual exclusion using handshaking with arbiter process.

nondeterminism stands for the concurrent execution of the actions scanning the bar code and the synchronous transfer of the price to the printer. ■

Example 2.30. Railroad Crossing

For a railroad crossing a control system needs to be developed that on receipt of a signal indicating that a train is approaching closes the gates, and only opens these gates after the train has sent a signal indicating that it crossed the road. The requirement that should be met by the control system is that the gates are always closed when the train is crossing the road. The complete system consists of the three components *Train*, *Gate*, and *Controller*:

$$\mathit{Train} \parallel \mathit{Gate} \parallel \mathit{Controller}.$$

Figure 2.15 depicts the transition systems of these components from left (modeling the *Train*) to right (modeling the *Gate*). For simplicity, it is assumed that all trains pass the relevant track section in the same direction—from left to right. The states of the transition system for the *Train* have the following intuitive meaning: in state *far* the train is not close to the crossing, in state *near* it is approaching the crossing and has just sent a signal to notify this, and in state *in* it is at the crossing. The states of the *Gate* have the obvious interpretation. The state changes of the *Controller* stand for handshaking with

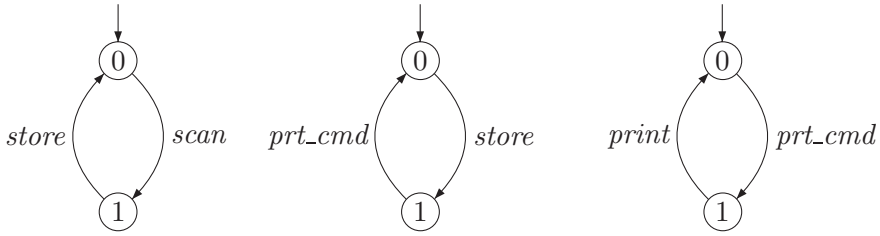


Figure 2.13: The components of the book keeping example.

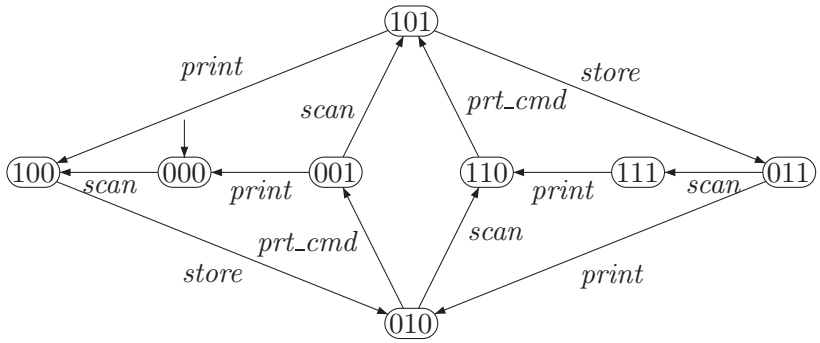


Figure 2.14: Transition system representation of the booking system.

the trains (via the actions *approach* and *exit*) and the *Gate* (via the actions *lower* and *raise* via which the *Controller* causes the gate to close or to open, respectively).

Figure 2.16 (above) illustrates the transition system of the overall system. A closer inspection of this transition system reveals that the system suffers from a design flaw. This can be seen from the following initial execution fragment:

$$\langle far, 0, up\rangle \xrightarrow{approach} \langle near, 1, up\rangle \xrightarrow{enter} \langle in, 1, up\rangle$$

in which the gate is about to close, while the train is (already) at the crossing. The nondeterminism in global state $\langle near, 1, up\rangle$ stands for concurrency: the train approaches the crossing, while the gate is being closed. In fact, the basic concept of the design is correct if and only if closing the gate does not take more time than the train needs to get to the crossing once it signals—"I am approaching". Such real-time constraints cannot be formulated by the concepts introduced so far. The interleaving representation for parallel systems is completely *time-abstract*. In Chapter 9, concepts and techniques will be introduced to specify and verify such real-time aspects. ■

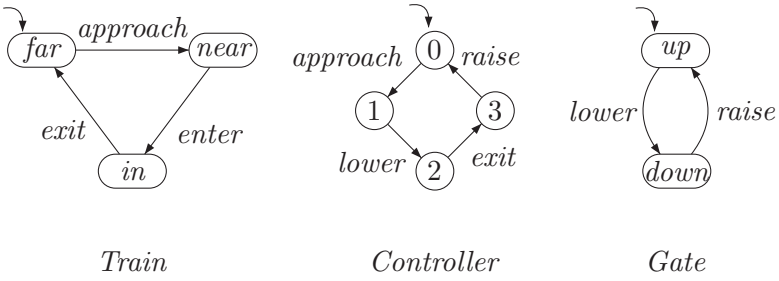


Figure 2.15: The components of the railroad crossing.

2.2.4 Channel Systems

This section introduces channel systems, parallel systems where processes communicate via so-called *channels*, i.e., first-in, first-out buffers that may contain messages. We consider channel systems that are closed. That is to say, processes may communicate with other processes in the system (via channels), but not with processes outside the system. Channel systems are popular for describing communication protocols and form the basis of PROMELA, the input language of the SPIN model checker.

Intuitively, a channel system consists of n (data-dependent) processes P_1 through P_n. Each P_i is specified by a program graph PG_i which is extended with *communication actions*. Transitions of these program graphs are either the usual conditional transitions (labeled with guards and actions) as before, or one of the communication actions with their respective intuitive meaning:

$c!v$	transmit the value v along channel c,
$c?x$	receive a message via channel c and assign it to variable x.

When considering channel c as buffer, the communication action $c!v$ puts value v (at the rear of) the buffer whereas $c?x$ retrieves an element from (the front of) the buffer while assigning it to x. It is assumed implicitly that variable x is of the right type, i.e., it has a type that is compatible to that of the messages that are put into channel c. Let

$$Comm \;=\; \Big\{\; c!v,\; c?x \;\mid\; c \in Chan,\; v \in dom(c),\; x \in Var \text{ with } \; dom(x) \supseteq dom(c) \;\Big\}$$

denote the set of communication actions where *Chan* is a finite set of channels with typical element c.

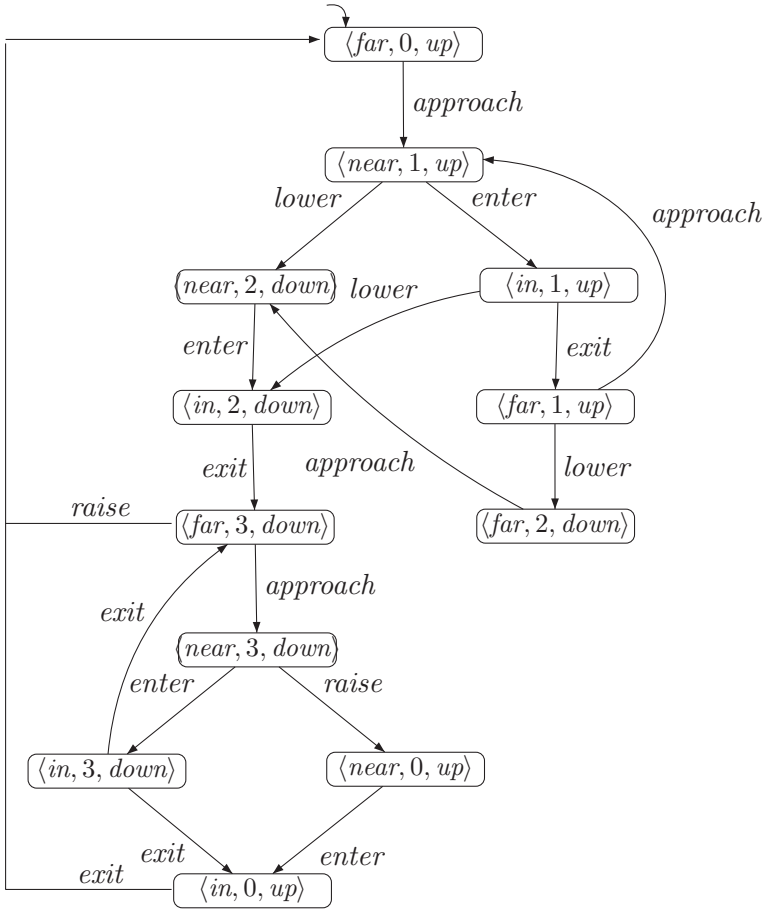


Figure 2.16: Transition system for the railroad crossing.

A channel c has a (finite or infinite) *capacity* indicating the maximum number of messages it can store, and a type (or *domain*) specifying the type of messages that can be transmitted over c. Each channel c has a capacity $cap(c) \in \mathbb{N} \cup \{\infty\}$, and a domain $dom(c)$. For a channel c that can only transfer bits, $dom(c) = \{0, 1\}$. If complete texts (of maximum length of 200, say) need to be transmitted over channel c, then another type of channel has to be used such that $dom(c) = \Sigma^{\leqslant 200}$, where Σ is the alphabet that forms the basis of the texts, e.g., Σ is the set of all letters and special characters used in German texts.

The capacity of a channel defines the size of the corresponding buffer, i.e., the number of messages not yet read that can be stored in the buffer. When $cap(c) \in \mathbb{N}$, c is a channel with finite capacity; $cap(c) = \infty$ indicates that c has an infinite capacity. Note that the special case $cap(c) = 0$ is permitted. In this case, channel c has *no* buffer. Communication via such a channel c corresponds to handshaking (simultaneous transmission and receipt, i.e., synchronous message passing) plus the exchange of some data. When $cap(c) > 0$, there is a "delay" between the transmission and the receipt of a message, i.e., sending and reading of the same message take place at different moments. This is called asynchronous message passing. Sending and reading a message from a channel with a nonzero capacity can never appear simultaneously. By means of channel systems, both synchronous and asynchronous message passing can thus be modeled.

Definition 2.31. Channel System

A program graph over $(Var, Chan)$ is a tuple

$$PG = (Loc, Act, Effect, \hookrightarrow, Loc_0, g_0)$$

according to Definition 2.13 (page 32) with the only difference that

$$\hookrightarrow \quad \subseteq \quad Loc \times (Cond(Var) \times (Act \cup Comm) \times Loc.$$

A *channel system* CS over $(Var, Chan)$ consists of program graphs PG_i over $(Var_i, Chan)$ (for $1 \leqslant i \leqslant n$) with $Var = \bigcup_{1 \leqslant i \leqslant n} Var_i$. We denote

$$CS \;=\; [PG_1 \mid \ldots \mid PG_n] \;.$$

■

The transition relation $\hookrightarrow$ of a program graph over $(Var, Chan)$ consists of two types of conditional transitions. As before, conditional transitions $\ell \overset{g:\alpha}{\hookrightarrow} \ell'$ are labeled with guards and actions. These conditional transitions can happen whenever the guard holds. Alternatively, conditional transitions may be labeled with communication actions. This yields conditional transitions of type $\ell \overset{g:c!v}{\hookrightarrow} \ell'$ (for sending v along c) and $\ell \overset{g:c?x}{\hookrightarrow} \ell'$ (for receiving a message along c). When can such conditional transitions happen? Stated

differently, when are these conditional transitions executable? This depends on the current variable evaluation and the capacity and content of the channel c. For the sake of simplicity assume in the following that the guard is satisfied.

- *Handshaking.* If $cap(c) = 0$, then process P_i can transmit a value v over channel c by performing
$$\ell_i \overset{c!v}{\hookrightarrow} \ell_i'$$
only if another process P_j, say, "offers" a complementary receive action, i.e., can perform
$$\ell_j \overset{c?x}{\hookrightarrow} \ell_j'.$$
P_i and P_j should thus be able to perform $c!v$ (in P_i) and $c?x$ (in P_j) simultaneously. Then, message passing can take place between P_i and P_j. The effect of message passing corresponds to the (distributed) assignment $x := v$.

  Note that when handshaking is only used for synchronization purposes and not for data transfer, the name of the channel as well as the value v are not of any relevance.

- *Asynchronous message passing.* If $cap(c) > 0$, then process P_i can perform the conditional transition
$$\ell_i \overset{c!v}{\hookrightarrow} \ell_i'$$
if and only if channel c is not full, i.e., if less than $cap(c)$ messages are stored in c. In this case, v is stored at the rear of the buffer c. Channels are thus considered as first-in, first-out buffers. Accordingly, P_j may perform
$$\ell_j \overset{c?x}{\hookrightarrow} \ell_j'$$
if and only if the buffer of c is not empty. In this case, the first element v of the buffer is extracted and assigned to x (in an atomic manner). This is summarized in Table 2.1.

	executable if ...	effect
$c!v$	c is not "full"	$\mathit{Enqueue}(c, v)$
$c?x$	c is not empty	$\langle x := \mathit{Front}(c)\,;\ \mathit{Dequeue}(c)\rangle$;

Table 2.1: Enabledness and effect of communication actions if $cap(c) > 0$.

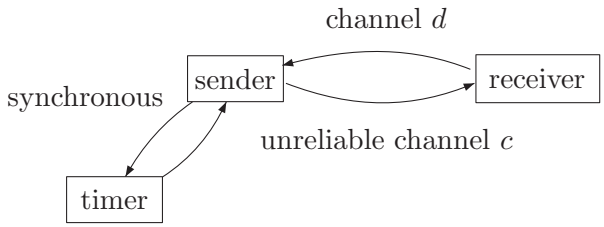


Figure 2.17: Schematic view of the alternating bit protocol.

Channel systems are often used to model communication protocols. One of the most elementary and well-known protocols is the alternating bit protocol.

Example 2.32. Alternating Bit Protocol (ABP)

Consider a system essentially consisting of a sender S and a receiver R that communicate with each other over channels c and d, see Figure 2.17. It is asssumed that both channels have an unlimited buffer, i.e., $cap(c) = cap(d) = \infty$. Channel c is unreliable in the sense that data may get lost when being transmitted from the sender S to channel c. Once messages are stored in the buffer of channel c, they are neither corrupted nor lost. Channel d is assumed to be perfect. The goal is to design a communication protocol that ensures any *distinct* transmitted datum by S to be delivered to R. To ensure this in the presence of possible message losses, sender S resorts to retransmissions. Messages are transmitted one by one, i.e., S starts sending a new message once the transmission of the previous message has been successful. This is a simple flow control principle, known as "send-and-wait".

We abstract from the real activities of S and R and, instead, concentrate on a simplified representation of the communication structure of the system. S sends the successive messages $m_0, m_1, \ldots$ together with control bits $b_0, b_1, \ldots$ over channel c to R. Transmitted messages are thus pairs:

$$\langle m_0, 0\rangle, \langle m_1, 1\rangle, \langle m_2, 0\rangle, \langle m_3, 1\rangle, \ldots$$

On receipt of $\langle m, b\rangle$ (along channel c), R sends an acknowledgment (ack) consisting of the control bit b just received. On receipt of ack $\langle b\rangle$, S transmits a new message m' with control bit $\neg b$. If, however, S has to wait "too long" for the ack, it timeouts and retransmits $\langle m, b\rangle$. The program graphs for S and R are sketched in Figure 2.18 and 2.19. For simplicity, the data that is transmitted is indicated by m instead of m_i.

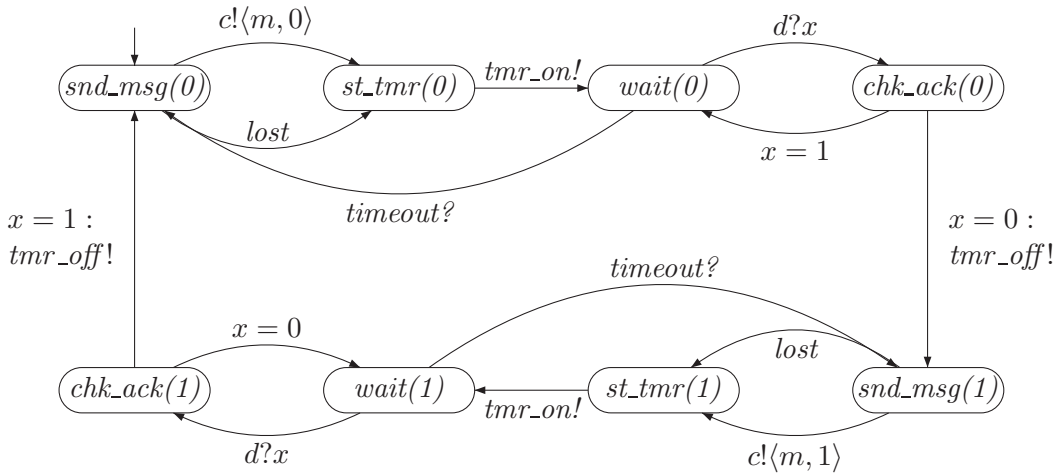


Figure 2.18: Program graph of ABP sender S.

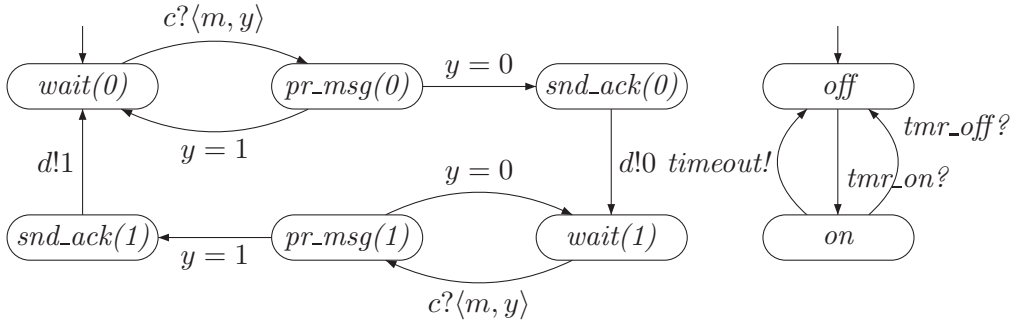


Figure 2.19: Program graph of (left) ABP receiver R and (right) *Timer*.

Control bit b—also called the alternating bit—is thus used to distinguish retransmissions of m from transmissions of subsequent (and previous) messages. Due to the fact that the transmission of a new datum is initiated only when the last datum has been received correctly (and this is acked), a single bit is sufficient for this purpose and notions like, e.g., sequence numbers, are not needed.

The timeout mechanism of S is modeled by a *Timer* process. S activates this timer on sending a message (along c), and it stops the timer on receipt of an ack. When raising a timeout, the timer signals to S that a retransmission should be initiated. (Note that due to this way of modeling, so-called premature timeouts may occur, i.e., a timeout may occur whereas an ack is still on its way to S.) The communication between the timer and S is modeled by means of handshaking, i.e., by means of channels with capacity 0.

The complete system can now be represented as the following channel system over $Chan = \{ c, d, tmr\_on, tmr\_off, timeout \}$ and $Var = \{ x, y, m_i \}$:

$$ABP \;=\; [S \mid Timer \mid R\,]\;.$$

■

The following definition formalizes the successive behavior of a channel system by means of a transition system. The basic concept is similar to the mapping from a program graph onto a transition system. Let $CS = [PG_1 \mid \ldots \mid PG_n]$ be a channel system over $(Chan, Var)$. The (global) states of $TS(CS)$ are tuples of the form

$$\langle \ell_1, \ldots, \ell_n, \eta, \xi \rangle$$

where ℓ_i indicates the current location of component PG_i, η keeps track of the current values of the variables, and ξ records the current content of the various channels (as sequences). Formally, $\eta \in Eval(Var)$ is an evaluation of the variables (as we have encountered before), and ξ is a *channel evaluation*, i.e., a mapping from channel $c \in Chan$ onto a sequence $\xi(c) \in dom(c)^*$ such that the length of the sequence cannot exceed the capacity of c, i.e., $len(\xi(c)) \leqslant cap(c)$ where $len(\cdot)$ denotes the length of a sequence. $Eval(Chan)$ denotes the set of all channel evaluations. For initial states, the control components $\ell_i \in Loc_{0,i}$ must be initial and variable evaluation η must satisfy the initial condition g_0. In addition, every channel is initially assumed to be empty, denoted ε.

Before providing the details of the semantics of a transition system, let us introduce some notations. Channel evaluation $\xi(c) = v_1\, v_2\, \ldots v_k$ (where $cap(c) \geqslant k$) denotes that v_1 is at the front of the buffer of c, v_2 is the second element, etc., and v_k is the element at the rear of c. $len(\xi(c)) = k$ in this case. Let $\xi[c := v_1, \ldots, v_k]$ denote the channel evaluation where sequence $v_1, \ldots, v_k$ is assigned to c and all other channels are unaffected, i.e.,

$$\xi[c := v_1 \ldots v_k](c') \;=\; \begin{cases} \xi(c') & \text{if } c \neq c' \\ v_1 \ldots v_k & \text{if } c = c'. \end{cases}$$

The channel evaluation ξ_0 maps any channel to the empty sequence, denoted ε, i.e., $\xi_0(c) = \varepsilon$ for any channel c. Let $len(\varepsilon) = 0$. The set of actions of $TS(CS)$ consists of actions $\alpha \in Act_i$ of component PG_i and the distinguished symbol τ representing all communication actions in which data is exchanged.

Definition 2.33. Transition System Semantics of a Channel System

Let $CS = [PG_1 \mid \ldots \mid PG_n]$ be a channel system over $(Chan, Var)$ with

$$PG_i = (Loc_i, Act_i, Effect_i, \hookrightarrow_i, Loc_{0,i}, g_{0,i})\,, \qquad \text{for } 0 < i \leqslant n.$$

The transition system of CS, denoted $TS(CS)$, is the tuple $(S, Act, \rightarrow, I, AP, L)$ where:

- $S \;=\; (Loc_1 \times \ldots \times Loc_n) \times Eval(Var) \times Eval(Chan)$,
- $Act \;=\; \biguplus_{0<i\leqslant n} Act_i \;\uplus\; \{\,\tau\,\}$,
- $\rightarrow$ is defined by the rules of Figure 2.20 (page 61),
- $I \;=\; \Big\{\; \langle \ell_1, \ldots, \ell_n, \eta, \xi_0 \rangle \;\mid\; \forall 0 < i \leqslant n.\; (\ell_i \in Loc_{0,i} \;\wedge\; \eta \models g_{0,i}) \;\Big\}$,
- $AP \;= \biguplus_{0<i\leqslant n} Loc_i \;\uplus\; Cond(Var)$,
- $L(\langle \ell_1, \ldots, \ell_n, \eta, \xi \rangle) \;=\; \{\,\ell_1, \ldots, \ell_n\,\} \;\cup\; \{\, g \in Cond(Var) \mid \eta \models g \,\}$.

■

This definition is a formalization of the informal description of the interpretation of a channel system given before. Note that the labeling of the atomic propositions is similar to that for program graphs (see Definition 2.15). For the sake of simplicity, the above definition does not allow for propositions on channels. This could be accommodated by allowing for conditions on channels such as, e.g., "the channel c is empty" or "the channel c is full", and checking these conditions on the channel evaluation ξ in a state.

Example 2.34. Alternating Bit Protocol (Revisited)

Consider the alternating bit protocol that was modeled as a channel system in Example 2.32. The underlying transition system $TS(ABP)$ has, despite various simplifying assumptions, infinitely many states. This is, e.g., due to the fact that the timer may signal a timeout on each transmission of a datum by S resulting in infinitely many messages in channel c.

To clarify the functionality of the alternating bit protocol, consider two execution fragments represented by indicating the states of the various components (sender S, receiver R, the timer, and the contents of channels c and d). The first execution fragment shows the loss of a message. Here, R does not execute any action at all as it only acts if channel c contains at least one message:

sender S	timer	receiver R	channel c	channel d	event
$snd\_msg(0)$	*off*	$wait(0)$	$\varnothing$	$\varnothing$	
$st\_tmr(0)$	*off*	$wait(0)$	$\varnothing$	$\varnothing$	loss of message
$wait(0)$	*on*	$wait(0)$	$\varnothing$	$\varnothing$	
$snd\_msg(0)$	*off*	$wait(0)$	$\varnothing$	$\varnothing$	timeout
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

- interleaving for $\alpha \in Act_i$:

$$\frac{\ell_i \overset{g:\alpha}{\hookrightarrow} \ell_i' \quad \wedge \quad \eta \models g}{\langle \ell_1, \ldots, \ell_i, \ldots, \ell_n, \eta, \xi \rangle \xrightarrow{\alpha} \langle \ell_1, \ldots, \ell_i', \ldots, \ell_n, \eta', \xi \rangle}$$

  where $\eta' = \mathit{Effect}(\alpha, \eta)$.

- asynchronous message passing for $c \in \mathit{Chan}$, $\mathit{cap}(c) > 0$:
  - receive a value along channel c and assign it to variable x:

$$\frac{\ell_i \overset{g:c?x}{\hookrightarrow} \ell_i' \ \wedge \ \eta \models g \ \wedge \ \mathit{len}(\xi(c)) = k > 0 \ \wedge \ \xi(c) \ = \ v_1 \ldots v_k}{\langle \ell_1, \ldots, \ell_i, \ldots, \ell_n, \eta, \xi \rangle \xrightarrow{\tau} \langle \ell_1, \ldots, \ell_i', \ldots, \ell_n, \eta', \xi' \rangle}$$

    where $\eta' = \eta[x := v_1]$ and $\xi' = \xi[c := v_2 \ldots v_k]$.
  - transmit value $v \in \mathit{dom}(c)$ over channel c:

$$\frac{\ell_i \overset{g:c!v}{\hookrightarrow} \ell_i' \ \wedge \ \eta \models g \ \wedge \ \mathit{len}(\xi(c)) = k < \mathit{cap}(c) \ \wedge \ \xi(c) \ = \ v_1 \ldots v_k}{\langle \ell_1, \ldots, \ell_i, \ldots, \ell_n, \eta, \xi \rangle \xrightarrow{\tau} \langle \ell_1, \ldots, \ell_i', \ldots, \ell_n, \eta, \xi' \rangle}$$

    where $\xi' = \xi[c := v_1\, v_2 \ldots v_k\, v]$.

- synchronous message passing over $c \in \mathit{Chan}$, $\mathit{cap}(c) = 0$:

$$\frac{\ell_i \overset{g_1:c?x}{\hookrightarrow} \ell_i' \ \wedge \ \eta \models g_1 \ \wedge \ \eta \models g_2 \ \wedge \ \ell_j \overset{g_2:c!v}{\hookrightarrow} \ell_j' \ \wedge \ i \neq j}{\langle \ell_1, \ldots, \ell_i, \ldots, \ell_j, \ldots, \ell_n, \eta, \xi \rangle \xrightarrow{\tau} \langle \ell_1, \ldots, \ell_i', \ldots, \ell_j', \ldots, \ell_n, \eta', \xi \rangle}$$

  where $\eta' = \eta[x := v]$.

Figure 2.20: Rules for the transition relation of a channel system.

When the receiver R is in location *wait*(0) and receives a message, it anticipates receiving a message with either control bit 0 (as it expects) or with control bit 1, see left side of Figure 2.19. Symmetrically, in location *wait*(1) also the possibility is taken into account to receive the (unexpected) ack with control bit 0. The following execution fragment indicates why this unexpected possibility is essential to take into consideration. In a nutshell, the execution fragment shows that it may happen that R receives $\langle m, 0\rangle$, notifies this by means of sending an ack (with control bit 0), and switches to "mode 1"—waiting to receive a message with control bit 1. In the meanwhile, however, sender S has initiated a retransmission of $\langle m, 0\rangle$ (as it timed out). On receipt of this (unexpected) message, receiver R should act accordingly and ignore the message. This is exactly what happens. Note that if this possibility would not have been taken into consideration in the program graph of R, the system would have come to a halt.

sender S	timer	receiver R	channel c	channel d	event
snd_msg(0)	*off*	*wait*(0)	$\varnothing$	$\varnothing$	
st_tmr(0)	*off*	*wait*(0)	$\langle m, 0\rangle$	$\varnothing$	message with bit 0 transmitted
wait(0)	*on*	*wait*(0)	$\langle m, 0\rangle$	$\varnothing$	
snd_msg(0)	*off*	*wait*(0)	$\langle m, 0\rangle$	$\varnothing$	timeout
st_tmr(0)	*off*	*wait*(0)	$\langle m, 0\rangle\ \langle m, 0\rangle$	$\varnothing$	retransmission
st_tmr(0)	*off*	*pr_msg*(0)	$\langle m, 0\rangle$	$\varnothing$	receiver reads first message
st_tmr(0)	*off*	*snd_ack*(0)	$\langle m, 0\rangle$	$\varnothing$	
st_tmr(0)	*off*	*wait*(1)	$\langle m, 0\rangle$	0	receiver changes into mode-1
st_tmr(0)	*off*	*pr_msg*(1)	$\varnothing$	0	receiver reads retransmission
st_tmr(0)	*off*	*wait*(1)	$\varnothing$	0	and ignores it
⋮	⋮	⋮	⋮	⋮	

We conclude this example by pointing out a possible simplification of the program graph of the sender S. Since the transmission of acks (over channel d) is reliable, it is unnecessary (but not wrong) for S to verify the control bit of the ack in location *chk_ack*$(\cdot)$. If S is in location *wait*(0) and channel d is not empty, then the (first) message in d corresponds to the expected ack 0, since R acknowledges each message m exactly once regardless of how many times m is received. Therefore, the program graph of S could be simplified such that by the action sequence $d?x$; *timer_off*, it moves from location *wait*(0) to location *gen_msg*(1). Location *chk_ack*(0) may thus be omitted. By similar arguments, location *chk_ack*(1) may be omitted. If, however, channel d would be unreliable (like channel c), these locations are necessary. ■

Remark 2.35. Open Channel Systems

The rule for synchronous message passing is subject to the idea that there is a closed channel system that does not communicate with the environment. To model open channel systems, only the rule for handshaking has to be modified. If there is a channel c with $cap(c) = 0$ over which the channel system is communicating with the environment, the rules

$$\frac{\ell_i \overset{c!v}{\hookrightarrow} \ell_i'}{\langle \ell_1, \ldots, \ell_i, \ldots, \ell_n, \eta, \xi \rangle \xrightarrow{c!v} \langle \ell_1, \ldots, \ell_i', \ldots, \ell_n, \eta, \xi \rangle}$$

and

$$\frac{\ell_i \overset{c?x}{\hookrightarrow} \ell_i' \quad \wedge \quad v \in dom(c)}{\langle \ell_1, \ldots, \ell_i, \ldots, \ell_n, \eta, \xi \rangle \xrightarrow{c?x} \langle \ell_1, \ldots, \ell_i', \ldots, \ell_n, \eta[x := v], \xi \rangle}$$

have to be used. The receipt of value v for variable x along channel c is modeled by means of nondeterminism that is resolved by the environment. That is to say, the environment selects value $v \in dom(c)$ in a purely nondeterministic way. ■

2.2.5 NanoPromela

The concepts that have been discussed in the previous sections (program graphs, parallel composition, channel systems) provide the mathematical basis for modeling reactive systems. However, for building automated tools for verifying reactive systems, one aims at simpler formalisms to specify the behavior of the system to be analyzed. On the one hand, such specification languages should be simple and easy to understand, such that nonexperts also are able to use them. On the other hand, they should be expressive enough to formalize the stepwise behavior of the processes and their interactions. Furthermore, they have to be equipped with a *formal semantics* which renders the intuitive meaning of the language commands in an unambiguous manner. In our case, the purpose of the formal semantics is to assign to each program of the specification language a transition system that can serve as a basis for the automated analysis, e.g., simulation or model checking against temporal logical specifications.

In this section, we present the core features of the language Promela, the input language for the prominent model checker SPIN by Holzmann [209]. Promela is short for "process metalanguage". Promela programs $\overline{\mathcal{P}}$ consist of a finite number of processes $\mathcal{P}_1, \ldots, \mathcal{P}_n$ to be executed concurrently. Promela supports communication over shared variables and message passing along either synchronous or buffered FIFO-channels. The formal semantics of a Promela-program can be provided by means of a *channel system*, which then can be unfolded into a transition system, as explained in Section 2.2.4. The stepwise behavior of the processes $\mathcal{P}_i$ is specified in Promela using a *guarded command language*

[130, 18] with several features of classical imperative programming languages (variable assignments, conditional and repetitive commands, sequential composition), communication actions where processes may send and receive messages from the channels, and atomic regions that avoid undesired interleavings. Guarded commands have already been used as labels for the edges of program graphs and channel systems. They consist of a condition (guard) and an action. Promela does not use action names, but specifies the effect of actions by statements of the guarded command language.

Syntax of nanoPromela We now explain the syntax and semantics of a fragment of Promela, called *nanoPromela*, which concentrates on the basic elements of Promela, but abstracts from details like variable declarations and skips several "advanced" concepts like abstract data types (arrays, lists, etc.) or dynamic process creation. A nanoPromela *program* consists of *statements* representing the stepwise behavior of the processes $\mathcal{P}_1, \ldots,$ $\mathcal{P}_n$ together with a Boolean condition on the initial values of the program variables. We write nanoPromela programs as:

$$\overline{\mathcal{P}} \;=\; [\mathcal{P}_1 | \ldots | \mathcal{P}_n].$$

The main ingredients of the statements that formalize the stepwise behavior of the processes $\mathcal{P}_i$ are the atomic commands `skip`, variable assignments $x := \mathsf{expr}$, communication actions $c?x$ (reading a value for variable x from channel c) and $c!\mathsf{expr}$ (sending the current value of an expression over channel c), conditional commands (if-then-else) and (while)loops. Instead of the standard if-then-else constructs or whileloops, nanoPromela supports nondeterministic choices and allows specifying a finite number of guarded commands in conditional and repetitive commands.

Variables, Expressions and Boolean Expressions The variables in a nanoPromela program $\overline{\mathcal{P}}$ may be typed (integer, Boolean, char, real, etc.) and either global or local to some process of $\mathcal{P}_i$. Similarly, data domains have to be specified for the channels and they have to be declared to be synchronous or fifo-channels of a predefined capacity. We skip the details of variable and channel declarations, as they are irrelevant for the purposes of this chapter. As local variables can be renamed in case they occur in more than one process or as the name of a global variable, we may treat all variables as global variables. Thus, we assume that *Var* is a set of variables occurring in $\overline{\mathcal{P}}$ and that for any a variable name x the domain (type) of x is given as the set $dom(x)$. Furthermore, in the declaration part of a Promela program, the type of a channel is specified. We simply write here $dom(c)$ for the type (domain) of channel c and $cap(c)$ for its capacity. In addition, we assume that the variable declaration of program $\overline{\mathcal{P}}$ contains a Boolean expression that specifies the legal initial values for the variables $x \in Var$.

$$
\begin{array}{rcl}
\mathsf{stmt} & ::= & \texttt{skip} \mid x := \mathsf{expr} \mid c?x \mid c!\mathsf{expr} \mid \\
& & \mathsf{stmt}_1 \,;\, \mathsf{stmt}_2 \mid \texttt{atomic}\{\mathsf{assignments}\} \mid \\
& & \textbf{if} \quad :: g_1 \Rightarrow \mathsf{stmt}_1 \quad \ldots \quad :: g_n \Rightarrow \mathsf{stmt}_n \quad \textbf{fi} \quad \mid \\
& & \textbf{do} \quad :: g_1 \Rightarrow \mathsf{stmt}_1 \quad \ldots \quad :: g_n \Rightarrow \mathsf{stmt}_n \quad \textbf{do}
\end{array}
$$

Figure 2.21: Syntax of nanoPromela-statements.

The intuitive meaning of an assignment $x := \mathsf{expr}$ is obvious: variable x is assigned the value of the expression expr given the current variable evaluation. The precise syntax of the expressions and Boolean expressions is not of importance here. We may assume that the expressions used in assignments for variable x are built by constants in $dom(x)$, variables y of the same type as x (or a subtype of x), and operators on $dom(x)$, such as Boolean connectives $\wedge$, $\vee$, $\neg$, etc. for $dom(x) = \{0, 1\}$ and arithmetic operators $+$, $*$, etc. for $dom(x) = \mathbb{R}$.[3] The guards are Boolean expressions that impose conditions on the values of the variables, i.e., we treat the guards as elements of $Cond(Var)$.

Statements The syntax of the *statements* that specify the behavior of the nanoPromela-processes is shown in Figure 2.21 on page 65. Here, x is a variable in Var, expr an expression, and c is a channel of arbitrary capacity. Type consistency of the variable x and the expression expr is required in assignments $x := \mathsf{expr}$. Similarly, for the message-passing actions $c?x$ and $c!\mathsf{expr}$ we require that $dom(c) \subseteq dom(x)$ and that the type of expr corresponds to the domain of c. The g_i's in **if**–**fi**- and **do**–**od**-statements are *guards*. As mentioned above, we assume that $g_i \in Cond(Var)$. The body $\mathsf{assignments}$ of an atomic region is a nonempty sequential composition of assignments, i.e., $\mathsf{assignments}$ has the form

$$x_1 := \mathsf{expr}_1 \,;\, x_2 := \mathsf{expr}_2 \,;\, \ldots \,;\, x_m := \mathsf{expr}_m$$

where $m \geqslant 1$, $x_1, \ldots, x_m$ are variables and $\mathsf{expr}_1, \ldots, \mathsf{expr}_m$ expressions such that the types of x_i and expr_i are compatible.

Intuitive Meaning of the Commands Before presenting the formal semantics, let us give some informal explanations on the meaning of the commands. `skip` stands for a process that terminates in one step, without affecting the values of the variables or contents of the channels. The meaning of assignments is obvious. $\mathsf{stmt}_1 \,;\, \mathsf{stmt}_2$ denotes sequential composition, i.e., stmt_1 is executed first and after its termination stmt_2 is executed. The concept of *atomic regions* is realized in nanoPromela by statements of the

[3] For simplicity, the operators are supposed to be total. For operators that require special arguments (e.g., division requires a nonzero second argument) we assume that the corresponding domain contains a special element with the meaning of "undefined".

form `atomic{stmt}`. The effect of atomic regions is that the execution of stmt is treated as an atomic step that cannot be interleaved with the activities of other processes. As a side effect atomic regions can also serve as a compactification technique that compresses the state space by ignoring the intermediate configurations that are passed during the executions of the commands inside an atomic region. The assumption that the body of an atomic region consists of a sequence of assignments will simplify the inference rules given below.

Statements build by **if**–**fi** or **do**–**od** are generalizations of standard if-then-else commands and whileloops. Let us first explain the intuitive meaning of conditional commands. The statement

$$\textbf{if } :: g_1 \Rightarrow \mathsf{stmt}_1 \ \ldots \ :: g_n \Rightarrow \mathsf{stmt}_n \textbf{ fi}$$

stands for a nondeterministic choice between the statements stmt_i for which the guard g_i is satisfied in the current state, i.e., g_i holds for the current valuation of the variables. We assume a *test-and-set semantics* where the evaluation of the guards, the choice between the enabled guarded commands and the execution of the first atomic step of the selected statement, are performed as an atomic unit that cannot be interleaved with the actions of concurrent processes. If none of the guards $g_1, \ldots, g_n$ is fulfilled in the current state, then the **if**–**fi**–command *blocks*. Blocking has to be seen in the context of the other processes that run in parallel and that might abolish the blocking by changing the values of shared variables such that one or more of the guards may eventually evaluate to true. For instance, the process given by the statement

$$\textbf{if } :: y > 0 \Rightarrow x := 42 \textbf{ fi}$$

in a state where y has the value 0 waits until another process assigns a nonzero value to y. Standard if-then-else commands, say "**if** g **then** stmt_1 **else** stmt_2 **fi**", of imperative programming languages can be obtained by:

$$\textbf{if } :: g \Rightarrow \mathsf{stmt}_1 \ :: \neg g \Rightarrow \mathsf{stmt}_2 \textbf{ fi},$$

while statements "**if** g **then** stmt_1 **fi**" without an else option are modeled by:

$$\textbf{if } :: g \Rightarrow \mathsf{stmt}_1 \ :: \neg g \Rightarrow \mathtt{skip} \textbf{ fi}.$$

In a similar way, the **do**–**od**-command generalizes whileloops. These specify the repetition of the body, as long as one of the guards is fulfilled. That is, statements of the form:

$$\textbf{do } :: g_1 \Rightarrow \mathsf{stmt}_1 \ \ldots \ :: g_n \Rightarrow \mathsf{stmt}_n \textbf{ od}$$

stand for the iterative execution of the nondeterministic choice among the guarded commands $g_i \Rightarrow \mathsf{stmt}_i$ where guard g_i holds in the current configuration. Unlike conditional commands, **do**–**od**-loops do not block in a state if all guards are violated. Instead, if

$g_1, \ldots, g_n$ do not hold in the current state, then the whileloop is aborted. In fact, a single guarded loop **do** :: $g \Rightarrow$ stmt **od** has the same effect as an ordinary whileloop "**while** g **do** stmt **od**" with body stmt and termination condition $\neg g$. (As opposed to Promela, loops are not terminated by the special command "break".)

Example 2.36. Peterson's Mutual Exclusion Algorithm

Peterson's mutual exclusion algorithm for two processes (see Example 2.25 on page 45) can be specified in nanoPromela as follows. We deal with two Boolean variables b_1 and b_2 and the variable x with domain $dom(x) = \{1, 2\}$ and two Boolean variables $crit_1$ and $crit_2$. The activities of the processes inside their noncritical sections are modeled by the action `skip`. For the critical section, we use the assignment $crit_i :=$ true. Initially, we have $b_1 = b_2 = crit_1 = crit_2 =$ false, while x is arbitrary. Then the nanoPromela-code of process $\mathcal{P}_1$ is given by the statement

do :: true $\Rightarrow$ `skip`;
`atomic`$\{b_1 :=$ true$; x := 2\}$;
if :: $(x = 1) \vee \neg b_2 \Rightarrow crit_1 :=$ true **fi**
`atomic`$\{crit_1 :=$ false$; b_1 :=$ false$\}$
od

The statement for modeling the second process is similar. The infinite repetition of the three phases "noncritical section", "waiting phase" and "critical section" is modeled by the **do**–**od**-loop with the trivial guard true. The request action corresponds to the statement `atomic`$\{b_1 :=$ true$; x := 2\}$ and the release action to the statement `atomic`$\{crit_1 :=$ false$; b_1 :=$ false$\}$. The waiting phase where process $\mathcal{P}_1$ has to await until $x = 1$ or $b_2 =$ false is modeled by the **if**–**fi**-command.

The use of atomic regions is not necessary, but serves here as a compactification technique. As we mentioned in Example 2.25, the request action can also be split into the two assignments $b_1 :=$ true and $x := 2$. As long as both assignments are inside an atomic region the order of the assignments $b_1 :=$ true and $x := 2$ is irrelevant. If, however, we drop the atomic region, then we have to use the order $b_1 :=$ true; $x := 2$, as otherwise the mutual exclusion property cannot be ensured. That is, the process

do :: true $\Rightarrow$ `skip`;
$x := 2$;
$b_1 :=$ true;
if :: $(x = 1) \vee \neg b_2 \Rightarrow crit_1 :=$ true **fi**
`atomic`$\{crit_1 :=$ false$; b_1 :=$ false$\}$
od

for $\mathcal{P}_1$ together with the symmetric protocol for $\mathcal{P}_2$ constitutes a nanoPromela program where the mutual exclusion property "never $crit_1 = crit_2 =$ true" cannot be guaranteed. ■

Example 2.37. Vending Machine

In the above example there are no nondeterministic choices caused by a conditional or repetitive command. For an example where nondeterminism arises through simultaneously enabled guarded commands of a loop, consider the beverage vending machine of Example 2.14 (page 33). The following nanoPromela program describes its behavior:

```
do  ::  true ⇒
            skip;
            if      ::  nsoda > 0   ⇒   nsoda := nsoda − 1
                    ::  nbeer > 0   ⇒   nbeer := nbeer − 1
                    ::  nsoda = nbeer = 0 ⇒ skip
            fi
    ::  true ⇒  atomic{nbeer := max; nsoda := max}
od
```

In the starting location, there are two options that are both enabled. The first is the insertion of a coin by the user, modeled by the command `skip`. The first two options of the **if**–**fi**-command represent the cases where the user selects soda or beer, provided some bottles of soda and beer, respectively, are left. The third guarded command in the **if**–**fi** substatement applies to the case where neither soda nor beer is available anymore and the machine automatically returns to the initial state. The second alternative that is enabled in the starting location is the refill action whose effect is specified by the atomic region where the variables *nbeer* and *nsoda* are reset to *max*. ■

Semantics The *operational semantics* of a nanoPromela-statement with variables and channels from $(Var, Chan)$ is given by a *program graph* over $(Var, Chan)$. The program graphs $PG_1, \ldots, PG_n$ for the processes $\mathcal{P}_1, \ldots, \mathcal{P}_n$ of a nanoPromela program $\overline{\mathcal{P}} = [\mathcal{P}_1 | \ldots | \mathcal{P}_n]$ constitute a *channel system* over $(Var, Chan)$. The transition system semantics for channel systems (Definition 2.31 on page 55) then yields a transition system $TS(\overline{\mathcal{P}})$ that formalizes the stepwise behavior of $\overline{\mathcal{P}}$.

The program graph associated with a nanoPromela-statement stmt formalizes the control flow when executing stmt. That is, the substatements play the role of the locations. For modeling termination, a special location exit is used. Roughly speaking, any guarded command $g \Rightarrow$ stmt corresponds to an edge with the label $g : \alpha$ where α stands for the first action of stmt. For example, for the statement

```
cond_cmd  =  if  ::  x > 1   ⇒   y := x + y
                 ::  true    ⇒   x := 0; y := x
             fi
```

from cond_cmd – viewed as a location of a program graph – there are two edges: one with the guard $x > 1$ and action $y := x + y$ leading to exit, and one with the guard true and action $x := 0$ yielding the location for the statement $y := x$. Since $y := x$ is deterministic there is a single edge with guard true, action $y := x$ leading to location exit.

As another example, consider the statement

$$\begin{array}{lcll} \mathsf{loop} & = & \textbf{do} & :: \quad x > 1 \quad \Rightarrow \quad y := x + y \\ & & & :: \quad y < x \quad \Rightarrow \quad x := 0;\ y := x \\ & & \textbf{od} & \end{array}$$

Here, the repetition semantics of the **do**–**od**-loop is modeled by returning the control to stmt whenever the body of the selected alternative has been executed. Thus, from location loop there are three outgoing edges, see Figure 2.22 on page 69. One is labeled with the guard $x > 1$ and action $y := x + y$ and leads back to location loop. The second edge has the guard $y < x$ and action $x := 0$ and leads to the statement $y := x\,;$ loop. The third edge covers the case where the loop terminates. It has the guard $\neg(x > 1) \wedge \neg(y < x)$ and an action without any effect on the variables and leads to location exit.

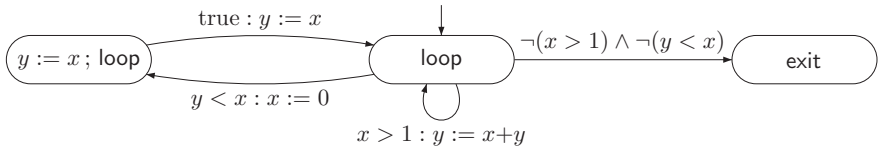


Figure 2.22: Program graph for a loop

The goal is now to formalize the ideas sketched above. We start with a formal definition of substatements of stmt. Intuitively, these are the potential locations of intermediate states during the execution of stmt.

Notation 2.38. Substatement

The set of substatements of a nanoPromela-statement stmt is recursively defined. For statements $\mathsf{stmt} \in \{\mathtt{skip}, x := \mathsf{expr}, c?x, c!\mathsf{expr}\}$ the set of substatements is $sub(\mathsf{stmt}) = \{\mathsf{stmt}, \mathsf{exit}\}$. For sequential composition let

$$sub(\mathsf{stmt}_1\,;\,\mathsf{stmt}_2) \;=$$

$$\{\,\mathsf{stmt}'\,;\,\mathsf{stmt}_2 \mid \mathsf{stmt}' \in sub(\mathsf{stmt}_1) \setminus \{\mathsf{exit}\}\,\} \cup sub(\mathsf{stmt}_2).$$

For conditional commands, the set of substatements is defined as the set consisting of the **if**–**fi**-statement itself and substatements of its guarded commands. That is, for cond_cmd

being **if** $:: g_1 \Rightarrow \mathsf{stmt}_1 \ldots :: g_n \Rightarrow \mathsf{stmt}_n$ **fi** we have

$$sub(\mathsf{cond\_cmd}) \;=\; \{\,\mathsf{cond\_cmd}\,\} \cup \bigcup_{1 \leqslant i \leqslant n} sub(\mathsf{stmt}_i).$$

The substatements of a loop loop given by **do** $:: g_1 \Rightarrow \mathsf{stmt}_1 \ldots :: g_n \Rightarrow \mathsf{stmt}_n$ **od** are defined similarly, but taking into account that control moves back to loop when guarded commands terminate. That is:

$$sub(\mathsf{loop}) \;=$$

$$\{\,\mathsf{loop},\, \mathsf{exit}\,\} \cup \bigcup_{1 \leqslant i \leqslant n} \{\,\mathsf{stmt}'\,;\, \mathsf{loop} \mid \mathsf{stmt}' \in sub(\mathsf{stmt}_i) \setminus \{\mathsf{exit}\}\,\}.$$

For atomic regions let $sub(\mathtt{atomic}\{\mathsf{stmt}\}) \;=\; \{\,\mathtt{atomic}\{\mathsf{stmt}\},\, \mathsf{exit}\,\}$. ■

The definition of $sub(\mathsf{loop})$ relies on the observation that the effect of a loop with a single guarded command, say "**do** $:: g \Rightarrow \mathsf{stmt}$ **od**", corresponds to the effect of

$$\textbf{if } g \textbf{ then } \;\mathsf{stmt}\,;\; \textbf{do } :: g \Rightarrow \mathsf{stmt} \textbf{ od } \;\textbf{ else } \mathtt{skip} \textbf{ fi}$$

An analogous characterization applies to loops with two or more guarded commands. Thus, the definition of the substatements of loop relies on combining the definitions of the sets of substatements for sequential composition and conditional commands.

The formal semantics of nanoPromela program $\overline{\mathcal{P}} = [\mathcal{P}_1 | \ldots | \mathcal{P}_n]$ where the behavior of the process $\mathcal{P}_i$ is specified by a nanoPromela statement is a channel system $[PG_1 | \ldots | PG_n]$ over $(Var, Chan)$ where Var is the set of variables and $Chan$ the set of channels that are declared in $\overline{\mathcal{P}}$. As mentioned before, a formal syntax for the variable and channel declarations will not be provided, and global and local variables will not be distinguished. We assume that the set Var of typed variables and the set $Chan$ of channels (together with a classification of the channels into synchronous and FIFO-channels of some capacity $cap(\cdot)$) are given. Hence, local variable and channel declarations for the processes $\mathcal{P}_i$ are not considered. It is assumed that they are given by a nanoPromela-statement over some fixed tuple $(Var, Chan)$.

We now provide inference rules for the nanoPromela constructs. The inference rules for the atomic commands (skip, assignment, communication actions) and sequential composition, conditional and repetitive commands give rise to the edges of a "large" program graph where the set of locations agrees with the set of nanoPromela-statements. Thus, the edges have the form

$$\mathsf{stmt} \overset{g:\alpha}{\hookrightarrow} \mathsf{stmt}' \quad \text{or} \quad \mathsf{stmt} \xrightarrow{g:comm} \mathsf{stmt}'$$

where stmt is a nanoPromela statement, stmt' a substatement of stmt, and g a guard, α an action, and $comm$ a communication action $c?x$ or $c!\mathsf{expr}$. The subgraph consisting of

the substatements of $\mathcal{P}_i$ then yields the program graph PG_i of process $\mathcal{P}_i$ as a component of the program $\overline{\mathcal{P}}$.

The semantics of the atomic statement `skip` is given by a single axiom formalizing that the execution of `skip` terminates in one step without affecting the variables

$$\frac{}{\texttt{skip} \xrightarrow{\text{true: } id} \mathsf{exit}}$$

where *id* denotes an action that does not change the values of the variables, i.e., $\mathit{Effect}(id, \eta) = \eta$ for all variable evaluations η. Similarly, the execution of a statement consisting of an assignment $x := \mathsf{expr}$ has the trivial guard and terminates in one step:

$$\frac{}{x := \mathsf{expr} \xrightarrow{\text{true} \,:\, \mathrm{assign}(x, \mathsf{expr})} \mathsf{exit}}$$

where $\mathrm{assign}(x, \mathsf{expr})$ denotes the action that changes the value of x according to the assignment $x := \mathsf{expr}$ and does not affect the other variables, i.e., if $\eta \in \mathit{Eval}(\mathit{Var})$ and $y \in \mathit{Var}$ then $\mathit{Effect}(\mathrm{assign}(x, \mathsf{expr}), \eta)(y) = \eta(y)$ if $y \neq x$ and $\mathit{Effect}(\mathrm{assign}(x, \mathsf{expr}), \eta)(x)$ is the value of expr when evaluated over η. For the communication actions $c!\mathsf{expr}$ and $c?x$ the following axioms apply:

$$\frac{}{c?x \xrightarrow{c?x} \mathsf{exit}} \qquad \frac{}{c!\mathsf{expr} \xrightarrow{c!\mathsf{expr}} \mathsf{exit}}$$

The effect of an atomic region $\texttt{atomic}\{x_1 := \mathsf{expr}_1; \ldots; x_m := \mathsf{expr}_m\}$ is the cumulative effect of the assignments $x_i := \mathsf{expr}_i$. It can be defined by the rule:

$$\frac{}{\texttt{atomic}\{x_1 := \mathsf{expr}_1; \ldots; x_m := \mathsf{expr}_m\} \xrightarrow{\text{true} \,:\, \alpha_m} \mathsf{exit}}$$

where $\alpha_0 = id$, $\alpha_i = \mathit{Effect}(\mathrm{assign}(x_i, \mathsf{expr}_i), \mathit{Effect}(\alpha_{i-1}, \eta))$ for $1 \leqslant i \leqslant m$.

Sequential composition $\mathsf{stmt}_1; \mathsf{stmt}_2$ is defined by two rules that distinguish whether or not stmt_1 terminates in one step. If the first step of stmt_1 leads to a location (statement) different from exit, then the following rule applies:

$$\frac{\mathsf{stmt}_1 \overset{g:\alpha}{\hookrightarrow} \mathsf{stmt}_1' \neq \mathsf{exit}}{\mathsf{stmt}_1; \mathsf{stmt}_2 \overset{g:\alpha}{\hookrightarrow} \mathsf{stmt}_1'; \mathsf{stmt}_2}$$

If the computation of stmt_1 terminates in one step by executing action α, then control of $\mathsf{stmt}_1; \mathsf{stmt}_2$ moves to stmt_2 after executing α:

$$\frac{\mathsf{stmt}_1 \overset{g:\alpha}{\hookrightarrow} \mathsf{exit}}{\mathsf{stmt}_1; \mathsf{stmt}_2 \overset{g:\alpha}{\hookrightarrow} \mathsf{stmt}_2}$$

The effect of a conditional command $\mathsf{cond\_cmd} \;=\; \mathbf{if} :: g_1 \Rightarrow \mathsf{stmt}_1 \ldots :: g_n \Rightarrow \mathsf{stmt}_n\, \mathbf{fi}$ is formalized by means of the following rule:

$$\frac{\mathsf{stmt}_i \overset{h:\alpha}{\hookrightarrow} \mathsf{stmt}_i'}{\mathsf{cond\_cmd} \overset{g_i \wedge h:\alpha}{\longrightarrow} \mathsf{stmt}_i'}$$

This rule relies on the test-and-set semantics where choosing one of the enabled guarded commands and performing its first action are treated as an atomic step. The blocking of $\mathsf{cond\_cmd}$ when none of its guards is enabled needs no special treatment. The reason is that $\mathsf{cond\_cmd}$ has no other edges than the ones specified by the rule above. Thus, in a global state $s = \langle \ell_1, \ldots, \ell_n, \eta, \xi \rangle$ where the location ℓ_i of the ith process is $\ell_i = \mathsf{cond\_cmd}$ and all guards $g_1, \ldots, g_n$ evaluate to false, then there is no action of the ith process that is enabled in s. However, actions of other processes might be enabled. Thus, the ith process has to wait until the other processes modify the variables appearing in $g_1, \ldots, g_n$ such that one or more of the guarded commands $g_i \Rightarrow \mathsf{stmt}_i$ become enabled.

For loops, say $\mathsf{loop} \;=\; \mathbf{do} :: g_1 \Rightarrow \mathsf{stmt}_1 \ldots :: g_n \Rightarrow \mathsf{stmt}_n\, \mathbf{od}$, we deal with three rules. The first two rules are similar to the rule for conditional commands, but taking into account that control moves back to loop after the execution of the selected guarded command has been completed. This corresponds to the following rules:

$$\frac{\mathsf{stmt}_i \overset{h:\alpha}{\hookrightarrow} \mathsf{stmt}_i' \neq \mathsf{exit}}{\mathsf{loop} \overset{g_i \wedge h:\alpha}{\longrightarrow} \mathsf{stmt}_i'; \mathsf{loop}} \qquad \frac{\mathsf{stmt}_i \overset{h:\alpha}{\hookrightarrow} \mathsf{exit}}{\mathsf{loop} \overset{g_i \wedge h:\alpha}{\longrightarrow} \mathsf{loop}}$$

If none of the guards $g_1, \ldots, g_n$ holds in the current state then the **do**–**od**-loop will be aborted. This is formalized by the following axiom:

$$\frac{}{\mathsf{loop} \overset{\neg g_1 \wedge \ldots \wedge \neg g_n}{\longrightarrow} \mathsf{exit}}$$

Remark 2.39. Test-and-Set Semantics vs. Two-Step Semantics

The rules for **if**–**fi**- and **do**–**od**-statements formalize the so-called *test-and-set semantics* of guarded commands. This means that evaluating guard g_i and performing the first step of the selected enabled guarded command $g_i \Rightarrow \mathsf{stmt}_i$ are performed atomically. In contrast, SPIN's interpretation of Promela relies on a two-step-semantics where the selection of an enabled guarded command and the execution of its first action are split into two steps. The rule for a conditional command is formalized by the axiom

$$\frac{}{\mathbf{if} :: g_1 \Rightarrow \mathsf{stmt}_1 \ldots :: g_n \Rightarrow \mathsf{stmt}_n\, \mathbf{fi} \overset{g_i \,:\, id}{\longrightarrow} \mathsf{stmt}_i}$$

where *id* is an action symbol for an action that does not affect the variables. Similarly, the first two rules for loops have to be replaced for the two-step semantics by the following rule:

$$\frac{}{\mathsf{loop} \xrightarrow{g_i : id} \mathsf{stmt}_i; \mathsf{loop}}$$

The rule for terminating the loop remains unchanged.

As long as we consider the statements in isolation, the test-and-set semantics and the two-step semantics are equal. However, when running several processes in parallel, the interleaving might cause undesired side effects. For example, consider the semaphore-based solution of the mutual exclusion problem, modeled by a nanoPromela program where the behavior of $\mathcal{P}_i$ is given by the following nanoPromela-statement:

do :: true ⇒ `skip`;
if :: $y > 0 \Rightarrow$ $y := y - 1$;
$crit_i := \text{true}$
fi;
$y := y + 1$
od

The initial value of the semaphore y is zero. Under the two-step semantics the mutual exclusion property is not guaranteed as it allows the processes to verify that the guard $y > 0$ of the **if**–**fi**-statement holds, without decreasing the value of y, and moving control to the assignment $y := y - 1$. But from there the processes can enter their critical sections. However, the protocol works correctly for the test-and-set semantics since then checking $y > 0$ and decreasing y is an atomic step that cannot be interleaved by the actions of the other process. ■

Remark 2.40. Generalized Guards

So far we required that the guards in conditional or repetitive commands consist of Boolean conditions on the program variables. However, it is also often useful to ask for interaction facilities in the guards, e.g., to specify that a process has to wait for a certain input along a FIFO-channel by means of a conditional command **if** :: $c?x \Rightarrow \mathsf{stmt}$ **fi**. The intuitive meaning of the above statement is that the process has to wait until the buffer for c is nonempty. If so, then it first performs the action $c?x$ and then executes stmt. The use of communication actions in the guards leads to a more general class of program graphs with guards consisting of Boolean conditions on the variables or communication actions. For the case of an asynchronous channel the rules in Figure 2.20 on page 61 then have to be extended by:

$$\frac{\ell_i \xrightarrow{c?x:\alpha} \ell_i' \ \wedge \ len(\xi(c)) = k > 0 \ \wedge \ \xi(c) = v_1 v_2 \ldots v_k}{\langle \ldots, \ell_i, \ldots, \eta, \xi \rangle \xrightarrow{\tau} \langle \ldots, \ell_i', \ldots, \eta', \xi' \rangle}$$

where $\eta' = \mathit{Effect}(\alpha, \eta[x := v_1])$ and $\xi[c := v_2 \dots v_k]$, and

$$\frac{\ell_i \xrightarrow{c!v:\alpha} \ell_i' \ \wedge \ \mathit{len}(\xi(c)) = k < \mathit{cap}(c) \ \wedge \ \xi(c) = v_1 \dots v_k}{\langle \dots, \ell_i, \dots, \eta, \xi \rangle \xrightarrow{\tau} \langle \dots, \ell_i', \dots, \eta', \xi' \rangle}$$

where $\eta' = \mathit{Effect}(\alpha, \eta)$ and $\xi[c := v_1 \dots v_k v]$.

Another convenient concept is the special guard **else** which specifies configurations where no other guarded command can be taken. The intuitive semantics of:

$$\begin{array}{lllll} \textbf{if} & :: & g_1 & \Rightarrow & \mathsf{stmt}_1 \\ & & & \vdots & \\ & :: & g_n & \Rightarrow & \mathsf{stmt}_n \\ & :: & \textbf{else} & \Rightarrow & \mathsf{stmt}' \\ \textbf{fi} & & & & \end{array}$$

is that the else option is enabled if none of the guards $g_1, \dots g_n$ evaluates to true. In this case, the execution evolves to a state in which the statement stmt' is to be executed. Here, the g_i's can be Boolean expressions on the variables or communication guards. For example,

$$\textbf{if} :: d?x \Rightarrow \mathsf{stmt} :: \textbf{else} \Rightarrow x := x + 1 \, \textbf{fi}$$

increases x unless a message is obtained from channel d. The else option used in loops leads to nonterminating behaviors. ■

Remark 2.41. Atomic Regions

The axiom for atomic regions yields that if $s = \langle \ell_1 \dots, \ell_n, \eta, \xi \rangle$ is a state in the transition system for the channel system associated with $\overline{\mathcal{P}} = [\mathcal{P}_1 | \dots | \mathcal{P}_n]$ and $\ell_i = \mathtt{atomic}\{x_1 := \mathsf{expr}_1; \dots; x_m := \mathsf{expr}_m\}; \dots$ then in state s the ith process can perform the sequence of assignments $x_1 := \mathsf{expr}_1; \dots; x_m := \mathsf{expr}_m$ in a single transition. With this semantics we abstract from the intermediate states that are passed when having performed the first i assignments ($1 \leqslant i < m$) and avoid that other processes can interleave their activities with these assignments.

This concept can be generalized for atomic regions $\mathtt{atomic}\{\mathsf{stmt}\}$ where the body stmt is an arbitrary statement. The idea is that any terminating execution of stmt is collapsed into a single transition, leading from a state with location stmt to a state with location exit. For this more general approach, the semantic rule for atomic regions operates on execution sequences in the transition system rather than just edges in the program graph. This is not problematic as one could provide the meaning of the statements on the level of transition systems rather than program graph level. However, the semantics is less

obvious for, e.g., infinite executions inside atomic regions, synchronous communication actions inside atomic regions and blocking conditional commands inside atomic regions. One possibility is to insert transitions to a special deadlock state. Another possibility is to work with a semantics that represents also the intermediate steps of an atomic region (but avoids interleaving) and to abort atomic regions as soon as a synchronous communication is required or blocking configurations are reached. ■

As we mentioned in the beginning of the section, Promela provides many more features than nanoPromela, such as atomic regions with more complex statements than sequences of assignments, arrays and other data types, and dynamic process creation. These concepts will not be explained here and we refer to the literature on the model checker SPIN (see, e.g., [209]).

2.2.6 Synchronous Parallelism

When representing asynchronous systems by transition systems, there are no assumptions concerning the relative velocities of the processors on which the components are executed. The residence time of the system in a state and the execution time of the actions are completely ignored. For instance, in the example of the two independent traffic lights (see Example 2.17), no assumption has been made concerning the amount of time a light should stay red or green. The only assumption is that both time periods are finite. The concurrent execution of components is time-abstract.

This is opposed to synchronous systems where components evolve in a lock step fashion. This is a typical computation mechanism in synchronous hardware circuits, for example, where the different components (like adders, inverters, and multiplexers) are connected to a central clock and all perform a (possibly idle) step on each clock pulse. As clock pulses occur periodically with a fixed delay, these pulses may be considered in a discrete manner, and transition systems can be adequately used to describe these synchronous systems. Synchronous composition of two transition systems is defined as follows.

Definition 2.42. Synchronous Product

Let $TS_i = (S_i, Act, \rightarrow_i, I_i, AP_i, L_i)$, i=1, 2, be transition systems with the same set of actions Act. Further, let

$$Act \times Act \rightarrow Act, \quad (\alpha, \beta) \rightarrow \alpha * \beta$$

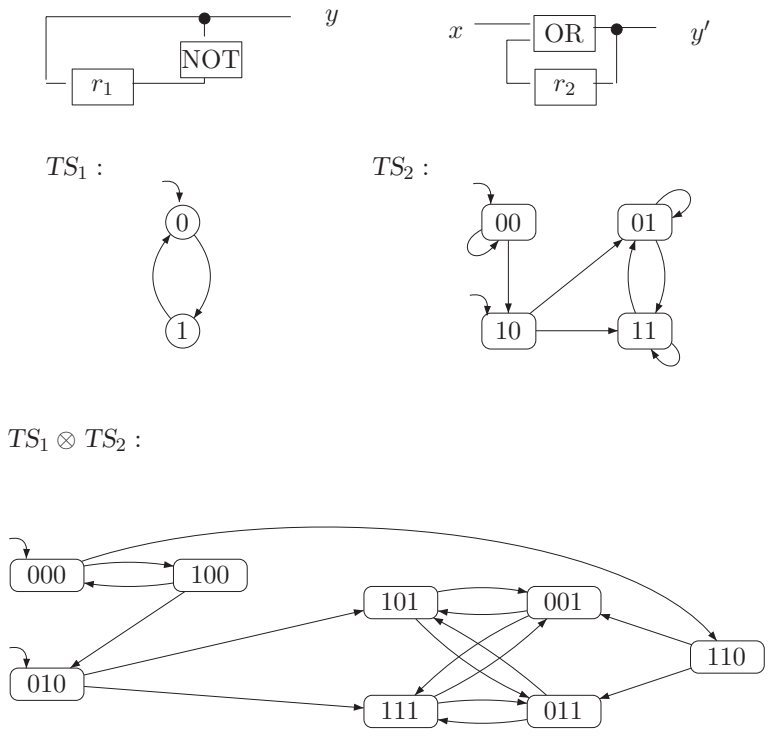


Figure 2.23: Synchronous composition of two hardware circuits.

be a mapping[4] that assigns to each pair of actions α, β, the action name $\alpha * \beta$. The *synchronous product* $TS_1 \otimes TS_2$ is given by:

$$TS_1 \otimes TS_2 \;=\; (S_1 \times S_2, Act, \rightarrow, I_1 \times I_2, AP_1 \cup AP_2, L),$$

where the transition relation is defined by the following rule

$$\frac{s_1 \xrightarrow{\alpha}_1 s_1' \quad \wedge \quad s_2 \xrightarrow{\beta}_2 s_2'}{\langle s_1, s_2 \rangle \xrightarrow{\alpha * \beta} \langle s_1', s_2' \rangle}$$

and the labeling function is defined by: $L(\langle s_1, s_2 \rangle) \;=\; L_1(s_1) \cup L_2(s_2)$. ■

Action $\alpha * \beta$ denotes the synchronous execution of actions α and β. Note that compared to the parallel operator $\|$ where components perform actions in common synchronously, and other action autonomously (i.e., asynchronously), in $TS_1 \otimes TS_2$, both transition systems have to perform all steps synchronously. There are no autonomous transitions of either TS_1 or TS_2.

[4]Operator $*$ is typically assumed to be commutative and associative.

Example 2.43. Synchronous Product of Two Circuits

Let C_1 be a circuit without input variables and with output variable y and register r. The control functions for output and register transitions are

$$\lambda_y \;=\; r_1, \quad \delta_{r_1} \;=\; \neg\, r_1.$$

Circuit C_2 has input variable x', output variable y', and register variable r_2 with the control functions

$$\lambda_{y'} \;=\; \delta_{r_2} \;=\; x' \;\vee\; r_2.$$

The transition system $TS_{C_1} \otimes TS_{C_2}$ is depicted in Figure 2.23 on page 76. Since action names are omitted for transition systems of circuits, action labels for $TS_{C_1} \otimes TS_{C_2}$ are irrelevant. $TS_{C_1} \otimes TS_{C_2}$ is thus the transition system of the circuit with input variable x', output variables y and y', and registers r_1 and r_2, whose control functions are λ_y, $\lambda_{y'}$, δ_{r_1}, and δ_{r_2}. ■

2.3 The State-Space Explosion Problem

The previous two sections have shown that various kinds of systems can be modeled using transition systems. This applies to program graphs representing data-dependent systems, and hardware circuits. Different communication mechanisms can be modeled in terms of appropriate operators on transition systems. This section considers the cardinality of the resulting transition systems, i.e., the number of states in these models. Verification techniques are based on systematically analyzing these transition systems. The runtimes of such verification algorithms are mainly determined by the number of states of the transition system to be analyzed. For many practical systems, the state space may be extremely large, and this is a major limitation for state-space search algorithms such as model checking. Chapter 8, Section 6.7, and Chapter 7 introduce a number of techniques to combat this problem.

Program Graph Representation Transition systems generated by means of "unfolding" a program graph may be extremely large, and in some cases—e.g., if there are infinitely many program locations or variables with infinite domains—even have infinitely many states. Consider a program graph over the set of variables *Var* with $x \in Var$. Recall that states of the unfolded transition system are of the form $\langle \ell, \eta \rangle$ with location ℓ and variable evaluation η. In case all variables in *Var* have a finite domain, like bits, or bounded integers, and the number of locations is finite, the number of states in the transition system is

$$|\, Loc \,| \cdot \prod_{x \in Var} |\, dom(x) \,| \;.$$

The number of states thus grows *exponentially* in the number of variables in the program graph: for N variables with a domain of k possible values, the number of states grows up to k^N. This exponential growth is also known as the *state-space explosion problem*.

It is important to realize that for even simple program graphs with just a small number of variables, this bound may already be rather excessive. For instance, a program graph with ten locations, three Boolean variables and five bounded integers (with domain in $\{0, \ldots, 9\}$) has $10 \cdot 2^3 \cdot 10^5 = 8,000,000$ states. If a single bit array of 50 bits is added to this program graph, for example, this bound grows even to $800{,}000 \cdot 2^{50}$! This observation clearly shows why the verification of data-intensive systems (with many variables or complex domains) is extremely hard. Even if there are only a few variables in a program, the state space that must be analyzed may be very large.

If $dom(x)$ is infinite for some $x \in Var$, as for reals or integers, the underlying transition system has infinitely many states as there are infinitely many values for x. Such program graphs usually yield undecidable verification problems. This is not to say that the verification of all transition systems with an infinite state space is undecidable, however.

It should be remarked that not only the state space of a transition system, but also the number of atomic propositions to represent program graphs (see Definition 2.15, page 34) may in principle be extremely large. Besides, any location, any condition on the variables in the program graph is allowed as an atomic proposition. However, in practice, only a small fragment of the possible atomic propositions is needed. An explicit representation of the labeling function is mostly not necessary, as the truth-values of the atomic formulae are typically derived from the state information. For these reasons, the number of atomic propositions plays only a secondary role.

For sequential hardware circuits (see page 26), states in the transition system are determined by the possible evaluations of the input variables and the registers. The size of the transition system thus grows exponentially in the number of registers and input variables. For N input variables and K registers, the total state space consists of 2^{N+K} states.

Parallelism In all variants of parallel operators for transition systems and program graphs, the state space of the complete system is built as the Cartesian product of the local state spaces S_i of the components. For example, for state space S of transition system

$$TS = TS_1 \,|||\, \ldots \,|||\, TS_n$$

we have $S = S_1 \times \ldots \times S_n$ where S_i denotes the state space of transition system TS_i. The state space of the parallel composition of a system with n states and a system with

k states yields $n{\cdot}k$ states. The total state space S is thus

$$|S_1| \cdot \ldots \cdot |S_n|.$$

The number of states in S is growing (at most) *exponentially* in the number of components: the parallel composition of N components of size k each yields k^N states. Even for small parallel systems this may easily run out of control.

Additionally, the variables (and their domains) represented in the transition system essentially influence the size of the state space. If one of the domains is infinite, then the state space is infinitely large. If the domains are finite, then the size of the state space grows exponentially in the number of variables (as we have seen before for program graphs).

The "exponential blowup" in the number of parallel components and the number of variables explains the enormous size of the state space of practically relevant systems. This observation is known under the heading *state explosion* and is another evidence for the fact that verification problems are particularly space-critical.

Channel Systems For the size of transition systems of channel systems, similar observations can be made as for the representation of program graphs. An important additional component for these systems is the size of the channels, i.e., their capacity. Clearly, if one of these channels has an infinite capacity, this may yield infinitely many states in the transition system. If all channels have finite capacity, however, the number of states is bound in the following way. Let $CS = [PG_1 \,|\ldots|\, PG_n]$ be a channel system over $Var = Var_1 \cup \ldots \cup Var_n$ and channels $Chan$. The state space of CS is of cardinality

$$\prod_{i=1}^{n} |\, PG_i \,| \cdot \prod_{c \in Chan} |\, dom(c) \,|^{cp(c)},$$

which can be rewritten as

$$\prod_{i=1}^{n} \left(|\, Loc_i \,| \cdot \prod_{x \in Var_i} |\, dom(x) \,| \right) \cdot \prod_{c \in Chan} |\, dom(c) \,|^{cp(c)}.$$

For L locations per component, K bit channels of capacity k each, and M variables x with $|\, dom(x) \,| \leqslant m$ totally, the total number of states in the transition system is $L^n{\cdot}m^M{\cdot}2^{K{\cdot}k}$. This is typically enormous.

Example 2.44. State-Space Size of the Alternating Bit Protocol

Consider a variant of the alternating bit protocol (see Example 2.32, page 57) where the

channels c and d have a fixed capacity, 10 say. Recall that along channel d, control bits are sent, and along channel c, data together with a control bit. Let us assume that data items are also simply bits. The timer has two locations, the sender eight, and the receiver six. As there are no further variables, we obtain that the total number of states is $2{\cdot}8{\cdot}6{\cdot}4^{10}{\cdot}2^{10}$, which equals $3{\cdot}2^{35}$. This is around 10^{11} states. ■

2.4 Summary

- Transition systems are a fundamental model for modeling software and hardware systems.
- An execution of a transition system is an alternating sequence of states and actions that starts in an initial state and that cannot be prolonged.
- Interleaving amounts to represent the evolvement of "simultaneous" activities of independent concurrent processes by the nondeterministic choice between these activities.
- In case of shared variable communication, parallel composition on the level of transition systems does not faithfully reflect the system's behavior. Instead, composition on program graphs has to be considered.
- Concurrent processes that communicate via handshaking on the set H of actions execute actions outside H autonomously whereas they execute actions in H synchronously.
- In channel systems, concurrent processes communicate via FIFO-buffers (i.e., channels). Handshaking communication is obtained when channels have capacity 0. For channels with a positive capacity, communication takes place asynchronously—sending and receiving a message takes place at different moments.
- The size of transition system representations grows exponentially in various components, such as the number of variables in a program graph or the number of components in a concurrent system. This is known as the state-space explosion problem.

2.5 Bibliographic Notes

Transition systems. Keller was one of the first researchers that explicitly used transition systems [236] for the verification of concurrent programs. Transition systems are used

as semantical models for a broad range of high-level formalisms for concurrent systems, such as process algebras [45, 57, 203, 298, 299], Petri nets [333], and statecharts [189]. The same is true for hardware synthesis and analysis, in which variants of finite-state automata (Mealy and Moore automata) play a central role; these variants can also be described by transition systems [246]. Program graphs and their unfolding into transition systems have been used extensively by Manna and Pnueli in their monograph(s) on temporal logic verification [283].

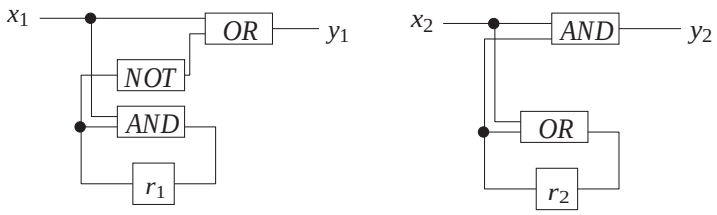
Synchronization paradigms. Shared variable "communication" dates back to the mid-sixties and is due to Dijkstra [126]. He also coined the term *interleaving* in 1971 [128]. Handshaking communication has been the main interaction paradigm in process algebras such as ACP [45], CCS [298, 299], CSP [202, 203], and LOTOS [57]. For a detailed account of process algebra we refer to [46]. The principle of synchronized parallelism has been advocated in Milner's synchronous variant of CCS, SCCS [297], and is used by Arnold to model the interaction between finite transition systems [19]. Synchronous parallelism is also at the heart of Lustre [183], a declarative programming language for reactive systems, and is used in many other hardware-oriented languages.

The interaction between concurrent processes by means of buffers (or channels) has first been considered by Dijkstra [129]. This paradigm has been adopted by specification languages for communication protocols, such as SDL (Specification and Description Language [37]) which is standardized by the ITU. The idea of guarded command languages goes back to Dijkstra [130]. The combination of guarded command languages in combination with channel-based communication is also used in Promela [205], the input language of the model checker SPIN [208]. The recent book by Holzmann [209] gives a detailed account of Promela and SPIN. Structured operational semantics has been introduced by Plotkin [334, 336] in 1981. Its origins are described in [335]. Atomic regions have been first discussed by Lipton [276], Lamport [257] and Owicki [317]. Further details on semantic rules for specification languages for reactive systems can be found, e.g., in [15, 18].

The examples. Most examples that have been provided in this chapter are rather classical. The problem of mutual exclusion was first proposed in 1962 by Dekker together with an algorithm that guarantees two-process mutual exclusion. Dijkstra's solution [126] was the first solution to mutual exclusion for an arbitrary number of processes. He also introduced the concept of semaphores [127] and their use for solving the mutual exclusion problem. A simpler and more elegant solution has been proposed by Lamport in 1977 [256]. This was followed up by Peterson's mutual exclusion protocol [332] in 1981. This algorithm is famous by now due to its beauty and simplicity. For other mutual exclusion algorithms, see e.g. [283, 280]. The alternating bit protocol stems from 1969 and is one of the first flow control protocols [34]. Holzmann gives a historical account of this and related protocols in his first book [205].

2.6 Exercises

Exercise 2.1. Consider the following two sequential hardware circuits:



Questions:

(a) Give the transition systems of both hardware circuits.

(b) Determine the reachable part of the transition system of the synchronous product of these transition systems. Assume that the initial values of the registers are r_1=0 and r_2=1.

Exercise 2.2. We are given three (primitive) processes P_1, P_2, and P_3 with shared integer variable x. The program of process $P_i$is as follows:

Algorithm 1 Process P_i

for $k_i = 1, \ldots, 10$ **do**
  LOAD(x);
  INC(x);
  STORE(x);
od

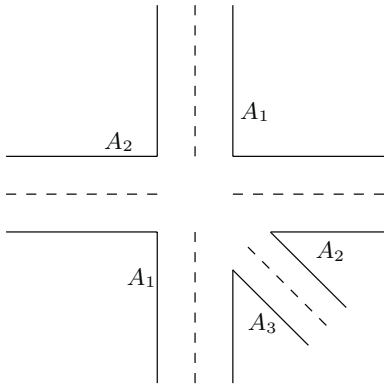
That is, P_i executes ten times the assignment $x := x{+}1$. The assignment $x := x{+}1$ is realized using the three actions LOAD(x), INC(x) and STORE(x). Consider now the parallel program:

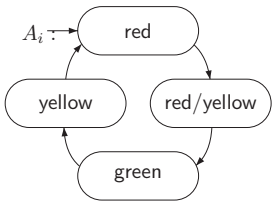
Algorithm 2 Parallel program P

$x := 0$;
$P_1 \;||\; P_2 \;||\; P_3$

Question: Does P have an execution that halts with the terminal value $x = 2$?

Exercise 2.3. Consider the following street junction with the specification of a traffic light as outlined on the right.





(a) Choose appropriate actions and label the transitions of the traffic light transition system accordingly.

(b) Give the transition system representation of a (reasonable) controller C that switches the green signal lamps in the following order: $A_1, A_2, A_3, A_1, A_2, A_3, \ldots$.
(*Hint: Choose an appropriate communication mechanism.*)

(c) Outline the transition system $A_1 \| A_2 \| A_3 \| C$.

Exercise 2.4. Show that the handshaking operator $\|$ that forces two transition systems to synchronize over their common actions (see Definition 2.26 on page 48) is associative. That is, show that

$$(TS_1 \| TS_2) \| TS_3 \quad = \quad TS_1 \| (TS_2 \| TS_3)$$

where TS_1, TS_2, TS_3 are arbitrary transition systems.

Exercise 2.5. The following program is a mutual exclusion protocol for two processes due to Pnueli [118]. There is a single shared variable s which is either 0 or 1, and initially 1. Besides, each process has a local Boolean variable y that initially equals 0. The program text for process P_i $(i = 0, 1)$ is as follows:

l0: **loop forever do**
  begin
  l1: Noncritical section
  l2: $(y_i, s) := (1, i)$;
  l3: **wait until** $((y_{1-i} = 0) \ \vee \ (s \neq i))$;
  l4: Critical section
  l5: $y_i := 0$
end.

Here, the statement $(y_i, s) := (1, i)$; is a *multiple assignment* in which variable $y_i := 1$ and $s := i$ is a single, atomic step.

Questions:

(a) Define the program graph of a process in Pnueli's algorithm.

(b) Determine the transition system for each process.

(c) Construct their parallel composition.

(d) Check whether the algorithm ensures mutual exclusion.

(e) Check whether the algorithm ensures starvation freedom.

The last two questions may be answered by inspecting the transition system.

EXERCISE 2.6. Consider a stack of nonnegative integers with capacity n (for some fixed n).

(a) Give a transition system representation of this stack. You may abstract from the values on the stack and use the operations *top*, *pop*, and *push* with their usual meaning.

(b) Sketch a transition system representation of the stack in which the concrete stack content is explicitly represented.

EXERCISE 2.7. Consider the following generalization of Peterson's mutual exclusion algorithm that is aimed at an arbitrary number n ($n \geqslant 2$) processes. The basic concept of the algorithm is that each process passes through n "levels" before acquiring access to the critical section. The concurrent processes share the bounded integer arrays $y[0..n{-}1]$ and $p[1..n]$ with $y[i] \in \{1, \ldots, n\}$ and $p[i] \in \{0, \ldots, n{-}1\}$. $y[j] = i$ means that process i has the lowest priority at level j, and $p[i] = j$ expresses that process i is currently at level j. Process i starts at level 0. On requesting access to the critical section, the process passes through levels 1 through $n{-}1$. Process i waits at level j until either all other processes are at a lower level (i.e., $p[k] < j$ for all $k \neq i$) or another process grants process i access to its critical section (i.e., $y[j] \neq i$). The behavior of process i is in pseudocode:

while true **do**
  ... noncritical section ...
  forall $j = 1, \ldots, n{-}1$ **do**
    $p[i] := j$;
    $y[j] := i$;
    wait until $(y[j] \neq i) \vee \left(\bigwedge_{0<k\leqslant n, k\neq i} p[k] < j\right)$
  od
  ... critical section ...
  $p[i] := 0$;
od

Questions:

(a) Give the program graph for process i.

(b) Determine the number of states (including the unreachable states) in the parallel composition of n processes.

(c) Prove that this algorithm ensures mutual exclusion for n processes.

(d) Prove that it is impossible that all processes are waiting in the for-iteration.

(e) Establish whether it is possible that a process that wants to enter the critical section waits ad infinitum.

EXERCISE 2.8. In channel systems, values can be transferred from one process to another process. As this is somewhat limited, we consider in this exercise an extension that allows for the transfer of *expressions*. That is to say, the send and receive statements $c!v$ and $c?x$ (where x and v are of the same type) are generalized into $c!expr$ and $c?x$, where for simplicity it is assumed that $expr$ is a correctly typed expression (of the same type as x). Legal expressions are, e.g., $x \wedge (\neg y \vee z)$ for Boolean variables x, y, and z, and channel c with $dom(c) = \{0, 1\}$. For integers x, y, and an integer-channel c, $|2x + (x - y)\text{div}17|$ is a legal expression.

Question: Extend the transition system semantics of channel systems such that expressions are allowed in send statements.
(Hint: Use the function η such that for expression expr, $\eta(expr)$ is the evaluation of expr under the variable valuation η.)

EXERCISE 2.9. Consider the following mutual exclusion algorithm that uses the shared variables y_1 and y_2 (initially both 0).

Process P_1:	Process P_2:
while true **do**	**while** true **do**
... noncritical section ...	... noncritical section ...
$y_1 := y_2 + 1$;	$y_2 := y_1 + 1$;
wait until $(y_2 = 0) \vee (y_1 < y_2)$	**wait until** $(y_1 = 0) \vee (y_2 < y_1)$
... critical section ...	... critical section ...
$y_1 := 0$;	$y_2 := 0$;
od	**od**

Questions:

(a) Give the program graph representations of both processes. (A pictorial representation suffices.)

(b) Give the reachable part of the transition system of $P_1 \parallel P_2$ where $y_1 \leqslant 2$ and $y_2 \leqslant 2$.

(c) Describe an execution that shows that the entire transition system is infinite.

(d) Check whether the algorithm indeed ensures mutual exclusion.

(e) Check whether the algorithm never reaches a state in which both processes are mutually waiting for each other.

(f) Is it possible that a process that wants to enter the critical section has to wait ad infinitum?

Exercise 2.10. Consider the following mutual exclusion algorithm that was proposed 1966 [221] as a simplification of Dijkstra's mutual exclusion algorithm in case there are just two processes:

```
 1 Boolean array b(0;1) integer k, i, j,
 2 comment  This is the program for computer i, which may be
      either 0 or 1, computer j =/= i is the other one, 1 or 0;
 3 C0: b(i) := false;
 4 C1: if k != i then begin
 5 C2: if not b(j) then go to C2;
 6     else k := i; go to C1 end;
 7     else critical section;
 8     b(i) := true;
 9     remainder of program;
10     go to C0;
11     end
```

Here `C0`, `C1`, and `C2` are program labels, and the word "computer" should be interpreted as process.

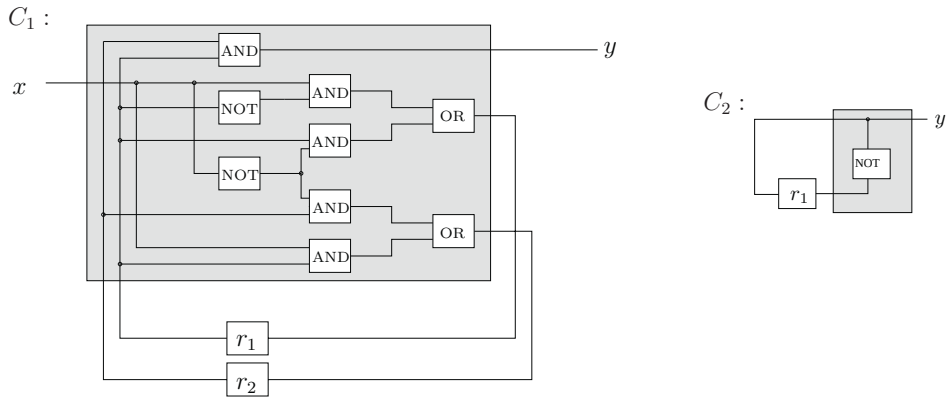
Questions:

(a) Give the program graph representations for a single process. (A pictorial representation suffices.)

(b) Give the reachable part of the transition system of $P_1 \parallel P_2$.

(c) Check whether the algorithm indeed ensures mutual exclusion.

EXERCISE 2.11. Consider the following two sequential hardware circuits C_1 and C_2:



(a) Give the transition system representation $TS(C_1)$ of the circuit C_1.

(b) Let $TS(C_2)$ be the transition system of the circuit C_2. Outline the transition system $TS(C_1) \otimes TS(C_2)$.

EXERCISE 2.12. Consider the following leader election algorithm: For $n \in \mathbb{N}$, n processes $P_1, \ldots, P_n$ are located in a ring topology where each process is connected by an unidirectional channel to its neighbor in a clockwise manner.

To distinguish the processes, each process is assigned a unique identifier $id \in \{1, \ldots, n\}$. The aim is to elect the process with the highest identifier as the leader within the ring. Therefore each process executes the following algorithm:

send (id); *initially set to process' id*
while (true) **do**
  receive (m);
  if ($m = id$) **then stop**; *process is the leader*
  if ($m > id$) **then send** (m); *forward identifier*
od

(a) Model the leader election protocol for n processes as a channel system.

(b) Give an initial execution fragment of $TS([P_1|P_2|P_3])$ such that at least one process has executed the send statement within the body of the whileloop. Assume for $0 < i \leqslant 3$, that process P_i has identifier $id_i = i$.

Chapter 3

Linear-Time Properties

For verification purposes, the transition system model of the system under consideration needs to be accompanied with a specification of the property of interest that is to be verified. This chapter introduces some important, though relatively simple, classes of properties. These properties are formally defined and basic model-checking algorithms are presented to check such properties in an automated manner. This chapter focuses on linear-time behavior and establishes relations between the different classes of properties and trace behavior. Elementary forms of fairness are introduced and compared.

3.1 Deadlock

Sequential programs that are not subject to divergence (i.e., endless loops) have a terminal state, a state without any outgoing transitions. For parallel systems, however, computations typically do not terminate—consider, for instance, the mutual exclusion programs treated so far. In such systems, terminal states are undesirable and mostly represent a design error. Apart from "trivial" design errors where it has been forgotten to indicate certain activities, in most cases such terminal states indicate a *deadlock*. A deadlock occurs if the complete system is in a terminal state, although at least one component is in a (local) nonterminal state. The entire system has thus come to a halt, whereas at least one component has the possibility to continue to operate. A typical deadlock scenario occurs when components mutually wait for each other to progress.

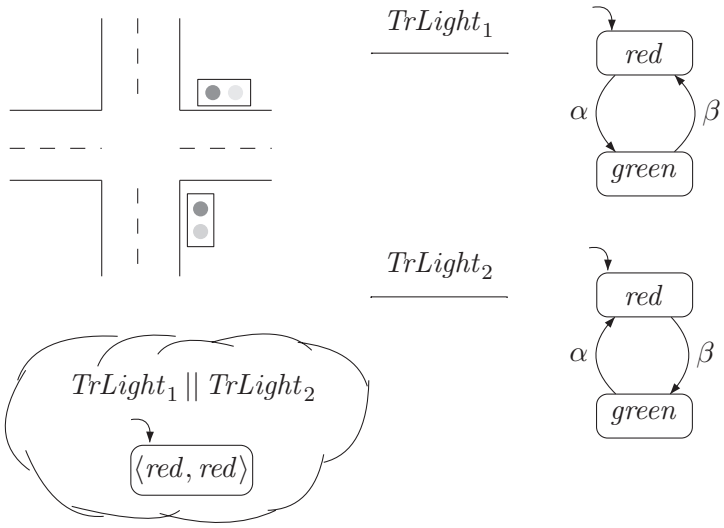


Figure 3.1: An example of a deadlock situation.

Example 3.1. Deadlock for Fault Designed Traffic Lights

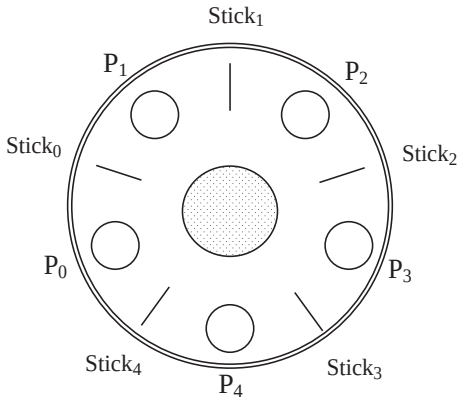
Consider the parallel composition of two transition systems

$$TrLight_1 \parallel TrLight_2$$

modeling the traffic lights of two intersecting roads. Both traffic lights synchronize by means of the actions α and β that indicate the change of light (see Figure 3.1). The apparently trivial error to let both traffic lights start with a red light results in a deadlock. While the first traffic light is waiting to be synchronized on action α, the second traffic light is blocked, since it is waiting to be synchronized with action β. ■

Example 3.2. Dining Philosophers

This example, originated by Dijkstra, is one of the most prominent examples in the field of concurrent systems.



Five philosophers are sitting at a round table with a bowl of rice in the middle. For the philosophers (being a little unworldly) life consists of thinking and eating (and waiting, as we will see). To take some rice out of the bowl, a philosopher needs two chopsticks. In between two neighboring philosophers, however, there is only a single chopstick. Thus, at any time only one of two neighboring philosophers can eat. Of course, the use of the chopsticks is exclusive and eating with hands is forbidden.

Note that a deadlock scenario occurs when all philosophers possess a single chopstick. The problem is to design a protocol for the philosophers, such that the complete system is deadlock-free, i.e., at least one philosopher can eat and think infinitely often. Additionally, a fair solution may be required with each philosopher being able to think and eat infinitely often. The latter characteristic is called freedom of *individual starvation*.

The following obvious design cannot ensure deadlock freedom. Assume the philosophers and the chopsticks are numbered from 0 to 4. Furthermore, assume all following calculations be "modulo 5", e.g., chopstick $i-1$ for $i=0$ denotes chopstick 4, and so on.

Philosopher i has stick i on his left and stick $i-1$ on his right side. The action $request_{i,i}$ express that stick i is picked up by philosopher i. Accordingly, $request_{i-1,i}$ denotes the action by means of which philosopher i picks up the $(i-1)$th stick. The actions $release_{i,i}$ and $release_{i-1,i}$ have a corresponding meaning.

The behavior of philosopher i (called process $Phil_i$) is specified by the transition system depicted in the left part of Figure 3.2. Solid arrows depict the synchronizations with the i-th stick, dashed arrows refer to communications with the $i-1$th stick. The sticks are modeled as independent processes (called $Stick_i$) with which the philosophers synchronize via actions *request* and *release*; see the right part of Figure 3.2 that represents the process of stick i. A stick process prevents philosopher i from picking up the ith stick when philosopher $i+1$ is using it.

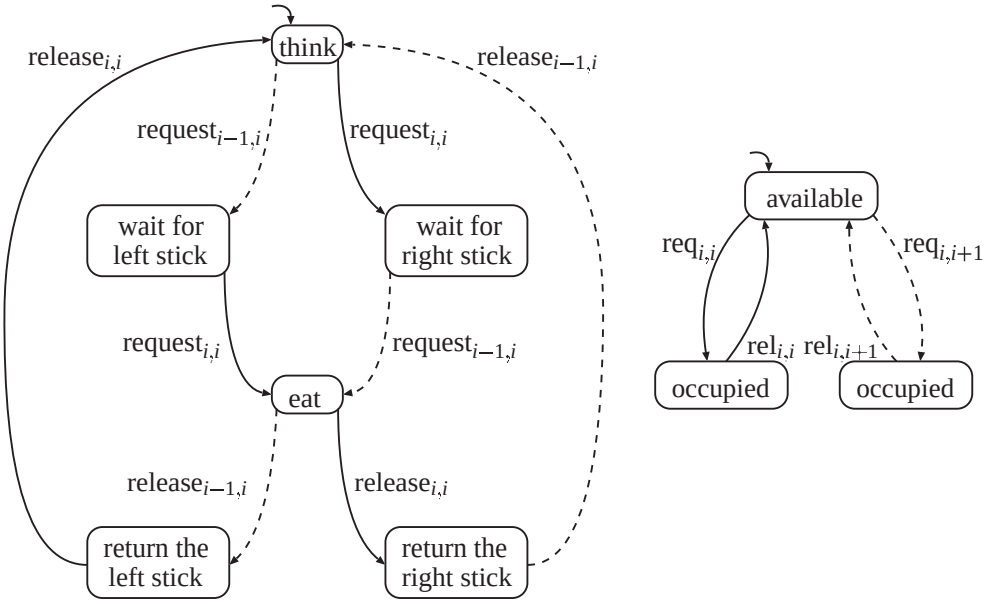


Figure 3.2: Transition systems for the ith philosopher and the ith stick.

The complete system is of the form:

$$Phil_4 \parallel Stick_3 \parallel Phil_3 \parallel Stick_2 \parallel Phil_2 \parallel Stick_1 \parallel Phil_1 \parallel Stick_0 \parallel Phil_0 \parallel Stick_4$$

This (initially obvious) design leads to a deadlock situation, e.g., if all philosophers pick up their left stick at the same time. A corresponding execution leads from the initial state

$$\langle think_4, avail_3, think_3, avail_2, think_2, avail_1, think_1, avail_0, think_0, avail_4 \rangle$$

by means of the action sequence $request_4$, $request_3$, $request_2$, $request_1$, $request_0$ (or any other permutation of these 5 request actions) to the terminal state

$$\langle wait_{4,0}, occ_{4,4}, wait_{3,4}, occ_{3,3}, wait_{2,3}, occ_{2,2}, wait_{1,2}, occ_{1,1}, wait_{0,1}, occ_{0,0} \rangle.$$

This terminal state represents a deadlock with each philosopher waiting for the needed stick to be released.

A possible solution to this problem is to make the sticks available for only one philosopher at a time. The corresponding chopstick process is depicted in the right part of Figure 3.3. In state $available_{i,j}$ only philosopher j is allowed to pick up the ith stick. The above-mentioned deadlock situation can be avoided by the fact that some sticks (e.g., the first, the third, and the fifth stick) start in state $available_{i,i}$, while the remaining sticks start in state $available_{i,i+1}$. It can be verified that this solution is deadlock- and starvation-free.

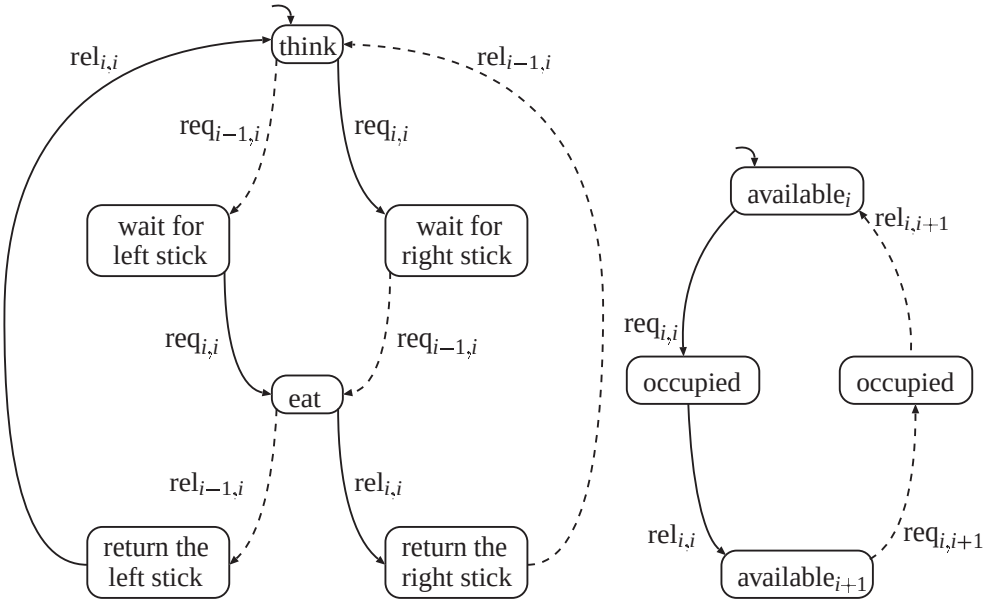


Figure 3.3: Improved variant of the ith philosopher and the ith stick.

A further characteristic often required for concurrent systems is robustness against failure of their components. In the case of the dining philosophers, robustness can be formulated in a way that ensures deadlock and starvation freedom even if one of the philosophers is "defective" (i.e., does not leave the think phase anymore).[1] The above-sketched deadlock- and starvation-free solution can be modified to a fault-tolerant solution by changing the transition systems of philosophers and sticks such that philosopher $i{+}1$ can pick up the ith stick even if philosopher i is thinking (i.e., does not need stick i) independent of whether stick i is in state $\mathit{available}_{i,i}$ or $\mathit{available}_{i,i+1}$. The corresponding is also true when the roles of philosopher i and $i{+}1$ are reversed. This can be established by adding a single Boolean variable x_i to philosopher i (see Figure 3.4). The variable x_i informs the neighboring philosophers about the current location of philosopher i. In the indicated sketch, x_i is a Boolean variable which is true if and only if the ith philosopher is thinking. Stick i is made available to philosopher i if stick i is in location $\mathit{available}_i$ (as before), or if stick i is in location $\mathit{available}_{i+1}$ while philosopher $i{+}1$ is thinking.

Note that the above description is at the level of program graphs. The complete system is a channel system with *request* and *release* actions standing for handshaking over a channel of capacity 0. ■

[1] Formally, we add a loop to the transition system of a defective philosopher at state think_i.

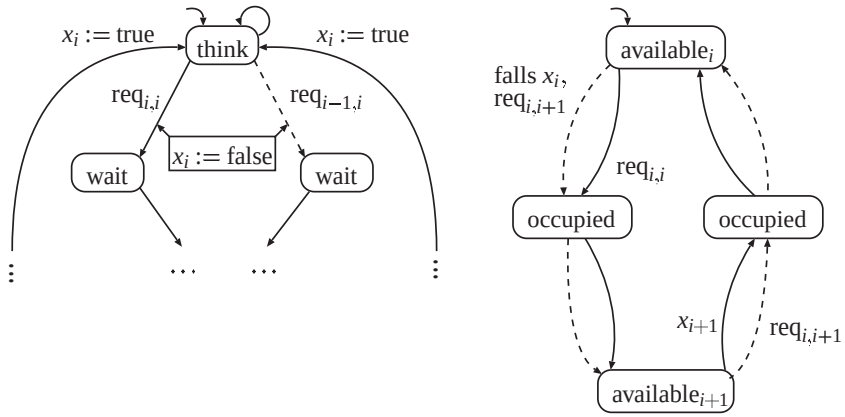


Figure 3.4: Fault-tolerant variant of the dining philosophers.

3.2 Linear-Time Behavior

To analyze a computer system represented by a transition system, either an action-based or a state-based approach can be followed. The state-based approach abstracts from actions; instead, only labels in the state sequences are taken into consideration. In contrast, the action-based view abstracts from states and refers only to the action labels of the transitions. (A combined action- and state-based view is possible, but leads to more involved definitions and concepts. For this reason it is common practice to abstract from either action or state labels.) Most of the existing specification formalisms and associated verification methods can be formulated in a corresponding way for both perspectives.

In this chapter, we mainly focus on the state-based approach. Action labels of transitions are only necessary for modeling communication; thus, they are of no relevance in the following chapters. Instead, we use the atomic propositions of the states to formulate system properties. Therefore, the verification algorithms operate on the *state graph* of a transition system, the digraph originating from a transition system by abstracting from action labels.

3.2.1 Paths and State Graph

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system.

Definition 3.3. State Graph

The *state graph* of TS, notation $G(TS)$, is the digraph (V, E) with vertices $V = S$ and edges $E = \{(s, s') \in S \times S \mid s' \in Post(s)\}$. ■

The state graph of transition system TS has a vertex for each state in TS and an edge between vertices s and s' whenever s' is a direct successor of s in TS for some action α. It is thus simply obtained from TS by omitting all state labels (i.e., the atomic propositions), all transition labels (i.e., the actions), and by ignoring the fact whether a state is initial or not. Moreover, multiple transitions (that have different action labels) between states are represented by a single edge. This seems to suggest that the state labels are no longer of any use; later on, we will see how these state labels will be used to check the validity of properties.

Let $Post^*(s)$ denote the states that are reachable in state graph $G(TS)$ from s. This notion is generalized toward sets of states in the usual way (i.e., pointwise extension): for $C \subseteq S$ let

$$Post^*(C) \;=\; \bigcup_{s \in C} Post^*(s).$$

The notations $Pre^*(s)$ and $Pre^*(C)$ have analogous meaning. The set of states that are reachable from some initial state, notation $Reach(TS)$, equals $Post^*(I)$.

As explained in Chapter 2, the possible behavior of a transition system is defined by an execution fragment. Recall that an execution fragment is an alternating sequence of states and actions. As we consider a state-based approach, the actions are not of importance and are omitted. The resulting "runs" of a transition system are called *paths*. The following definitions define path fragments, initial and maximal path fragments, and so on. These notions are easily obtained from the same notions for executions by omitting the actions.

Definition 3.4. Path Fragment

A *finite* path fragment $\widehat{\pi}$ of TS is a finite state sequence $s_0\, s_1 \ldots s_n$ such that $s_i \in Post(s_{i-1})$ for all $0 < i \leqslant n$, where $n \geqslant 0$. An *infinite* path fragment π is an infinite state sequence $s_0\, s_1\, s_2 \ldots$ such that $s_i \in Post(s_{i-1})$ for all $i > 0$. ■

We adopt the following notational conventions for infinite path fragment $\pi = s_0\, s_1 \ldots$. The initial state of π is denoted by $first(\pi) = s_0$. For $j \geqslant 0$, let $\pi[j] = s_j$ denote the jth state of

π and $\pi[..j]$ denote the jth prefix of π, i.e., $\pi[..j] = s_0\, s_1 \ldots s_j$. Similarly, the jth suffix of π, notation $\pi[j..]$, is defined as $\pi[j..] = s_j\, s_{j+1} \ldots$. These notions are defined analogously for finite paths. Besides, for finite path $\widehat{\pi} = s_0\, s_1 \ldots s_n$, let $\mathit{last}(\widehat{\pi}) = s_n$ denote the last state of $\widehat{\pi}$, and $\mathit{len}(\widehat{\pi}) = n$ denote the length of $\widehat{\pi}$. For infinite path π these notions are defined by $\mathit{len}(\pi) = \infty$ and $\mathit{last}(\pi) = \bot$, where $\bot$ denotes "undefined".

Definition 3.5. Maximal and Initial Path Fragment

A *maximal* path fragment is either a finite path fragment that ends in a terminal state, or an infinite path fragment. A path fragment is called *initial* if it starts in an initial state, i.e., if $s_0 \in I$. ■

A maximal path fragment is a path fragment that cannot be prolonged: either it is infinite or it is finite but ends in a state from which no transition can be taken. Let $\mathit{Paths}(s)$ denote the set of maximal path fragments π with $\mathit{first}(\pi) = s$, and $\mathit{Paths}_{\mathit{fin}}(s)$ denote the set of all finite path fragments $\widehat{\pi}$ with $\mathit{first}(\widehat{\pi}) = s$.

Definition 3.6. Path

A *path* of transition system TS is an initial, maximal path fragment.[2] ■

Let $\mathit{Paths}(TS)$ denote the set of all paths in TS, and $\mathit{Paths}_{\mathit{fin}}(TS)$ the set of all initial, finite path fragments of TS.

Example 3.7. Beverage Vending Machine

Consider the beverage vending machine of Example 2.2 on page 21. For convenience, its transition system is repeated in Figure 3.5. As the state labeling is simply $L(s) = \{ s \}$ for each state s, the names of states may be used in paths (as in this example), as well as atomic propositions (as used later on). Example path fragments of this transition system are

$$\begin{aligned} \pi_1 &= \mathit{pay}\ \mathit{select}\ \mathit{soda}\ \mathit{pay}\ \mathit{select}\ \mathit{soda} \ldots \\ \pi_2 &= \mathit{select}\ \mathit{soda}\ \mathit{pay}\ \mathit{select}\ \mathit{beer} \ldots \\ \widehat{\pi} &= \mathit{pay}\ \mathit{select}\ \mathit{soda}\ \mathit{pay}\ \mathit{select}\ \mathit{soda}\,. \end{aligned}$$

These path fragments result from the execution fragments indicated in Example 2.8 on page 25. Only π_1 is a path. The infinite path fragment π_2 is maximal but not initial. $\widehat{\pi}$ is initial but not maximal since it is finite while ending in a state that has outgoing

[2] It is important to realize the difference between the notion of a path in a transition system and the notion of a path in a digraph. A path in a transition system is maximal, whereas a path in a digraph in the graph-theoretical sense is not always maximal. Besides, paths in a digraph are usually required to be finite whereas paths in transition systems may be infinite.

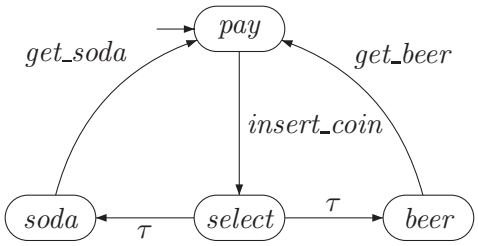


Figure 3.5: A transition system of a simple beverage vending machine.

transitions. We have that $last(\widehat{\pi}) = soda$, $first(\pi_2) = select$, $\pi_1[0] = pay$, $\pi_1[3] = pay$, $\pi_1[..5] = \widehat{\pi}$, $\widehat{\pi}[..2] = \widehat{\pi}[3..]$, $len(\widehat{\pi}) = 5$, and $len(\pi_1) = \infty$. ■

3.2.2 Traces

Executions (as introduced in Chapter 2) are alternating sequences consisting of states and actions. Actions are mainly used to model the (possibility of) interaction, be it synchronous or asynchronous communication. In the sequel, interaction is not our prime interest, but instead we focus on the states that are visited during executions. In fact, the states themselves are not "observable", but just their atomic propositions. Thus, rather than having an execution of the form $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \ldots$ we consider sequences of the form $L(s_0)\,L(s_1)\,L(s_2)\ldots$ that register the (set of) atomic propositions that are valid along the execution. Such sequences are called *traces*.

The traces of a transition system are thus words over the alphabet 2^{AP}. In the following it is assumed that a transition system has no terminal states. In this case, all traces are infinite words. (Recall that the traces of a transition system have been defined as traces induced by its initial maximal path fragments. See also Appendix A.2, page 912). This assumption is made for simplicity and does not impose any serious restriction. First of all, prior to checking any (linear-time) property, a reachability analysis could be carried out to determine the set of terminal states. If indeed some terminal state is encountered, the system contains a deadlock and has to be repaired before any further analysis. Alternatively, each transition system TS (that probably has a terminal state) can be extended such that for each terminal state s in TS there is a new state s_{stop}, transition $s \longrightarrow s_{stop}$, and s_{stop} is equipped with a self-loop, i.e., $s_{stop} \longrightarrow s_{stop}$. The resulting "equivalent" transition system obviously has no terminal states.[3]

[3] A further alternative is to adapt the linear-time framework for transition systems with terminal states. The main concepts of this chapter are still applicable, but require some adaptions to distinguish nonmax-

Definition 3.8. Trace and Trace Fragment

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system without terminal states. The *trace* of the infinite path fragment $\pi = s_0\, s_1 \ldots$ is defined as $trace(\pi) = L(s_0)\, L(s_1) \ldots$. The trace of the finite path fragment $\widehat{\pi} = s_0\, s_1 \ldots s_n$ is defined as $trace(\widehat{\pi}) = L(s_0)\, L(s_1) \ldots L(s_n)$. ■

The trace of a path fragment is thus the induced finite or infinite word over the alphabet 2^{AP}, i.e., the sequence of sets of atomic propositions that are valid in the states of the path. The set of traces of a set Π of paths is defined in the usual way:

$$trace(\Pi) \;=\; \{\, trace(\pi) \mid \pi \in \Pi \,\}.$$

A trace of state s is the trace of an infinite path fragment π with $first(\pi) = s$. Accordingly, a finite trace of s is the trace of a finite path fragment that starts in s. Let $Traces(s)$ denote the set of traces of s, and $Traces(TS)$ the set of traces of the initial states of transition system TS:

$$Traces(s) = trace(Paths(s)) \quad \text{and} \quad Traces(TS) = \bigcup_{s \in I} Traces(s).$$

In a similar way, the finite traces of a state and of a transition system are defined:

$$Traces_{fin}(s) = trace(Paths_{fin}(s)) \quad \text{and} \quad Traces_{fin}(TS) = \bigcup_{s \in I} Traces_{fin}(s).$$

Example 3.9. Semaphore-Based Mutual Exclusion

Consider the transition system TS_{Sem} as depicted in Figure 3.6. This two-process mutual exclusion example has been described before in Example 2.24 (page 43).

Assume the available atomic propositions are $crit_1$ and $crit_2$, i.e.,

$$AP = \{\, crit_1, crit_2 \,\}.$$

The proposition $crit_1$ holds in any state of the transition system TS_{Sem} where the first process (called P_1) is in its critical section. Proposition $crit_2$ has the same meaning for the second process (i.e., P_2).

Consider the execution in which the processes P_1 and P_2 enter their critical sections in an alternating fashion. Besides, they only request to enter the critical section when the

imal and maximal finite paths.

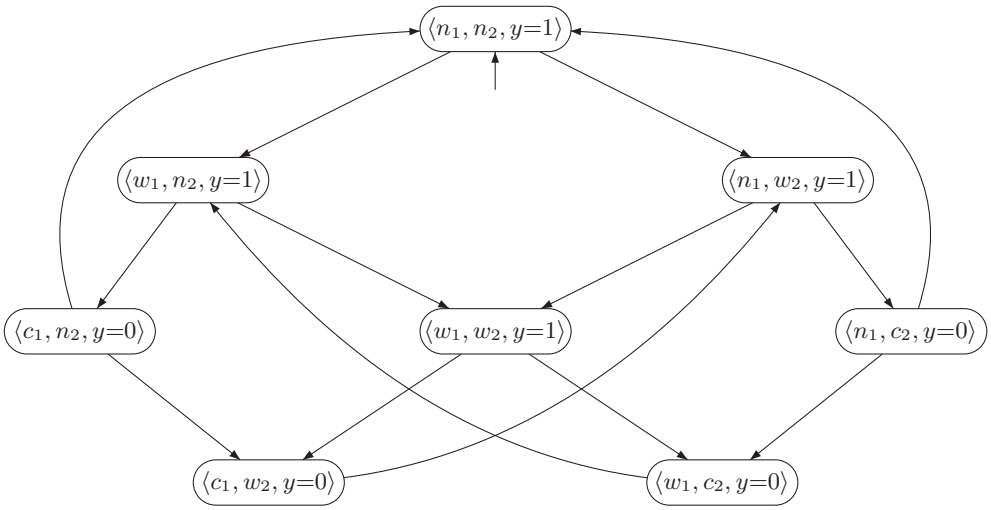


Figure 3.6: Transition system of semaphore-based mutual exclusion algorithm.

other process is no longer in its critical section. Situations in which one process is in its critical section whereas the other is moving from the noncritical state to the waiting state are impossible.

The path π in the state graph of TS_{Sem} where process P_1 is the first to enter its critical section is of the form

$$\begin{aligned}\pi \quad = \quad & \langle n_1, n_2, y=1\rangle \to \langle w_1, n_2, y=1\rangle \to \langle c_1, n_2, y=0\rangle \to \\ & \langle n_1, n_2, y=1\rangle \to \langle n_1, w_2, y=1\rangle \to \langle n_1, c_2, y=0\rangle \to \ldots\end{aligned}$$

The trace of this path is the infinite word:

$$\mathit{trace}(\pi) \;=\; \varnothing\,\varnothing\,\{\,\mathit{crit}_1\,\}\,\varnothing\,\varnothing\,\{\,\mathit{crit}_2\,\}\,\varnothing\,\varnothing\,\{\,\mathit{crit}_1\,\}\,\varnothing\,\varnothing\,\{\,\mathit{crit}_2\,\}\ldots.$$

The trace of the finite path fragment

$$\begin{aligned}\widehat{\pi} \quad = \quad & \langle n_1, n_2, y=1\rangle \to \langle w_1, n_2, y=1\rangle \to \langle w_1, w_2, y=1\rangle \to \\ & \langle w_1, c_2, y=0\rangle \to \langle w_1, n_2, y=1\rangle \to \langle c_1, n_2, y=0\rangle\end{aligned}$$

is $\mathit{trace}(\widehat{\pi}) \;=\; \varnothing\,\varnothing\,\varnothing\,\{\,\mathit{crit}_2\,\}\,\varnothing\,\{\,\mathit{crit}_1\,\}$. ■

3.2.3 Linear-Time Properties

Linear-time properties specify the traces that a transition system should exhibit. Informally speaking, one could say that a linear-time property specifies the admissible (or desired) behavior of the system under consideration. In the following we provide a formal definition of such properties. This definition is rather elementary, and gives a good basic understanding of what a linear-time property is. In Chapter 5, a logical formalism will be introduced that allows for the specification of linear-time properties.

In the following, we assume a fixed set of propositions AP. A linear-time (LT) property is a requirement on the traces of a transition system. Such property can be understood as a requirement over all words over AP, and is defined as the set of words (over AP) that are admissible:

Definition 3.10. LT Property

A *linear-time property* (LT property) over the set of atomic propositions AP is a subset of $(2^{AP})^{\omega}$. ■

Here, $(2^{AP})^{\omega}$ denotes the set of words that arise from the infinite concatenation of words in 2^{AP}. An LT property is thus a language (set) of infinite words over the alphabet 2^{AP}. Note that it suffices to consider infinite words only (and not finite words), as transition systems without terminal states are considered. The fulfillment of an LT property by a transition system is defined as follows.

Definition 3.11. Satisfaction Relation for LT Properties

Let P be an LT property over AP and $TS = (S, Act, \rightarrow, I, AP, L)$ a transition system without terminal states. Then, $TS = (S, Act, \rightarrow, I, AP, L)$ *satisfies* P, denoted $TS \models P$, iff $\mathit{Traces}(TS) \subseteq P$. State $s \in S$ satisfies P, notation $s \models P$, whenever $\mathit{Traces}(s) \subseteq P$. ■

Thus, a transition system satisfies the LT property P if all its traces respect P, i.e., if all its behaviors are admissible. A state satisfies P whenever all traces starting in this state fulfill P.

Example 3.12. Traffic Lights

Consider two simplified traffic lights that only have two possible settings: red and green. Let the propositions of interest be

$$AP = \{ red_1, green_1, red_2, green_2 \}$$

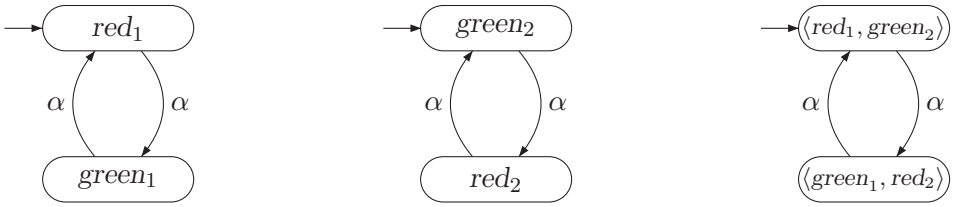


Figure 3.7: Two fully synchronized traffic lights (left and middle) and their parallel composition (right).

We consider two LT properties of these traffic lights and give some example words that are contained by such properties. First, consider the property P that states:

"The first traffic light is infinitely often green".

This LT property corresponds to the set of infinite words of the form $A_0\,A_1\,A_2\,\ldots$ over 2^{AP}, such that $green_1 \in A_i$ holds for infinitely many i. For example, P contains the infinite words

$$\{\, red_1, green_2 \,\}\,\{\, green_1, red_2 \,\}\,\{\, red_1, green_2 \,\}\,\{\, green_1, red_2 \,\}\ldots,$$
$$\varnothing\,\{\, green_1 \,\}\,\varnothing\,\{\, green_1 \,\}\,\varnothing\,\{\, green_1 \,\}\,\varnothing\,\{\, green_1 \,\}\,\varnothing\ldots$$
$$\{\, red_1, green_1 \,\}\,\{\, red_1, green_1 \,\}\,\{\, red_1, green_1 \,\}\,\{\, red_1, green_1 \,\}\ldots \quad \text{and}$$
$$\{\, green_1, green_2 \,\}\,\{\, green_1, green_2 \,\}\,\{\, green_1, green_2 \,\}\,\{\, green_1, green_2 \,\}\ldots$$

The infinite word $\{\, red_1, green_1 \,\}\,\{\, red_1, green_1 \,\}\,\varnothing\,\varnothing\,\varnothing\,\varnothing\ldots$ is not in P as it contains only finitely many occurrences of $green_1$.

As a second LT property, consider P':

"The traffic lights are never both green simultaneously".

This property is formalized by the set of infinite words of the form $A_0\,A_1\,A_2\ldots$ such that either $green_1 \notin A_i$ or $green_2 \notin A_i$, for all $i \geqslant 0$. For example, the following infinite words are in P':

$$\{\, red_1, green_2 \,\}\,\{\, green_1, red_2 \,\}\,\{\, red_1, green_2 \,\}\,\{\, green_1, red_2 \,\}\ldots,$$
$$\varnothing\,\{\, green_1 \,\}\,\varnothing\,\{\, green_1 \,\}\,\varnothing\,\{\, green_1 \,\}\,\varnothing\,\{\, green_1 \,\}\,\varnothing\ldots \quad \text{and}$$
$$\{\, red_1, green_1 \,\}\,\{\, red_1, green_1 \,\}\,\{\, red_1, green_1 \,\}\,\{\, red_1, green_1 \,\}\ldots,$$

whereas the infinite word $\{\, red_1\, green_2 \,\}\,\{\, green_1, green_2 \,\},\ldots$ is not in P'.

The traffic lights depicted in Figure 3.7 are at intersecting roads and their switching is synchronized, i.e., if one light switches from red to green, the other switches from green to

red. In this way, the lights always have complementary colors. Clearly, these traffic lights satisfy both P and P'. Traffic lights that switch completely autonomously will neither satisfy P—there is no guarantee that the first traffic light is green infinitely often—nor P'. ■

Often, an LT property does not refer to all atomic propositions occurring in a transition system, but just to a relatively small subset thereof. For a property P over a set of propositions $AP' \subseteq AP$, only the labels in AP' are relevant. Let $\widehat{\pi}$ be a finite path fragment of TS. We write $trace_{AP'}(\widehat{\pi})$ to denote the finite trace of $\widehat{\pi}$ where only the atomic propositions in AP' are considered. Accordingly, $trace_{AP'}(\pi)$ denotes the trace of an infinite path fragment π by focusing on propositions in AP' Thus, for $\pi = s_0\, s_1\, s_2 \ldots$, we have

$$trace_{AP'}(\pi) \;=\; L'(s_0)\, L'(s_1) \ldots \;=\; (L(s_0) \cap AP')\, (L(s_1) \cap AP') \ldots$$

Let $Traces_{AP'}(TS)$ denote the set of traces $trace_{AP'}(Paths(TS))$. Whenever the set AP' of atomic propositions is clear from the context, the subscript AP' is omitted. In the rest of this chapter, the restriction to a relevant subset of atomic propositions is often implicitly made.

Example 3.13. The Mutual Exclusion Property

In Chapter 2, several mutual exclusion algorithms have been considered. For specifying the mutual exclusion property—always at most one process is in its critical section—it suffices to only consider the atomic propositions $crit_1$ and $crit_2$. Other atomic propositions are not of any relevance for this property. The formalization of the mutual exclusion property is given by the LT property

$$P_{mutex} \;=\; \text{set of infinite words } A_0\, A_1\, A_2 \ldots \text{ with } \{\, crit_1, crit_2 \,\} \not\subseteq A_i \text{ for all } 0 \leqslant i.$$

For example, the infinite words

$$\{\, crit_1 \,\}\, \{\, crit_2 \,\}\, \{\, crit_1 \,\}\, \{\, crit_2 \,\}\, \{\, crit_1 \,\}\, \{\, crit_2 \,\} \ldots, \quad \text{and}$$
$$\{\, crit_1 \,\}\, \{\, crit_1 \,\}\, \{\, crit_1 \,\}\, \{\, crit_1 \,\}\, \{\, crit_1 \,\}\, \{\, crit_1 \,\} \ldots, \quad \text{and}$$
$$\varnothing\, \varnothing\, \varnothing\, \varnothing\, \varnothing\, \varnothing\, \varnothing \ldots$$

are all contained in P_{mutex}. However, this does not apply to words of the form

$$\{\, crit_1 \,\}\, \varnothing\, \{\, crit_1, crit_2 \,\} \ldots$$

The transition system $TS_{Arb} = (TS_1 \,|||\, TS_2) \parallel Arbiter$ described in Example 2.28 (page 50) fulfills the mutex property, i.e.,

$$TS_{Arb} \;\models\; P_{mutex}.$$

It is left to the reader to check that the mutex property is also fulfilled by the semaphore-based mutual exclusion algorithm (see Figure 3.6 on page 99) and Peterson's algorithm (see Example 2.25 on page 45). ■

Example 3.14. Starvation Freedom

Guaranteeing mutual exclusion is a significant property of mutual exclusion algorithms, but is not the only relevant property. An algorithm that never allows a process to enter its critical section will do, but is certainly not intended. Besides, a property is imposed that requires a process that wants to enter the critical section to be able to eventually do so. This property prevents a process from waiting ad infinitum and is formally specified as the LT property $P_{finwait}$ = set of infinite words $A_0\,A_1\,A_2\ldots$ such that

$$\forall j.\, wait_i \in A_j \;\Rightarrow\; \exists k \geqslant j.\, wait_i \in A_k \quad \text{for each } i \in \{1,2\}.$$

Here, we assumed the set of propositions to be:

$$AP \;=\; \{\, wait_1, crit_1, wait_2, crit_2 \,\}.$$

Property $P_{finwait}$ expresses that each of the two processes enters its critical section eventually if they are waiting. That is, a process has to wait some finite amount before entering the critical section. It does not express that a process that waits often, is often entering the critical section.

Consider the following variant. The LT property $P_{nostarve}$ = set of infinite words $A_0\,A_1\,A_2\ldots$ such that:

$$(\forall k \geqslant 0.\, \exists j \geqslant k.\; wait_i \in A_j\;) \;\Rightarrow\; (\forall k \geqslant 0.\, \exists j \geqslant k.\; crit_i \in A_j\;) \qquad \text{for each } i \in \{1,2\}.$$

In abbreviated form we write:

$$\left(\overset{\infty}{\exists}\; j.\; wait_i \in A_j\right) \;\Rightarrow\; \left(\overset{\infty}{\exists}\; j.\; crit_i \in A_j\right) \qquad \text{for each } i \in \{1,2\}$$

where $\overset{\infty}{\exists}$ stands for "there are infinitely many".

Property $P_{nostarve}$ expresses that each of the two processes enters its critical section infinitely often if they are waiting infinitely often. This natural requirement is, however, *not* satisfied for the semaphore-based solution, since

$$\varnothing\;(\{\, wait_2 \,\}\,\{\, wait_1, wait_2 \,\}\,\{\, crit_1, wait_2 \,\})^{\omega}$$

is a possible trace of the transition system but does not belong to $P_{nostarve}$. This trace represents an execution in which only the first process enters its critical section infinitely often. In fact, the second process waits infinitely long to enter its critical section.

It is left to the reader to check that the transition system modeling Peterson's algorithm (see Example 2.25, page 45) does indeed satisfy $P_{nostarve}$. ■

3.2.4 Trace Equivalence and Linear-Time Properties

LT properties specify the (infinite) traces that a transition system should exhibit. If transition systems TS and TS' have the same traces, one would expect that they satisfy the same LT properties. Clearly, if $TS \models P$, then all traces of TS are contained in P, and when $\mathit{Traces}(TS) = \mathit{Traces}(TS')$, the traces of TS' are also contained in P. Otherwise, whenever $TS \not\models P$, there is a trace in $\mathit{Traces}(TS)$ that is prohibited by P, i.e., not included in the set P of traces. As $\mathit{Traces}(TS) = \mathit{Traces}(TS')$, also TS' exhibits this prohibited trace, and thus $TS' \not\models P$. The precise relationship between trace equivalence, trace inclusion, and the satisfaction of LT properties is the subject of this section.

We start by considering trace inclusion and its importance in concurrent system design. Trace inclusion between transition systems TS and TS' requires that all traces exhibited by TS can also be exhibited by TS', i.e., $\mathit{Traces}(TS) \subseteq \mathit{Traces}(TS')$. Note that transition system TS' may exhibit more traces, i.e., may have some (linear-time) behavior that TS does not have. In stepwise system design, where designs are successively refined, trace inclusion is often viewed as an implementation relation in the sense that

$$\mathit{Traces}(TS) \subseteq \mathit{Traces}(TS') \text{ means } TS \text{ “is a correct implementation of” } TS'.$$

For example, let TS' be a (more abstract) design where parallel composition is modeled by interleaving, and TS its realization where (some of) the interleaving is resolved by means of some scheduling mechanism. TS may thus be viewed as an “implementation” of TS', and clearly, $\mathit{Traces}(TS) \subseteq \mathit{Traces}(TS')$.

What does trace inclusion have to do with LT properties? The following theorem shows that trace inclusion is *compatible* with requirement specifications represented as LT properties.

Theorem 3.15. Trace Inclusion and LT Properties

Let TS and TS' be transition systems without terminal states and with the same set of propositions AP. Then the following statements are equivalent:

(a) $\mathit{Traces}(TS) \subseteq \mathit{Traces}(TS')$

(b) For any LT property P: $TS' \models P$ *implies* $TS \models P$.

Proof: (a) $\Longrightarrow$ (b): Assume $\mathit{Traces}(TS) \subseteq \mathit{Traces}(TS')$, and let P be an LT property such that $TS' \models P$. From Definition 3.11 it follows that $\mathit{Traces}(TS') \subseteq P$. Given $\mathit{Traces}(TS) \subseteq$

$Traces(TS')$, it now follows that $Traces(TS) \subseteq P$. By Definition 3.11 it follows that $TS \models P$.

(b) $\Longrightarrow$ (a): Assume that for all LT properties it holds that: $TS' \models P$ implies $TS \models P$. Let $P = Traces(TS')$. Obviously, $TS' \models P$, as $Traces(TS') \subseteq Traces(TS')$. By assumption, $TS \models P$. Hence, $Traces(TS) \subseteq Traces(TS')$. ■

This simple observation plays a decisive role for the design by means of successive refinement. If TS' is the transition system representing a preliminary design and TS is a transition system originating from a refinement of TS' (i.e., a more detailed design), then it can immediately—without explicit proof—be concluded from the relation $Traces(TS) \subseteq Traces(TS')$ that any LT property that holds in TS' also holds for TS.

Example 3.16. Refining the Semaphore-Based Mutual Exclusion Algorithm

Let $TS' = TS_{Sem}$, the transition system representing the semaphore-based mutual exclusion algorithm (see Figure 3.6 on page 99) and let TS be the transition system obtained from TS' by removing the transition

$$\langle wait_1, wait_2, y = 1\rangle \;\rightarrow\; \langle wait_1, crit_2, y = 0\rangle.$$

Stated in words, from the situation in which both processes are waiting, it is no longer possible that the second process (P_2) acquires access to the critical section. This thus yields a model that assigns higher priority to process P_1 than to process P_2 when both processes are competing to access the critical section. As a transition is removed, it immediately follows that $Traces(TS) \subseteq Traces(TS')$. Consequently, by the fact that TS' ensures mutual exclusion, i.e., $TS' \models P_{mutex}$, it follows by Theorem 3.15 that $TS \models P_{mutex}$. ■

Transition systems are said to be trace-equivalent if they have the same set of traces:

Definition 3.17. Trace Equivalence

Transition systems TS and TS' are *trace-equivalent* with respect to the set of propositions AP if $Traces_{AP}(TS) = Traces_{AP}(TS')$. [4] ■

Theorem 3.15 implies equivalence of two trace-equivalent transition systems with respect to requirements formulated as LT properties.

[4] Here, we assume two transition systems with sets of propositions that include AP.

Corollary 3.18. Trace Equivalence and LT Properties

Let TS and TS' be transition systems without terminal states and with the same set of atomic propositions. Then:

$$Traces(TS) = Traces(TS') \iff \text{TS and TS' satisfy the same LT properties.}$$

There thus does not exist an LT property that can distinguish between trace-equivalent transition systems. Stated differently, in order to establish that the transition systems TS and TS' are *not* trace-equivalent it suffices to find one LT property that holds for one but not for the other.

Example 3.19. Two Beverage Vending Machines

Consider the two transition systems in Figure 3.8 that both model a beverage vending

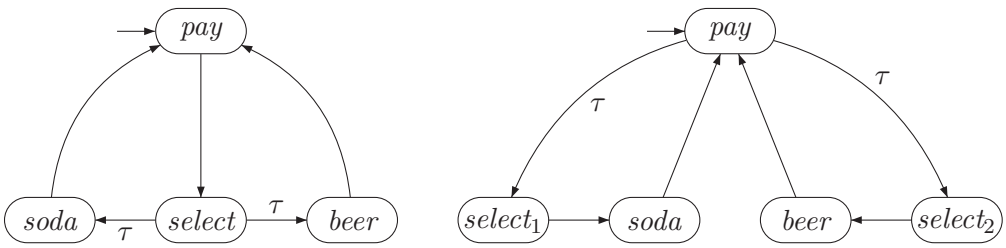


Figure 3.8: Two beverage vending machines.

machine. For simplicity, the observable action labels of transitions have been omitted. Both machines are able to offer soda and beer. The left transition system models a beverage machine that after insertion of a coin nondeterministically chooses to either provide soda or beer. The right one, however, has two selection buttons (one for each beverage), and after insertion of a coin, nondeterministically blocks one of the buttons. In either case, the user has no control over the beverage obtained—the choice of beverage is under full control of the vending machine.

Let $AP = \{ pay, soda, beer \}$. Although the two vending machines behave differently, it is not difficult to see that they exhibit the same traces when considering AP, as for both machines traces are alternating sequences of *pay* and either *soda* or *beer*. The vending machines are thus trace-equivalent. By Corollary 3.18 both vending machines satisfy exactly the same LT properties. Stated differently, it means that there does not exist an LT property that distinguishes between the two vending machines. ■

3.3 Safety Properties and Invariants

Safety properties are often characterized as "nothing bad should happen". The mutual exclusion property—always at most one process is in its critical section—is a typical safety property. It states that the bad thing (having two or more processes in their critical section simultaneously) never occurs. Another typical safety property is deadlock freedom. For the dining philosophers (see Example 3.2, page 90), for example, such deadlock could be characterized as the situation in which all philosophers are waiting to pick up the second chopstick. This bad (i.e., unwanted) situation should never occur.

3.3.1 Invariants

In fact, the above safety properties are of a particular kind: they are *invariants*. Invariants are LT properties that are given by a condition Φ for the states and require that Φ holds for all reachable states.

Definition 3.20. Invariant

An LT property P_{inv} over AP is an *invariant* if there is a propositional logic formula[5] Φ over AP such that

$$P_{inv} = \Big\{ \; A_0 A_1 A_2 \ldots \in (2^{AP})^\omega \mid \; \forall j \geqslant 0.\; A_j \models \Phi \; \Big\}.$$

Φ is called an invariant condition (or state condition) of P_{inv}. ■

Note that

$$\begin{array}{lll} TS \models P_{inv} & \text{iff} & \mathit{trace}(\pi) \in P_{inv} \text{ for all paths } \pi \text{ in } TS \\ & \text{iff} & L(s) \models \Phi \text{ for all states } s \text{ that belong to a path of } TS \\ & \text{iff} & L(s) \models \Phi \text{ for all states } s \in \mathit{Reach}(TS). \end{array}$$

Thus, the notion "invariant" can be explained as follows: the condition Φ has to be fulfilled by all initial states and satisfaction of Φ is invariant under all transitions in the reachable fragment of the given transition system. The latter means that if Φ holds for the source state s of a transition $s \xrightarrow{a} s'$, then Φ holds for the target state s' too.

Let us return to the examples of mutual exclusion and deadlock freedom for the dining philosophers. The mutual exclusion property can be described by an invariant using the

[5] The basic principles of propositional logic are treated in Appendix A.3.

propositional logic formula

$$\Phi \;=\; \neg \mathit{crit}_1 \vee \neg \mathit{crit}_2.$$

For deadlock freedom of the dining philosophers, the invariant ensures that at least one of the philosophers is not waiting to pick up the chopstick. This can be established using the propositional formula:

$$\Phi \;=\; \neg \mathit{wait}_0 \vee \neg \mathit{wait}_1 \vee \neg \mathit{wait}_2 \vee \neg \mathit{wait}_3 \vee \neg \mathit{wait}_4.$$

Here, the proposition wait_i characterizes the state(s) of philosopher i in which he is waiting for a chopstick.

How do we check whether a transition system satisfies an invariant? As checking an invariant for the propositional formula Φ amounts to checking the validity of Φ in every state that is reachable from some initial state, a slight modification of standard graph traversal algorithms like depth-first search (DFS) or breadth-first search (BFS) will do, provided the given transition system TS is *finite*.

Algorithm 3 on page 109 summarizes the main steps for checking the invariant condition Φ by means of a forward depth-first search in the state graph $G(TS)$. The notion *forward search* means that we start from the initial states and investigate all states that are reachable from them. If at least one state s is visited where Φ does not hold, then the invariance induced by Φ is violated. In Algorithm 3, R stores all visited states, i.e., if Algorithm 3 terminates, then $R = \mathit{Reach}(TS)$ contains all reachable states. Furthermore, U is a stack that organizes all states that still have to be visited, provided they are not yet contained in R. The operations *push*, *pop*, and *top* are the standard operations on stacks. The symbol ε is used to denote the empty stack. Alternatively, a *backward search* could have been applied that starts with all states where Φ does not hold and calculates (by a DFS or BFS) the set $\bigcup_{s \in S, s \not\models \Phi} \mathit{Pre}^*(s)$.

Algorithm 3 could be slightly improved by aborting the computation once a state s is encountered that does not fulfill Φ. This state is a "bad" state as it makes the transition system refute the invariant and could be returned as an error indication. Such error indication, however, is not very helpful.

Instead, an initial path fragment $s_0\, s_1\, s_2 \ldots s_n$ in which all states (except the last one) satisfy Φ and $s_n \not\models \Phi$ would be more useful. Such a path fragment indicates a possible behavior of the transition system that violates the invariant. Algorithm 3 can be easily adapted such that a counterexample is provided on encountering a state that violates Φ. To that end we exploit the (depth-first search) stack U. When encountering s_n that violates Φ, the stack content, read from bottom to top, contains the required initial path fragment. Algorithm 4 on page 110 thus results.

Algorithm 3 Naïve invariant checking by forward depth-first search

Input: finite transition system TS and propositional formula Φ
Output: true if TS satisfies the invariant "always Φ", otherwise false

```
set of  state R := ∅;                                  (* the set of visited states *)
stack of  state U := ε;                                (* the empty stack *)
bool b := true;                                        (* all states in R satisfy Φ *)
for all s ∈ I do
  if s ∉ R then
    visit(s)                                           (* perform a dfs for each unvisited initial state *)
  fi
od
return b
```

```
procedure visit (state s)
  push(s, U);                                          (* push s on the stack *)
  R := R ∪ { s };                                      (* mark s as reachable *)
  repeat
    s' := top(U);
    if Post(s') ⊆ R then
      pop(U);
      b := b ∧ (s' ⊨ Φ);                               (* check validity of Φ in s' *)
    else
      let s'' ∈ Post(s') \ R
      push(s'', U);
      R := R ∪ { s'' };                                (* state s'' is a new reachable state *)
    fi
  until (U = ε)
endproc
```

Algorithm 4 Invariant checking by forward depth-first search

Input: finite transition system TS and propositional formula Φ
Output: "yes" if $TS \models$ "always Φ", otherwise "no" plus a counterexample

```
set of states R := ∅;                          (* the set of reachable states *)
stack of states U := ε;                        (* the empty stack *)
bool b := true;                                (* all states in R satisfy Φ *)
while (I \ R ≠ ∅ ∧ b) do
  let s ∈ I \ R;                               (* choose an arbitrary initial state not in R *)
  visit(s);                                    (* perform a DFS for each unvisited initial state *)
od
if b then
  return("yes")                                (* TS ⊨ "always Φ" *)
else
  return("no", reverse(U))                     (* counterexample arises from the stack content *)
fi
```

```
procedure visit (state s)
  push(s, U);                                  (* push s on the stack *)
  R := R ∪ { s };                              (* mark s as reachable *)
  repeat
    s' := top(U);
    if Post(s') ⊆ R then
      pop(U);
      b := b ∧ (s' ⊨ Φ);                       (* check validity of Φ in s' *)
    else
      let s'' ∈ Post(s') \ R
      push(s'', U);
      R := R ∪ { s'' };                        (* state s'' is a new reachable state *)
    fi
  until ((U = ε) ∨ ¬b)
endproc
```

The worst-case time complexity of the proposed invariance checking algorithm is dominated by the cost for the DFS that visits all reachable states. The latter is linear in the number of states (nodes of the state graph) and transitions (edges in the state graph), provided we are given a representation of the state graph where the direct successors $s' \in \mathit{Post}(s)$ for any state s can be encountered in time $\Theta(|\mathit{Post}(s)|)$. This holds for a representation of the sets $\mathit{Post}(s)$ by adjacency lists. An explicit representation of adjacency lists is not adequate in our context where the state graph of a complex system has to be analyzed. Instead, the adjacency lists are typically given in an *implicit* way, e.g., by a syntactic description of the concurrent processes, such as program graphs or higher-level description languages with a program graph semantics such as nanoPromela, see Section 2.2.5, page 63). The direct successors of a state s are then obtained by the axioms and rules for the transition relation for the composite system. Besides the space for the syntactic descriptions of the processes, the space required by Algorithm 4 is dominated by the representation of the set R of visited states (this is typically done by appropriate hash techniques) and stack U. Hence, the additional space complexity of invariant checking is linear in the number of reachable states.

Theorem 3.21. Time Complexity of Invariant Checking

The time complexity of Algorithm 4 is $\mathcal{O}(\, N * (1 + |\Phi|) + M \,)$ *where* N *denotes the number of reachable states, and* $M = \sum_{s \in S} |\mathit{Post}(s)|$ *the number of transitions in the reachable fragment of TS.*

Proof: The time complexity of the forward reachability on the state graph $G(TS)$ is $\mathcal{O}(N + M)$. The time needed to check $s \models \Phi$ for some state s is linear in the length of Φ.[6] As for each state s it is checked whether Φ holds, this amounts to a total of $N + M + N * (1 + |\Phi|)$ operations. ■

3.3.2 Safety Properties

As we have seen in the previous section, invariants can be viewed as state properties and can be checked by considering the reachable states. Some safety properties, however, may impose requirements on finite path fragments, and cannot be verified by considering the reachable states only. To see this, consider the example of a cash dispenser, also known as an automated teller machine (ATM). A natural requirement is that money can only be withdrawn from the dispenser once a correct personal identifier (PIN) has been provided. This property is not an invariant, since it is not a state property. It is, however, considered

[6] To cover the special case where Φ is an atomic proposition, in which case $|\Phi| = 0$, we deal with $1 + |\Phi|$ for the cost to check whether Φ holds for a given state s.

to be a safety property, as any infinite run violating the requirement has a finite prefix that is "bad", i.e., in which money is withdrawn without issuing a PIN before.

Formally, safety property P is defined as an LT property over AP such that any infinite word σ where P does not hold contains a *bad prefix*. The latter means a finite prefix $\widehat{\sigma}$ where the bad thing has happened, and thus no infinite word that starts with this prefix $\widehat{\sigma}$ fulfills P.

Definition 3.22. Safety Properties, Bad Prefixes

An LT property P_{safe} over AP is called a *safety* property if for all words $\sigma \in (2^{AP})^\omega \setminus P_{safe}$ there exists a finite prefix $\widehat{\sigma}$ of σ such that

$$P_{safe} \cap \left\{ \sigma' \in (2^{AP})^\omega \mid \widehat{\sigma} \text{ is a finite prefix of } \sigma' \right\} \;=\; \varnothing.$$

Any such finite word $\widehat{\sigma}$ is called a *bad prefix* for P_{safe}. A *minimal* bad prefix for P_{safe} denotes a bad prefix $\widehat{\sigma}$ for P_{safe} for which no proper prefix of $\widehat{\sigma}$ is a bad prefix for P_{safe}. In other words, minimal bad prefixes are bad prefixes of minimal length. The set of all bad prefixes for P_{safe} is denoted by $\mathit{BadPref}(P_{safe})$, the set of all minimal bad prefixes by $\mathit{MinBadPref}(P_{safe})$. ■

Let us first observe that any invariant is a safety property. For propositional formula Φ over AP and its invariant P_{inv}, all finite words of the form

$$A_0\, A_1\, \ldots A_n \;\in\; (2^{AP})^+$$

with $A_0 \models \Phi, \ldots, A_{n-1} \models \Phi$ and $A_n \not\models \Phi$ constitute the minimal bad prefixes for P_{inv}. The following two examples illustrate that there are safety properties that are not invariants.

Example 3.23. A Safety Property for a Traffic Light

We consider a specification of a traffic light with the usual three phases "red", "green", and "yellow". The requirement that each red phase should be immediately preceded by a yellow phase is a safety property but not an invariant. This is shown in the following.

Let *red*, *yellow*, and *green* be atomic propositions. Intuitively, they serve to mark the states describing a red (yellow or green) phase. The property "always at least one of the lights is on" is specified by:

$$\{\, \sigma = A_0\, A_1 \ldots \mid A_j \subseteq AP \;\wedge\; A_j \neq \varnothing \,\}.$$

The bad prefixes are finite words that contain $\varnothing$. A minimal bad prefix ends with $\varnothing$. The property "it is never the case that two lights are switched on at the same time" is specified

by

$$\{\, \sigma = A_0\, A_1 \ldots \mid A_j \subseteq AP \ \wedge\ |A_j| \leqslant 1 \,\}.$$

Bad prefixes for this property are words containing sets such as $\{\, red, green \,\}$, $\{\, red, yellow \,\}$, and so on. Minimal bad prefixes end with such sets.

Now let $AP' = \{\, red, yellow \,\}$. The property "a red phase must be preceded immediately by a yellow phase" is specified by the set of infinite words $\sigma = A_0\, A_1 \ldots$ with $A_i \subseteq \{\, red, yellow \,\}$ such that for all $i \geqslant 0$ we have that

$$red \in A_i \text{ implies } i > 0 \text{ and } yellow \in A_{i-1}.$$

The bad prefixes are finite words that violate this condition. An example of bad prefixes that are minimal is:

$$\varnothing\, \varnothing\, \{\, red \,\} \quad \text{and} \quad \varnothing\, \{\, red \,\}.$$

The following bad prefix is not minimal:

$$\{\, yellow \,\}\, \{\, yellow \,\}\, \{\, red \,\}\, \{\, red \,\}\, \varnothing\, \{\, red \,\}$$

since it has a proper prefix $\{\, yellow \,\}\, \{\, yellow \,\}\, \{\, red \,\}\, \{\, red \,\}$ which is also a bad prefix.

The minimal bad prefixes of this safety property are regular in the sense that they constitute a regular language. The finite automaton in Figure 3.9 accepts precisely the minimal bad prefixes for the above safety property.[7] Here, $\neg yellow$ should be read as either $\varnothing$ or $\{\, red \,\}$. Note the other properties given in this example are also regular. ■

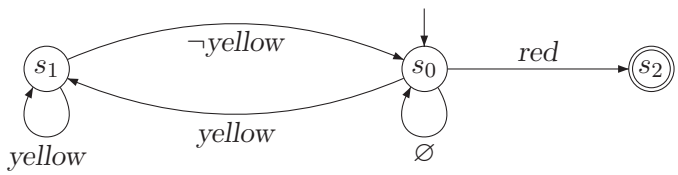


Figure 3.9: A finite automaton for the minimal bad prefixes of a regular safety property.

Example 3.24. A Safety Property for a Beverage Vending Machine

For a beverage vending machine, a natural requirement is that

"The number of inserted coins is always at least the number of dispensed drinks."

[7] The main concepts of a finite automaton as acceptors for languages over finite words are summarized in Section 4.1.

Using the set of propositions $\{ pay, drink \}$ and the obvious labeling function, this property could be formalized by the set of infinite words $A_0 A_1 A_2 \ldots$ such that for all $i \geqslant 0$ we have

$$| \{ 0 \leqslant j \leqslant i \mid pay \in A_j \} | \;\geqslant\; | \{ 0 \leqslant j \leqslant i \mid drink \in A_j \} |$$

Bad prefixes for this safety property are, for example

$$\varnothing \{ pay \} \{ drink \} \{ drink \} \quad \text{and}$$
$$\varnothing \{ pay \} \{ drink \} \varnothing \{ pay \} \{ drink \} \{ drink \}$$

It is left to the interested reader to check that both beverage vending machines from Figure 3.8 satisfy the above safety property. ■

Safety properties are requirements for the finite traces which is formally stated in the following lemma:

Lemma 3.25. Satisfaction Relation for Safety Properties

For transition system TS without terminal states and safety property P_{safe}*:*

$$TS \models P_{safe} \;\; \textit{if and only if} \;\; Traces_{fin}(TS) \cap BadPref(P_{safe}) = \varnothing.$$

Proof: "if": By contradiction. Let $Traces_{fin}(TS) \cap BadPref(P_{safe}) = \varnothing$ and assume that $TS \not\models P_{safe}$. Then, $trace(\pi) \notin P_{safe}$ for some path π in TS. Thus, $trace(\pi)$ starts with a bad prefix $\widehat{\sigma}$ for P_{safe}. But then, $\widehat{\sigma} \in Traces_{fin}(TS) \cap BadPref(P_{safe})$. Contradiction.

"only if": By contradiction. Let $TS \models P_{safe}$ and assume that $\widehat{\sigma} \in Traces_{fin}(TS) \cap BadPref(P_{safe})$. The finite trace $\widehat{\sigma} = A_1 \ldots A_n \in Traces_{fin}(TS)$ can be extended to an infinite trace $\sigma = A_1 \ldots A_n A_{n+1} A_{n+2} \ldots \in Traces(TS)$. Then, $\sigma \notin P_{safe}$ and thus, $TS \not\models P_{safe}$. ■

We conclude this section with an alternative characterization of safety properties by means of their closure.

Definition 3.26. Prefix and Closure

For trace $\sigma \in (2^{AP})^\omega$, let $pref(\sigma)$ denote the set of finite prefixes of σ, i.e.,

$$pref(\sigma) = \{ \widehat{\sigma} \in (2^{AP})^* \mid \widehat{\sigma} \text{ is a finite prefix of } \sigma \}.$$

that is, if $\sigma = A_0\,A_1\,\ldots$ then $\mathit{pref}(\sigma) = \{\varepsilon, A_0, A_0A_1, A_0A_1A_2, \ldots\}$ is an infinite set of finite words. This notion is lifted to sets of traces in the usual way. For property P over AP:

$$\mathit{pref}(P) = \bigcup_{\sigma \in P} \mathit{pref}(\sigma).$$

The *closure* of LT property P is defined by

$$\mathit{closure}(P) \;=\; \{\sigma \in (2^{AP})^\omega \mid \mathit{pref}(\sigma) \subseteq \mathit{pref}(P)\}.$$

■

For instance, for infinite trace $\sigma = ABABAB\ldots$ (where $A, B \subseteq AP$) we have $\mathit{pref}(\sigma) = \{\,\varepsilon, A, AB, ABA, ABAB, \ldots\}$ which equals the regular language given by the regular expression $(AB)^*(A + \varepsilon)$.

The *closure* of an LT property P is the set of infinite traces whose finite prefixes are also prefixes of P. Stated differently, infinite traces in the closure of P do not have a prefix that is not a prefix of P itself. As we will see below, the closure is a key concept in the characterization of safety and liveness properties.

Lemma 3.27. Alternative Characterization of Safety Properties

Let P be an LT property over AP. Then, P is a safety property iff $\mathit{closure}(P) = P$.

Proof: "if": Let us assume that $\mathit{closure}(P) = P$. To show that P is a safety property, we take an element $\sigma \in (2^{AP})^\omega \setminus P$ and show that σ starts with a bad prefix for P. Since $\sigma \notin P = \mathit{closure}(P)$ there exists a finite prefix $\widehat{\sigma}$ of σ with $\widehat{\sigma} \notin \mathit{pref}(P)$. By definition of $\mathit{pref}(P)$, none of the words $\sigma' \in (2^{AP})^\omega$ where $\widehat{\sigma} \in \mathit{pref}(\sigma')$ belongs to P. Hence, $\widehat{\sigma}$ is a bad prefix for P, and by definition, P is a safety property.

"only if": Let us assume that P is a safety property. We have to show that $P = \mathit{closure}(P)$. The inclusion $P \subseteq \mathit{closure}(P)$ holds for all LT properties. It remains to show that $\mathit{closure}(P) \subseteq P$. We do so by contradiction. Let us assume that there is some $\sigma = A_1\,A_2\,\ldots \in \mathit{closure}(P) \setminus P$. Since P is a safety property and $\sigma \notin P$, σ has a finite prefix

$$\widehat{\sigma} = A_1\,\ldots\,A_n \in \mathit{BadPref}(P).$$

As $\sigma \in \mathit{closure}(P)$ we have $\widehat{\sigma} \in \mathit{pref}(\sigma) \subseteq \mathit{pref}(P)$. Hence, there exists a word $\sigma' \in P$ of the form

$$\sigma' = \underbrace{A_1\,\ldots\,A_n}_{\text{bad prefix}}\,B_{n+1}\,B_{n+2}\,\ldots$$

This contradicts the fact that P is a safety property. ■

3.3.3 Trace Equivalence and Safety Properties

We have seen before that there is a strong relationship between trace inclusion of transition systems and the satisfaction of LT properties (see Theorem 3.15, page 104):

$$\mathit{Traces}(TS) \subseteq \mathit{Traces}(TS') \quad \text{if and only if} \quad \begin{array}{l}\text{for all LT properties } P\text{:}\\ TS' \models P \text{ implies } TS \models P\end{array}$$

for transition systems TS and TS' without terminal states. Note that this result considers all *infinite* traces. The above thus states a relationship between infinite traces of transition systems and the validity of LT properties. When considering only finite traces instead of infinite ones, a similar connection with the validity of safety properties can be established, as stated by the following theorem.

Theorem 3.28. Finite Trace Inclusion and Safety Properties

Let TS and TS' be transition systems without terminal states and with the same set of propositions AP. Then the following statements are equivalent:

(a) $\mathit{Traces}_{\mathit{fin}}(TS) \subseteq \mathit{Traces}_{\mathit{fin}}(TS')$,

(b) For any safety property P_{safe}*:* $TS' \models P_{\mathit{safe}}$ *implies* $TS \models P_{\mathit{safe}}$.

Proof:

(a) $\Longrightarrow$ (b): Let us assume that $\mathit{Traces}_{\mathit{fin}}(TS) \subseteq \mathit{Traces}_{\mathit{fin}}(TS')$ and let P_{safe} be a safety property with $TS' \models P_{\mathit{safe}}$. By Lemma 3.25, we have $\mathit{Traces}_{\mathit{fin}}(TS') \cap \mathit{BadPref}(P_{\mathit{safe}}) = \varnothing$, and hence, $\mathit{Traces}_{\mathit{fin}}(TS) \cap \mathit{BadPref}(P_{\mathit{safe}}) = \varnothing$. Again by Lemma 3.25, we get $TS \models P_{\mathit{safe}}$.

(b) $\Longrightarrow$ (a): Assume that (b) holds. Let $P_{\mathit{safe}} = \mathit{closure}(\mathit{Traces}(TS'))$. Then, P_{safe} is a safety property and we have $TS' \models P_{\mathit{safe}}$ (see Exercise 3.9, page 147). Hence, (b) yields $TS \models P_{\mathit{safe}}$, i.e.,

$$\mathit{Traces}(TS) \subseteq \mathit{closure}(\mathit{Traces}(TS')).$$

From this, we may derive

$$\begin{array}{rcl}\mathit{Traces}_{\mathit{fin}}(TS) & = & \mathit{pref}(\mathit{Traces}(TS))\\ & \subseteq & \mathit{pref}(\mathit{closure}(\mathit{Traces}(TS'))\\ & = & \mathit{pref}(\mathit{Traces}(TS'))\\ & = & \mathit{Traces}_{\mathit{fin}}(TS').\end{array}$$

Here we use the property that for any P it holds that $\mathit{pref}(\mathit{closure}(P)) = \mathit{pref}(P)$ (see Exercise 3.10, page 147). ■

Theorem 3.28 is of relevance for the gradual design of concurrent systems. If a preliminary design (i.e., a transition system) TS' is refined to a design TS such that

$$\mathit{Traces}(TS) \not\subseteq \mathit{Traces}(TS'),$$

then the LT properties of TS' cannot be carried over to TS. However, if the finite traces of TS are finite traces of TS' (which is a weaker requirement than full trace inclusion of TS and TS'), i.e.,

$$\mathit{Traces}_{\mathit{fin}}(TS) \subseteq \mathit{Traces}_{\mathit{fin}}(TS'),$$

then all safety properties that have been established for TS' also hold for TS. Other requirements for TS, i.e., LT properties that fall outside the scope of safety properties, need to be checked using different techniques.

Corollary 3.29. Finite Trace Equivalence and Safety Properties

Let TS and TS' be transition systems without terminal states and with the same set AP of atomic propositions. Then, the following statements are equivalent:

(a) $\mathit{Traces}_{\mathit{fin}}(TS) = \mathit{Traces}_{\mathit{fin}}(TS')$,

(b) For any safety property P_{safe} over AP: $TS \models P_{safe} \Longleftrightarrow TS' \models P_{safe}$.

A few remarks on the difference between finite trace inclusion and trace inclusion are in order. Since we assume transition systems without terminal states, there is only a slight difference between trace inclusion and finite trace inclusion. For *finite* transition systems TS and TS' without terminal states, trace inclusion and finite trace inclusion coincide. This can be derived from the following theorem.

Theorem 3.30. Relating Finite Trace and Trace Inclusion

Let TS and TS' be transition systems with the same set AP of atomic propositions such that TS has no terminal states and TS' is finite. Then:

$$\mathit{Traces}(TS) \subseteq \mathit{Traces}(TS') \iff \mathit{Traces}_{\mathit{fin}}(TS) \subseteq \mathit{Traces}_{\mathit{fin}}(TS').$$

Proof: The implication from left to right follows from the monotonicity of $\mathit{pref}(\cdot)$ and the fact that $\mathit{Traces}_{\mathit{fin}}(TS) = \mathit{pref}(\mathit{Traces}(TS))$ for any transition system TS.

It remains to consider the proof for the implication $\Longleftarrow$. Let us assume that $\mathit{Traces}_{\mathit{fin}}(TS) \subseteq \mathit{Traces}_{\mathit{fin}}(TS')$. As TS has no terminal states, all traces of TS are infinite. Let $A_0A_1 \ldots \in$

$\mathit{Traces}(TS)$. To prove that $A_0A_1\ldots \in \mathit{Traces}(TS')$ we have to show that there exists a path in TS', say $s_0\,s_1\ldots$, that generates this trace, i.e., $\mathit{trace}(s_0\,s_1\ldots) = A_0\,A_1\ldots$.

Any finite prefix $A_0\,A_1\ldots A_m$ of the infinite trace $A_0A_1\ldots$ is in $\mathit{Traces}_{\mathit{fin}}(TS)$, and as $\mathit{Traces}_{\mathit{fin}}(TS) \subseteq \mathit{Traces}_{\mathit{fin}}(TS')$, also in $\mathit{Traces}_{\mathit{fin}}(TS')$. Thus, for any natural number m, there exists a finite path $\pi^m = s_0^m s_1^m \ldots s_m^m$ in TS' such that

$$\mathit{trace}(\pi^m) \;=\; L(s_0^m)L(s_1^m)\ldots L(s_m^m) = A_0A_1\ldots A_m$$

where L denotes the labeling function of TS'. Thus, $L(s_j^m) = A_j$ for all $0 \leqslant j \leqslant m$.

Although $A_0\ldots A_m$ is a prefix of $A_0\ldots A_{m+1}$, it is not guaranteed that path π^m is a prefix of π^{m+1}. Due to the finiteness of TS', however, there is an infinite subsequence $\pi^{m_0}\,\pi^{m_1}\,\pi^{m_2}\ldots$ of $\pi^0\,\pi^1\,\pi^2\ldots$ such that π^{m_i} and $\pi^{m_{i+1}}$ agree on the first i states. Thus, $\pi^{m_0}\,\pi^{m_1}\,\pi^{m_2}\ldots$ induces an infinite path π in TS' with the desired property.

This is formally proven using a so-called *diagonalization technique*. This goes as follows. Let $I_0, I_1, I_2, \ldots$ be an infinite series of infinite sets of indices (i.e., natural numbers) with $I_n \subseteq \{\, m \in \mathbb{N} \mid m \geqslant n \,\}$ and $s_0, s_1, \ldots$ be states in TS' such that for all natural numbers n it holds that

(1) $n \geqslant 1$ implies $I_{n-1} \supseteq I_n$,

(2) $s_0\,s_1\,s_2\,\ldots\,s_n$ is an initial, finite path fragment in TS',

(3) for all $m \in I_n$ it holds that $s_0\,\ldots\,s_n = s_0^m\,\ldots\,s_n^m$.

The definition of the sets I_n and states s_n is by induction on n.

Base case ($n = 0$): As $\{\, s_0^m \mid m \in \mathbb{N} \,\}$ is finite (since it is a subset of the finite set of initial states of TS'), there exists an initial state s_0 in TS' and an infinite index set I_0 such that $s_0 = s_0^m$ for all $m \in I_0$.

Induction step $n \Longrightarrow n{+}1$. Assume that the index sets $I_0, \ldots, I_n$ and states $s_0, \ldots, s_n$ are defined. Since TS' is finite, $\mathit{Post}(s_n)$ is finite. Furthermore, by the induction hypothesis $s_n = s_n^m$ for all $m \in I_n$, and thus

$$\{\, s_{n+1}^m \mid m \in I_n, m \geqslant n{+}1 \,\} \;\subseteq\; \mathit{Post}(s_n).$$

Since I_n is infinite, there exists an infinite subset $I_{n+1} \subseteq \{\, m \in I_n \mid m \geqslant n{+}1 \,\}$ and a state $s_{n+1} \in \mathit{Post}(s_n)$ such that $s_{n+1}^m = s_{n+1}$ for all $m \in I_{n+1}$. It follows directly that the above properties (1) through (3) are fulfilled.

We now consider the state sequence $s_0\, s_1\, \ldots$ in TS'. Obviously, this state sequence is a path in TS' satisfying $trace(s_0\, s_1\, \ldots) = A_0\, A_1\, \ldots$. Consequently, $A_0\, A_1\, \ldots \in Traces(TS')$. ■

Remark 3.31. Image-Finite Transition Systems

The result stated in Theorem 3.30 also holds under slightly weaker conditions: it suffices to require that TS has no terminal states (as in Theorem 3.30) and that TS' is AP image-finite (rather than being finite).

Let $TS' = (S, Act, \rightarrow, I, AP, L)$. Then, TS' is called AP image-finite (or briefly *image-finite*) if

(i) for all $A \subseteq AP$, the set $\{\, s_0 \in I \mid L(s_0) = A \,\}$ is finite and

(ii) for all states s in TS' and all $A \subseteq AP$, the set of successors $\{\, s' \in Post(s) \mid L(s') = A \,\}$ is finite.

Thus, any finite transition system is image-finite. Moreover, any transition system that is AP-deterministic is image-finite. (Recall that AP-determinism requires $\{\, s_0 \in I \mid L(s_0) = A \,\}$ and $\{\, s' \in Post(s) \mid L(s') = A \,\}$ to be either singletons or empty sets; see Definition 2.5, page 24.)

In fact, a careful inspection of the proof of Theorem 3.30 shows that (i) and (ii) for TS' are used in the construction of the index sets I_n and states s_n. Hence, we have $Traces(TS) \subseteq Traces(TS')$ iff $Traces_{fin}(TS) \subseteq Traces_{fin}(TS')$, provided TS has no terminal states and TS' is image-finite. ■

Trace and finite trace inclusion, however, coincide neither for infinite transition systems nor for finite ones which have terminal states.

Example 3.32. Finite vs. Infinite Transition System

Consider the transition systems sketched in Figure 3.10, where b stands for an atomic proposition. Transition system TS (on the left) is finite, whereas TS' (depicted on the right) is infinite and not image-finite, because of the infinite branching in the initial state. It is not difficult to observe that

$$Traces(TS) \not\subseteq Traces(TS') \quad \text{and} \quad Traces_{fin}(TS) \subseteq Traces_{fin}(TS').$$

This stems from the fact that TS can take the self-loop infinitely often and never reaches a b-state, whereas TS' does not exhibit such behavior. Moreover, any finite trace of TS is

of the form $(\varnothing)^n$ for $n \geqslant 0$ and is also a finite trace of TS'. Consequently, LT properties of TS' do not carry over to TS (and those of TS may not hold for TS'). For example, the LT property "eventually b" holds for TS', but not for TS. Similarly, the LT property "never b" holds for TS, but not for TS'.

Although these transition systems might seem rather artificial, this is not the case: TS could result from an infinite loop in a program, whereas TS' could model the semantics of a program fragment that nondeterministically chooses a natural number k and then performs k steps. ■

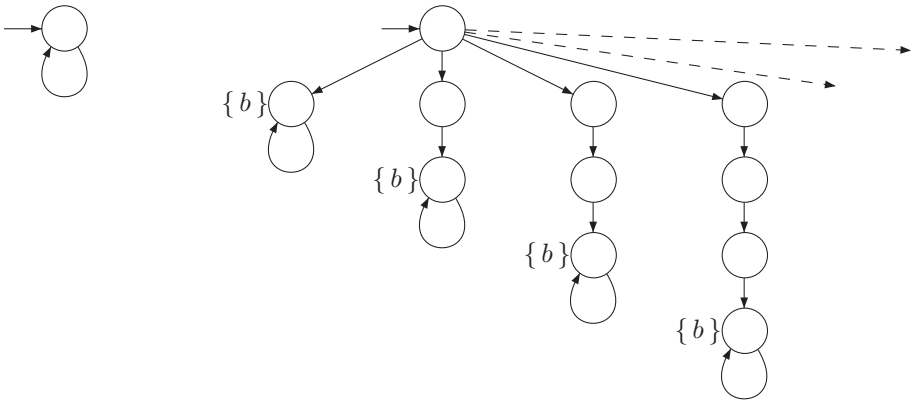


Figure 3.10: Distinguishing trace inclusion from finite trace inclusion.

3.4 Liveness Properties

Informally speaking, safety properties specify that "something bad never happens". For the mutual exclusion algorithm, the "bad" thing is that more than one process is in its critical section, while for the traffic light the "bad" situation is whenever a red light phase is not preceded by a yellow light phase. An algorithm can easily fulfill a safety property by simply doing nothing as this will never lead to a "bad" situation. As this is usually undesired, safety properties are complemented by properties that require some progress. Such properties are called "liveness" properties (or sometimes "progress" properties). Intuitively, they state that "something good" will happen in the future. Whereas safety properties are violated in finite time, i.e., by a finite system run, liveness properties are violated in infinite time, i.e., by infinite system runs.

3.4.1 Liveness Properties

Several (nonequivalent) notions of liveness properties have been defined in the literature. We follow here the approach of Alpern and Schneider [5, 6, 7]. They provided a formal notion of liveness properties which relies on the view that liveness properties do not constrain the finite behaviors, but require a certain condition on the infinite behaviors. A typical example for a liveness property is the requirement that certain events occur infinitely often. In this sense, the "good event" of a liveness property is a condition on the infinite behaviors, while the "bad event" for a safety property occurs in a finite amount of time, if it occurs at all.

In our approach, a liveness property (over AP) is defined as an LT property that does not rule out any prefix. This entails that the set of finite traces of a system are of no use at all to decide whether a liveness property holds or not. Intuitively speaking, it means that any finite prefix can be extended such that the resulting infinite trace satisfies the liveness property under consideration. This is in contrast to safety properties where it suffices to have one finite trace (the "bad prefix") to conclude that a safety property is refuted.

Definition 3.33. Liveness Property

LT property P_{live} over AP is a *liveness* property whenever $pref(P_{live}) = (2^{AP})^*$. ■

Thus, a liveness property (over AP) is an LT property P such that each finite word can be extended to an infinite word that satisfies P. Stated differently, P is a liveness property if and only if for all finite words $w \in (2^{AP})^*$ there exists an infinite word $\sigma \in (2^{AP})^\omega$ satisfying $w\sigma \in P$.

Example 3.34. Repeated Eventually and Starvation Freedom

In the context of mutual exclusion algorithms the natural safety property that is required ensures the mutual exclusion property stating that the processes are never simultaneously in their critical sections. (This is even an invariant.) Typical liveness properties that are desired assert that

- (eventually) each process will eventually enter its critical section;
- (repeated eventually) each process will enter its critical section infinitely often;
- (starvation freedom) each waiting process will eventually enter its critical section.

Let's see how these liveness properties are formalized as LT properties and let us check that

they are liveness properties. As in Example 3.14, we will deal with the atomic propositions $wait_1, crit_1, wait_2, crit_2$ where $wait_i$ characterizes the states where process P_i has requested access to its critical section and is in its waiting phase, while $crit_i$ serves as a label for the states where P_i has entered its critical section. We now formalize the three properties by LT properties over $AP = \{wait_1, crit_1, wait_2, crit_2\}$. The first property (eventually) consists of all infinite words $A_0\,A_1\,\ldots$ with $A_j \subseteq AP$ such that

$$(\exists j \geqslant 0.\ crit_1 \in A_j)\ \wedge\ (\exists j \geqslant 0.\ crit_2 \in A_j)$$

which requires that P_1 and P_2 are in their critical sections at least once. The second property (repeated eventually) poses the condition

$$(\forall k \geqslant 0.\, \exists j \geqslant k.\ crit_1 \in A_j)\ \wedge\ (\forall k \geqslant 0.\, \exists j \geqslant k.\ crit_2 \in A_j)$$

stating that P_1 and P_2 are infinitely often in their critical sections. This formula is often abbreviated by

$$\left(\overset{\infty}{\exists}\ j \geqslant 0.\ crit_1 \in A_j\right)\ \wedge\ \left(\overset{\infty}{\exists}\ j \geqslant 0.\ crit_2 \in A_j\right).$$

The third property (starvation freedom) requires that

$$\begin{aligned}&\forall j \geqslant 0.\ (wait_1 \in A_j\ \Rightarrow\ (\exists k > j.\, crit_1 \in A_k))\ \wedge\\ &\forall j \geqslant 0.\ (wait_2 \in A_j\ \Rightarrow\ (\exists k > j.\, crit_2 \in A_k)).\end{aligned}$$

It expresses that each process that is waiting will acquire access to the critical section at some later time point. Note that here we implicitly assume that a process that starts waiting to acquire access to the critical section does not "give up" waiting, i.e., it continues waiting until it is granted access.

All aforementioned properties are liveness properties, as any finite word over AP is a prefix of an infinite word where the corresponding condition holds. For instance, for starvation freedom, a finite trace in which a process is waiting but never acquires access to its critical section can always be extended to an infinite trace that satisfies the starvation freedom property (by, e.g., providing access in an strictly alternating fashion from a certain point on). ■

3.4.2 Safety vs. Liveness Properties

This section studies the relationship between liveness and safety properties. In particular, it provides answers to the following questions:

- Are safety and liveness properties disjoint?, and

- Is any linear-time property a safety or liveness property?

As we will see, the first question will be answered affirmatively while the second question will result in a negative answer. Interestingly enough, though, for any LT property P an equivalent LT property P' does exist which is a combination (i.e., intersection) of a safety and a liveness property. All in all, one could say that the identification of safety and liveness properties thus provides an essential characterization of linear-time properties.

The first result states that safety and liveness properties are indeed almost disjoint. More precisely, it states that the only property that is both a safety and a liveness property is nonrestrictive, i.e., allows all possible behaviors. Logically speaking, this is the equivalent of "true".

Lemma 3.35. Intersection of Safety and Liveness Properties

The only LT property over AP that is both a safety and a liveness property is $(2^{AP})^\omega$.

Proof: Assume P is a liveness property over AP. By definition, $pref(P) = (2^{AP})^*$. It follows that $closure(P) = (2^{AP})^\omega$. If P is a safety property too, $closure(P) = P$, and hence $P = (2^{AP})^\omega$. ■

Recall that the closure of property P (over AP) is the set of infinite words (over 2^{AP}) for which all prefixes are also prefixes of P. In order to show that an LT property can be considered as a conjunction of a liveness and a safety property, the following result is helpful. It states that the closure of the union of two properties equals the union of their closures.

Lemma 3.36. Distributivity of Union over Closure

For any LT properties P *and* P'*:*

$$closure(P) \cup closure(P') = closure(P \cup P').$$

Proof: $\subseteq$: As $P \subseteq P'$ implies $closure(P) \subseteq closure(P')$, we have $P \subseteq P \cup P'$ implies $closure(P) \subseteq closure(P \cup P')$. In a similar way it follows that $closure(P') \subseteq closure(P \cup P')$. Thus, $closure(P) \cup closure(P') \subseteq closure(P \cup P')$.

$\supseteq$: Let $\sigma \in closure(P \cup P')$. By definition of closure, $pref(\sigma) \subseteq pref(P \cup P')$. As $pref(P \cup P') = pref(P) \cup pref(P')$, any finite prefix of σ is in $pref(P)$ or in $pref(P')$ (or in both).

As $\sigma \in (2^{AP})^\omega$, σ has infinitely many prefixes. Thus, infinitely many finite prefixes of σ belong to $pref(P)$ or to $pref(P')$ (or to both). W.l.o.g., assume $pref(\sigma) \cap pref(P)$ to be infinite. Then $pref(\sigma) \subseteq pref(P)$, which yields $\sigma \in closure(P)$, and thus $\sigma \in closure(P) \cup closure(P')$. The fact that $pref(\sigma) \subseteq pref(P)$ can be shown by contraposition. Assume $\widehat{\sigma} \in pref(\sigma) \setminus pref(P)$. Let $|\widehat{\sigma}| = k$. As $pref(\sigma) \cap pref(P)$ is infinite, there exists $\widehat{\sigma'} \in pref(\sigma) \cap pref(P)$ with length larger than k. But then, there exists $\sigma' \in P$ with $\widehat{\sigma'} \in pref(\sigma')$. It then follows that $\widehat{\sigma} \in pref(\widehat{\sigma'})$ (as both $\widehat{\sigma}$ and $\widehat{\sigma'}$ are prefixes of σ) and as $pref(\widehat{\sigma'}) \subseteq pref(P)$, it follows that $\widehat{\sigma} \in pref(P)$. This contradicts $\widehat{\sigma} \in pref(\sigma) \setminus pref(P)$. ■

Consider the beverage vending machine of Figure 3.5 (on page 97), and the following property:

"the machine provides beer infinitely often
after initially providing soda three times in a row"

In fact, this property consists of two parts. On the one hand, it requires beer to be provided infinitely often. As any finite trace can be extended to an infinite trace that enjoys this property it is a liveness property. On the other hand, the first three drinks it provides should all be soda. This is a safety property, since any finite trace in which one of the first three drinks provided is beer violates it. The property is thus a combination (in fact, a conjunction) of a safety and a liveness property. The following result shows that *every* LT property can be decomposed in this way.

Theorem 3.37. Decomposition Theorem

For any LT property P over AP there exists a safety property P_{safe} and a liveness property P_{live} (both over AP) such that

$$P = P_{safe} \cap P_{live}.$$

Proof: Let P be an LT property over AP. It is easy to see that $P \subseteq closure(P)$. Thus: $P = closure(P) \cap P$, which by set calculus can be rewritten into:

$$P \;=\; \underbrace{closure(P)}_{=P_{safe}} \;\cap\; \underbrace{\Big(P \cup \Big((2^{AP})^\omega \setminus closure(P)\Big)\Big)}_{=P_{live}}$$

By definition, $P_{safe} = closure(P)$ is a safety property. It remains to prove that $P_{live} = P \cup \Big((2^{AP})^\omega \setminus closure(P)\Big)$ is a liveness property. By definition, P_{live} is a liveness property whenever $pref(P_{live}) = (2^{AP})^*$. This is equivalent to $closure(P_{live}) = (2^{AP})^\omega$. As for any

LT property P, $\mathit{closure}(P) \subseteq (2^{AP})^\omega$ holds true, it suffices to showing that $(2^{AP})^\omega \subseteq \mathit{closure}(P_{\mathit{live}})$. This goes as follows:

$$\begin{aligned}
\mathit{closure}(P_{\mathit{live}}) &= \mathit{closure}\Big(P \cup ((2^{AP})^\omega \setminus \mathit{closure}(P))\Big) \\
&\overset{\text{Lemma 3.36}}{=} \mathit{closure}(P) \cup \mathit{closure}\Big((2^{AP})^\omega \setminus \mathit{closure}(P)\Big) \\
&\supseteq \mathit{closure}(P) \cup \Big((2^{AP})^\omega \setminus \mathit{closure}(P)\Big) \\
&= (2^{AP})^\omega
\end{aligned}$$

where in the one-but-last step in the derivation, we exploit the fact that $\mathit{closure}(P') \supseteq P'$ for all LT properties P'. ■

The proof of Theorem 3.37 shows that $P_{\mathit{safe}} = \mathit{closure}(P)$ is a safety property and $P_{\mathit{live}} = P \cup ((2^{AP})^\omega \setminus \mathit{closure}(P))$ a liveness property with $P = P_{\mathit{safe}} \cap P_{\mathit{live}}$. In fact, this decomposition is the "sharpest" one for P since P_{safe} is the *strongest* safety property and P_{live} the *weakest* liveness property that can serve for a decomposition of P:

Lemma 3.38. Sharpest Decomposition

Let P be an LT property and $P = P_{\mathit{safe}} \cap P_{\mathit{live}}$ where P_{safe} is a safety property and P_{live} a liveness property. We then have

1. $\mathit{closure}(P) \subseteq P_{\mathit{safe}}$,
2. $P_{\mathit{live}} \subseteq P \cup ((2^{AP})^\omega \setminus \mathit{closure}(P))$.

Proof: See Exercise 3.12, page 147. ■

A summary of the classification of LT properties is depicted as a Venn diagram in Figure 3.11. The circle denotes the set of all LT properties over a given set of atomic propositions.

Remark 3.39. Topological Characterizations of Safety and Liveness

Let us conclude this section with a remark for readers who are familiar with basic notions of topological spaces. The set $(2^{AP})^\omega$ can be equipped with the distance function given by $d(\sigma_1, \sigma_2) = 1/2^n$ if σ_1, σ_2 are two distinct infinite words $\sigma_1 = A_1 A_2 \ldots$ and $\sigma_2 = B_1 B_2 \ldots$ and n is the length of the longest common prefix. Moreover, we put $d(\sigma, \sigma) = 0$. Then, d is a metric on $(2^{AP})^\omega$, and hence induces a topology on $(2^{AP})^\omega$. Under this topology,

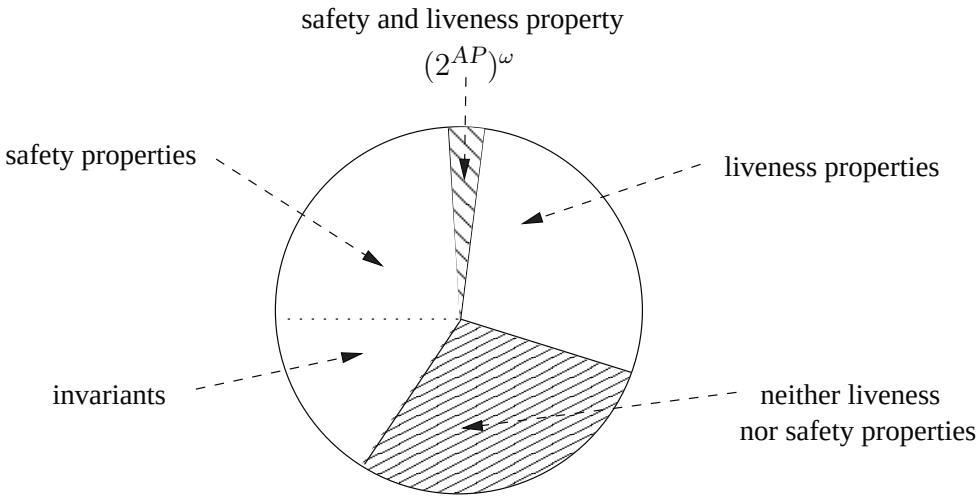


Figure 3.11: Classification of linear-time properties.

the safety properties are exactly the closed sets, while the liveness properties agree with the dense sets. In fact, *closure*(P) is the topological closure of P, i.e., the smallest closed set that contains P. The result stated in Theorem 3.37 then follows from the well-known fact that any subset of a topological space (of the kind described above) can be written as the intersection of its closure and a dense set. ■

3.5 Fairness

An important aspect of reactive systems is fairness. Fairness assumptions rule out infinite behaviors that are considered unrealistic, and are often necessary to establish liveness properties. We illustrate the concept of fairness by means of a frequently encountered problem in concurrent systems.

Example 3.40. Process Fairness

Consider N processes $P_1, \ldots, P_N$ which require a certain service. There is one server process *Server* that is expected to provide services to these processes. A possible strategy that *Server* can realize is the following. Check the processes starting with P_1, then P_2, and so on, and serve the first thus encountered process that requires service. On finishing serving this process, repeat this selection procedure once again starting with checking P_1.

Now suppose that P_1 is continuously requesting service. Then this strategy will result in *Server* always serving P_1. Since in this way another process has to wait infinitely long before being served, this is called an unfair strategy. In a fair serving strategy it is required that the server eventually responds to any request by any one of the processes. For instance, a round-robin scheduling strategy where each process is only served for a limited amount of time is a fair strategy: after having served one process, the next (in the round-robin order) is checked and, if needed, served. ■

When verifying concurrent systems one is often only interested in paths in which enabled transitions (statements) are executed in some "fair" manner. Consider, for instance, a mutual exclusion algorithm for two processes. In order to prove starvation freedom, the situation in which a process that wants to enter its critical section has to wait infinitely long, we want to exclude those paths in which the competitor process is always being selected for execution. This type of fairness is also known as *process fairness*, since it concerns the fair scheduling of the execution of processes. If we were to consider unfair paths when proving starvation freedom, we would usually fail, since there always exists an unfair strategy according to which some process is always neglected, and thus can never make progress. One might argue that such unfair strategy is unrealistic and should be avoided.

Example 3.41. Starvation Freedom

Consider the transition systems TS_{Sem} and TS_{Pet} for the semaphore-based mutual exclusion algorithms (see Example 2.24 on page 43) and Peterson's algorithm. The starvation freedom property

"Once access is requested, a process does not have to wait infinitely long before acquiring access to its critical section"

is violated by transition system TS_{Sem} while it permits only one of the processes to proceed, while the other process is starving (or only acquiring access to the critical section finitely often). The transition system TS_{Pet} for Peterson's algorithm, however, fulfills this property.

The property

"Each of the processes is infinitely often in its critical section"

is violated by both transition systems as none of them excludes the fact that a process would never (or only finitely often) request to enter the critical section. ■

Process fairness is a particular form of fairness. In general, fairness assumptions are needed to prove liveness or other properties stating that the system makes some progress ("something good will eventually happen"). This is of vital importance if the transition system to be checked contains nondeterminism. Fairness is then concerned with resolving nondeterminism in such a way that it is not biased to consistently ignore a possible option. In the above example, the scheduling of processes is nondeterministic: the choice of the next process to be executed (if there are at least two processes that can be potentially selected) is arbitrary. Another prominent example where fairness is used to "resolve" nondeterminism is in modeling concurrent processes by means of interleaving. Interleaving is equivalent to modeling the concurrent execution of two independent processes by enumerating all the possible orders in which activities of the processes can be executed (see Chapter 2).

Example 3.42. Independent Traffic Lights

Consider the transition system

$$TS \;=\; \mathit{TrLight}_1 \;|||\; \mathit{TrLight}_2$$

for the two independent traffic lights described in Example 2.17 (page 36). The liveness property

"Both traffic lights are infinitely often green"

is not satisfied, since

$$\{\, \mathit{red}_1, \mathit{red}_2 \,\}\, \{\, \mathit{green}_1, \mathit{red}_2 \,\}\, \{\, \mathit{red}_1, \mathit{red}_2 \,\}\, \{\, \mathit{green}_1, \mathit{red}_2 \,\} \ldots$$

is a trace of TS where only the first traffic light is infinitely often green. ■

What is wrong with the above examples? In fact, nothing. Let us explain this. In the traffic light example, the information whether each traffic light switches color infinitely often is lost by means of interleaving. The trace in which only the first traffic light is acting while the second light seems to be completely stopped is formally a trace of the transition system $\mathit{TrLight}_1 \,|||\, \mathit{TrLight}_2$. However, it does not represent a realistic behavior as in practice no traffic light is infinitely faster than another.

For the semaphore-based mutual exclusion algorithm, the difficulty is the degree of abstraction. A semaphore is not a willful individual that arbitrarily chooses a process which is authorized to enter its critical section. Instead, the waiting processes are administered in a queue (or another "fair" medium). The required liveness can be proven in one of the following refinement steps, in which the specification of the behavior of the semaphore is sufficiently detailed.

3.5.1 Fairness Constraints

The above considerations show that we—to obtain a realistic picture of the behavior of a parallel system modeled by a transition system—need a more alleviated form of satisfaction relation for LT properties, which implies an "adequate" resolution of the nondeterministic decisions in a transition system. In order to rule out the unrealistic computations, *fairness constraints* are imposed.

In general, a fair execution (or trace) is characterized by the fact that certain fairness constraints are fulfilled. Fairness constraints are used to rule out computations that are considered to be unreasonable for the system under consideration. Fairness constraints come in different flavors:

- *Unconditional fairness*: e.g.,"Every process gets its turn infinitely often."
- *Strong fairness*: e.g., "Every process that is enabled infinitely often gets its turn infinitely often."
- *Weak fairness:* e.g., "Every process that is continuously enabled from a certain time instant on gets its turn infinitely often."

Here, the term "is enabled" has to be understood in the sense of "ready to execute (a transition)". Similarly, "gets its turn" stands for the execution of an arbitrary transition. This can, for example, be a noncritical action, acquiring a shared resource, an action in the critical section, or a communication action.

An execution fragment is unconditionally fair with respect to, e.g., "a process enters its critical section" or "a process gets its turn", if these properties hold infinitely often. That is to say, a process enters its critical section infinitely often, or, in the second example, a process gets its turn infinitely often. Note that no condition (such as "a process is enabled") is expressed that constrains the circumstances under which a process gets its turn infinitely often. Unconditional fairness is sometimes referred to as impartiality.

Strong fairness means that if an activity is infinitely often enabled—but not necessarily always, i.e., there may be finite periods during which the activity is not enabled—then it will be executed infinitely often. An execution fragment is strongly fair with respect to activity α if it is not the case that α is infinitely often enabled without being taken beyond a certain point. Strong fairness is sometimes referred to as compassion.

Weak fairness means that if an activity, e.g., a transition in a process or an entire process itself, is continuously enabled—no periods are allowed in which the activity is not

enabled—then it has to be executed infinitely often. An execution fragment is weakly fair with respect to some activity, α say, if it is not the case that α is always enabled beyond some point without being taken beyond this point. Weak fairness is sometimes referred to as justice.

How to express these fairness constraints? There are different ways to formulate fairness requirements. In the sequel, we adopt the action-based view and define strong fairness for (sets of) actions. (In Chapter 5, also state-based notions of fairness will be introduced and the relationship between action-based and state-based fairness is studied in detail.) Let A be a set of actions. The execution fragment ρ is said to be strongly A-fair if the actions in A are not continuously ignored under the circumstance that they can be executed infinitely often. ρ is unconditionally A-fair if some action in A is infinitely often executed in ρ. Weak fairness is defined in a similar way as strong fairness (see below).

In order to formulate these fairness notions formally, the following auxiliary notion is convenient. For state s, let $Act(s)$ denote the set of actions that are executable in state s, that is,

$$Act(s) \;=\; \{\alpha \in Act \mid \exists s' \in S.\, s \xrightarrow{\alpha} s' \,\}.$$

Definition 3.43. Unconditional, Strong, and Weak Fairness

For transition system $TS = (S, Act, \rightarrow, I, AP, L)$ without terminal states, $A \subseteq Act$, and infinite execution fragment $\rho = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ of TS:

1. ρ is *unconditionally A-fair* whenever $\overset{\infty}{\exists}\, j.\ \alpha_j \in A$.
2. ρ is *strongly A-fair* whenever
$$\Big(\overset{\infty}{\exists}\, j.\ Act(s_j) \cap A \neq \varnothing\Big) \;\Longrightarrow\; \Big(\overset{\infty}{\exists}\, j.\ \alpha_j \in A\Big).$$
3. ρ is *weakly A-fair* whenever
$$\Big(\overset{\infty}{\forall}\, j.\ Act(s_j) \cap A \neq \varnothing\Big) \;\Longrightarrow\; \Big(\overset{\infty}{\exists}\, j.\ \alpha_j \in A\Big).$$

■

Here, $\overset{\infty}{\exists}\, j$ stands for "there are infinitely many j" and $\overset{\infty}{\forall}\, j$ for "for nearly all j" in the sense of "for all, except for finitely many j". The variable j, of course, ranges over the natural numbers.

To check whether a run is unconditionally A-fair it suffices to consider the actions that occur along the execution, i.e., it is not necessary to check which actions in A are enabled in visited states. However, in order to decide whether a given execution is strongly or weakly A-fair, it does not suffice to only consider the actions actually occurring in the execution. Instead, also the enabled actions in all visited states need to be considered. These enabled actions are possible in the visited states, but do not necessarily have to be taken along the considered execution.

Example 3.44. A Simple Shared-Variable Concurrent Program

Consider the following two processes that run in parallel and share an integer variable x that initially has value 0:

$$\begin{aligned} \textbf{proc}\ \mathsf{Inc} &= \textbf{while}\ \langle\, x \geqslant 0\ \textbf{do}\ x := x + 1 \,\rangle\ \textbf{od} \\ \textbf{proc}\ \mathsf{Reset} &= x := -1 \end{aligned}$$

The pair of brackets $\langle \ldots \rangle$ embraces an atomic section, i.e., process Inc performs the check whether x is positive and the increment of x (if the guard holds) as one atomic step. Does this parallel program terminate? When no fairness constraints are imposed, it is possible that process Inc is permanently executing, i.e., process Reset never gets its turn, and the assignment $x = -1$ is not executed. In this case, termination is thus not guaranteed, and the property is refuted. If, however, we require unconditional process fairness, then every process gets its turn, and termination is guaranteed. ■

An important question now is: given a verification problem, which fairness notion to use? Unfortunately, there is no clear answer to this question. Different forms of fairness do exist—the above is just a small, though important, fragment of all possible fairness notions—and there is no single favorite notion. For verification purposes, fairness constraints are crucial, though. Recall that the purpose of fairness constraints is to rule out certain "unreasonable" computations. If the fairness constraint is too strong, relevant computations may not be considered. In case a property is satisfied (for a transition system), it might well be the case that some reasonable computation that is not considered (as it is ruled out by the fairness constraint) refutes this property. On the other hand, if the fairness constraint is too weak, we may fail to prove a certain property as some unreasonable computations (that are not ruled out) refute it.

The relationship between the different fairness notions is as follows. Each unconditionally A-fair execution fragment is strongly A-fair, and each strongly A-fair execution fragment is weakly A-fair. In general, the reverse direction does not hold. For instance, an execution fragment that solely visits states in which no A-actions are possible is strongly A-fair (as the premise of strong A-fairness does not hold), but not unconditionally A-fair. Besides,

an execution fragment that only visits finitely many states in which some A-actions are enabled but never executes an A-action is weakly A-fair (as the premise of weak A-fairness does not hold), but not strongly A-fair. Summarizing, we have

$$\text{unconditional } A\text{-fairness} \implies \text{strong } A\text{-fairness} \implies \text{weak } A\text{-fairness}$$

where the reverse implication in general does not hold.

Example 3.45. Fair Execution Fragments

Consider the transition system TS_{Sem} for the semaphore-based mutual exclusion solution. We label the transitions with the actions req_i, $enter_i$ (for i=1, 2), and *rel* in the obvious way, see Figure 3.12.

In the execution fragment

$$\langle n_1, n_2, y=1\rangle \xrightarrow{req_1} \langle w_1, n_2, y=1\rangle \xrightarrow{enter_1} \langle c_1, n_2, y=0\rangle \xrightarrow{rel} \langle n_1, n_2, y=1\rangle \xrightarrow{req_1} \ldots$$

only the first process gets its turn. This execution fragment is indicated by the dashed arrows in Figure 3.12. It is *not* unconditionally fair for the set of actions

$$A = \{\, enter_2 \,\}.$$

It is, however, strongly A-fair, since no state is visited in which the action $enter_2$ is executable, and hence the premise of strong fairness is vacuously false. In the alternative execution fragment

$$\langle n_1, n_2, y=1\rangle \xrightarrow{req_2} \langle n_1, w_2, y=1\rangle \xrightarrow{req_1} \langle w_1, w_2, y=1\rangle \xrightarrow{enter_1}$$
$$\langle c_1, w_2, y=0\rangle \xrightarrow{rel} \langle n_1, w_2, y=1\rangle \xrightarrow{req_1} \ldots$$

the second process requests to enter its critical section but is ignored forever. This execution fragment is indicated by the dotted arrows in Figure 3.12. It is not strongly A-fair: although the action $enter_2$ is infinitely often enabled (viz. every time when visiting the state $\langle w_1, w_2, y = 1\rangle$ or $\langle n_1, w_2, y = 1\rangle$), it is never taken. It is, however, weakly A-fair, since the action $enter_2$ is not continuously enabled—it is not enabled in the state $\langle c_1, w_2, y = 0\rangle$. ■

A fairness constraint imposes a requirement on all actions in a set A. In order to enable different fairness constraints to be imposed on different, possibly nondisjoint, sets of actions, fairness *assumptions* are used. A fairness assumption for a transition system may require different notions of fairness with respect to several sets of actions.

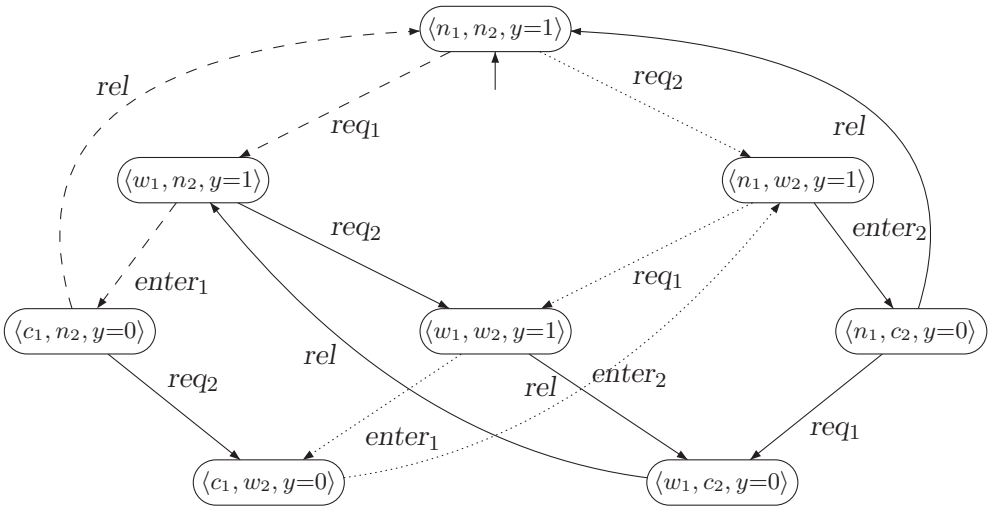


Figure 3.12: Two examples of fair execution fragments of the semaphore-based mutual exclusion algorithm.

Definition 3.46. Fairness Assumption

A *fairness assumption* for Act is a triple

$$\mathcal{F} \;=\; (\mathcal{F}_{ucond}, \mathcal{F}_{strong}, \mathcal{F}_{weak})$$

with $\mathcal{F}_{ucond}, \mathcal{F}_{strong}, \mathcal{F}_{weak} \subseteq 2^{Act}$. Execution ρ is $\mathcal{F}$-fair if

- it is unconditionally A-fair for all $A \in \mathcal{F}_{ucond}$,
- it is strongly A-fair for all $A \in \mathcal{F}_{strong}$, and
- it is weakly A-fair for all $A \in \mathcal{F}_{weak}$.

If the set $\mathcal{F}$ is clear from the context, we use the term fair instead of $\mathcal{F}$-fair. ■

Intuitively speaking, a fairness assumption is a triple of sets of (typically different) action sets, one such set of action sets is treated in a strongly fair manner, one in a weakly fair manner, and one in an unconditionally fair way. This is a rather general definition that allows imposing different fairness constraints on different sets of actions. Quite often, only a single type of fairness constraint suffices. In the sequel, we use the casual notations for these fairness assumptions. For $\mathcal{F} \subseteq 2^{Act}$, a strong fairness assumption denotes the fairness assumption $(\varnothing, \mathcal{F}, \varnothing)$. Weak, and unconditional fairness assumptions are used in a similar way.

The notion of $\mathcal{F}$-fairness as defined on execution fragments is lifted to traces and paths in the obvious way. An infinite trace σ is $\mathcal{F}$-fair if there is an $\mathcal{F}$-fair execution ρ with $\mathit{trace}(\rho) = \sigma$. $\mathcal{F}$-fair (infinite) path fragments and $\mathcal{F}$-fair paths are defined analogously.

Let $\mathit{FairPaths}_{\mathcal{F}}(s)$ denote the set of $\mathcal{F}$-paths of s (i.e., infinite $\mathcal{F}$-fair path fragments that start in state s), and $\mathit{FairPaths}_{\mathcal{F}}(TS)$ the set of $\mathcal{F}$-fair paths that start in some initial state of TS. Let $\mathit{FairTraces}_{\mathcal{F}}(s)$ denote the set of $\mathcal{F}$-fair traces of s, and $\mathit{FairTraces}_{\mathcal{F}}(TS)$ the set of $\mathcal{F}$-fair traces of the initial states of transition system TS:

$$\begin{aligned} \mathit{FairTraces}_{\mathcal{F}}(s) &= \mathit{trace}(\mathit{FairPaths}_{\mathcal{F}}(s)) \quad \text{and} \\ \mathit{FairTraces}_{\mathcal{F}}(TS) &= \bigcup_{s \in I} \mathit{FairTraces}_{\mathcal{F}}(s). \end{aligned}$$

Note that it does not make much sense to define these notions for finite traces as any finite trace is fair by default.

Example 3.47. Mutual Exclusion Again

Consider the following fairness requirement for two-process mutual exclusion algorithms:

"process P_i acquires access to its critical section infinitely often"

for any $i \in \{1, 2\}$. What kind of fairness assumption is appropriate to achieve this? Assume each process P_i has three states n_i (noncritical), w_i (waiting), and c_i (critical). As before, the actions req_i, enter_i, and rel are used to model the request to enter the critical section, the entering itself, and the release of the critical section. The strong-fairness assumption

$$\{\, \{\mathit{enter}_1, \mathit{enter}_2\} \,\}$$

ensures that one of the actions enter_1 or enter_2, is executed infinitely often. A behavior in which one of the processes gets access to the critical section infinitely often while the other gets access only finitely many times is strongly fair with respect to this assumption. This is, however, not intended. The strong-fairness assumption

$$\{\, \{\, \mathit{enter}_1 \,\}, \{\, \mathit{enter}_2 \,\} \,\}$$

indeed realizes the above requirement. This assumption should be viewed as a requirement on how to resolve the contention when both processes are awaiting to get access to the critical section. ■

Fairness assumptions can be *verifiable* properties whenever all infinite execution fragments are fair. For example, it can be verified that the transition system for Peterson's algorithm

satisfies the strong-fairness assumption

$$\mathcal{F}_{strong} \;=\; \{\,\{\,enter_1\,\},\{\,enter_2\,\}\,\}.$$

But in many cases it is necessary to *assume* the validity of the fairness conditions to verify liveness properties.

A transition system TS satisfies the LT property P under fairness assumption $\mathcal{F}$ if all $\mathcal{F}$-fair paths fulfill the property P. However, no requirements whatsoever are imposed on the unfair paths. This is formalized as follows.

Definition 3.48. Fair Satisfaction Relation for LT Properties

Let P be an LT property over AP and $\mathcal{F}$ a fairness assumption over Act. Transition system $TS = (S, Act, \rightarrow, I, AP, L)$ *fairly satisfies* P, notation $TS \models_{\mathcal{F}} P$, if and only if $FairTraces_{\mathcal{F}}(TS) \subseteq P$. ■

For a transition system that satisfies the fairness assumption $\mathcal{F}$ (i.e., *all* paths are $\mathcal{F}$-fair), the satisfaction relation $\models$ without fairness assumptions (see Definition 3.11, page 100) corresponds with the fair satisfaction relation $\models_{\mathcal{F}}$. In this case, the fairness assumption does not rule out any trace. However, in case a transition system has traces that are not $\mathcal{F}$-fair, then in general we are confronted with a situation

$$TS \;\models_{\mathcal{F}}\; P \quad \text{whereas} \quad TS \;\not\models\; P.$$

By restricting the validity of a property to the set of fair paths, the verification can be restricted to "realistic" executions.

Before turning to some examples, a few words on the relationship between unconditional, strong, and weak fairness are (again) in order. As indicated before, we have that the set of unconditional A-fair executions is a subset of all strong A-fair executions. In a similar way, the latter set of executions is a subset of all weak A-fair executions. Stated differently, unconditional fairness rules out more behaviors than strong fairness, and strong excludes more behaviors than weak fairness. For $\mathcal{F} = \{\,A_1, \ldots, A_k\,\}$, let fairness assumption $\mathcal{F}_{ucond} = (\mathcal{F}, \varnothing, \varnothing)$, $\mathcal{F}_{strong} = (\varnothing, \mathcal{F}, \varnothing)$, and $\mathcal{F}_{weak} = (\varnothing, \varnothing, \mathcal{F})$. Then for any transition system TS and LT property P it follows that:

$$TS \models_{\mathcal{F}_{weak}} P \;\Rightarrow\; TS \models_{\mathcal{F}_{strong}} P \;\Rightarrow\; TS \models_{\mathcal{F}_{ucond}} P.$$

Example 3.49. Independent Traffic Lights

Consider again the independent traffic lights. Let action *switch2green* denote the switching to green. Similarly *switch2red* denotes the switching to red. The fairness assumption

$$\mathcal{F} \;=\; \{\,\{\,switch2green_1, switch2red_1\,\}, \{\,switch2green_2, switch2red_2\,\}\,\}$$

expresses that both traffic lights infinitely often switch color. In this case, it is irrelevant whether strong, weak, or unconditional fairness is required.

Note that in this example $\mathcal{F}$ is *not* a verifiable system property (as it is not guaranteed to hold), but a natural property which is satisfied for a practical implementation of the system (with two independent processors). Obviously,

$$TrLight_1 \;|||\; TrLight_2 \;\models_{\mathcal{F}}\; \text{“each traffic light is green infinitely often”}$$

while the corresponding proposition for the nonfair relation $\models$ is refuted. ■

Example 3.50. Fairness for Mutual Exclusion Algorithms

Consider again the semaphore-based mutual exclusion algorithm, and assume the fairness assumption $\mathcal{F}$ consists of

$$\mathcal{F}_{weak} \;=\; \{\{\,req_1\,\}, \{\,req_2\,\}\} \quad \text{and} \quad \mathcal{F}_{strong} \;=\; \{\{\,enter_1\,\}, \{\,enter_2\,\}\}$$

and $\mathcal{F}_{ucond} = \varnothing$. The strong fairness constraint requires each process to enter its critical section infinitely often when it infinitely often gets the opportunity to do so. This does not forbid a process to never leave its noncritical section. To avoid this unrealistic scenario, the weak fairness constraint requires that any process infinitely often requests to enter the critical section. In order to do so, each process has to leave the noncritical section infinitely often. It follows that $TS_{Sem} \models_{\mathcal{F}} P$ where P stands for the property “every process enters its critical section infinitely often”.

Weak fairness is sufficient for request actions, as such actions are not critical: if req_i is executable in (global) state s, then it is executable in all direct successor states of s that are reached by an action that differs from req_i.

Peterson’s algorithm satisfies the strong fairness property

“Every process that requests access to the critical section
will eventually be able to do so”.

We can, however, not ensure that a process will ever leave its noncritical section and request the critical section. That is, the property P is refuted. This can be “repaired” by imposing the weak fairness constraint $\mathcal{F}_{weak} \;=\; \{\,\{\,req_1\,\}, \{\,req_2\,\}\,\}$. We now have $TS_{Pet} \models_{\mathcal{F}_{weak}} P$. ■

3.5.2 Fairness Strategies

The examples in the previous section indicate that fairness assumptions may be necessary to verify liveness properties of transition system TS. In order to rule out the "unrealistic" computations, fairness assumptions are imposed on the traces of TS, and it is checked whether $TS \models_{\mathcal{F}} P$ as opposed to checking $TS \models P$ (without fairness). Which fairness assumptions are appropriate to check P? Many model-checking tools provide the possibility to work with built-in fairness assumptions. Roughly speaking, the intention is to rule out executions that cannot occur in a realistic implementation. But what does that exactly mean? In order to give some insight into this, we consider several fairness assumptions for synchronizing concurrent systems. The aim is to establish a fair communication mechanism between the various processes involved. A rule of thumb is: Strong fairness is needed to obtain an adequate resolution of contentions (between processes), while weak fairness suffices for sets of actions that represent the concurrent execution of independent actions (i.e., interleaving).

For modeling *asynchronous* concurrency by means of transition systems, the following rule of thumb can be adopted:

concurrency = interleaving (i.e., nondeterminism) + fairness

Example 3.51. Fair Concurrency with Synchronization

Consider the concurrent transition system:

$$TS = TS_1 \parallel TS_2 \parallel \ldots \parallel TS_n \quad ,$$

where $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP_i, L_i)$, for $1 \leqslant i \leqslant n$, is a transition system without terminal states. Recall that each pair of processes TS_i and TS_j (for $i \neq j$) has to synchronize on their common sets of actions, i.e., $Syn_{i,j} = Act_i \cap Act_j$. It is assumed that $Syn_{i,j} \cap Act_k = \varnothing$ for any $k \neq i, j$. For simplicity, it is assumed that TS has no terminal states. (In case there are terminal states, each finite execution is considered to be fair.)

We consider several fairness assumptions on the transition system TS. First, consider the strong fairness assumption

$$\{Act_1, Act_2, \ldots, Act_n\}$$

which ensures that each transition system TS_i executes an action infinitely often, provided the composite system TS is infinitely often in a (global) state with a transition being executable in which TS_i participates. This fairness assumption, however, cannot ensure that a communication will ever occur—it is possible for each TS_i to only execute local actions ad infinitum.

In order to force a synchronization to take place every now and then, the strong fairness assumption

$$\{ \{ \alpha \} \mid \alpha \in Syn_{i,j}, 0 < i < j \leqslant n \} \tag{3.1}$$

could be imposed. It forces every synchronization action to happen infinitely often. Alternatively, a somewhat weaker fairness assumption can be imposed by requiring every pair of processes to synchronize—regardless of the synchronization action—infinitely often. The corresponding strong fairness assumption is

$$\{ Syn_{i,j} \mid 0 < i < j \leqslant n \}. \tag{3.2}$$

Whereas (3.2) allows processes to always synchronize on the same action, (3.1) does not permit this. The strong fairness assumption:

$$\{ \bigcup_{0<i<j\leqslant n} Syn_{i,j} \}$$

goes even one step further as it only requires a synchronization to take place infinitely often, regardless of the process involved. This fairness assumption does not rule out executions in which always the same synchronization takes place or in which always the same pair of processes synchronizes.

Note that all fairness assumptions in this example so far are strong. This requires that infinitely often a synchronization is enabled. As the constituting transition systems TS_i may execute internal actions, synchronizations are not continuously enabled, and hence weak fairness is in general inappropriate.

If the internal actions should be fairly considered, too, then we may use, e.g., the strong fairness assumption

$$\{ Act_1 \setminus Syn_1, \ldots, Act_n \setminus Syn_n \} \cup \{ \{ \alpha \} \mid \alpha \in Syn \},$$

where $Syn_i = \bigcup_{j\neq i} Syn_{i,j}$ denotes the set of all synchronization actions of TS_i and $Syn = \bigcup_i Syn_i$.

Under the assumption that in every (local) state either only internal actions or only synchronization actions are executable, it suffices to impose the *weak* fairness constraint

$$\{ Act_1 \setminus Syn_1, \ldots, Act_n \setminus Syn_n \}.$$

Weak fairness is appropriate for the internal actions $\alpha \in Act_i \setminus Syn_i$, as the ability to perform an internal action is preserved until it will be executed. ■

As an example of another form of fairness we consider the following sequential hardware circuit.

Example 3.52. Circuit Fairness

For sequential circuits we have modeled the environmental behavior, which provides the input bits, by means of nondeterminism. It may be necessary to impose fairness assumptions on the environment in order to be able to verify liveness properties, such as "the values 0 and 1 are output infinitely often". Let us illustrate this by means of a concrete example. Consider a sequential circuit with input variable x, output variable y, and register r. Let the transition function and the output function be defined as

$$\lambda_y \;=\; \delta_r \;=\; x \leftrightarrow \neg r.$$

That is, the circuit inverts the register and output evaluation if and only if the input bit is set. If x=0, then the register evaluation remains unchanged. The value of the register is output. Suppose all transitions leading to a state with a register evaluation of the form $[r = 1, \ldots]$ are labeled with the action *set*. Imposing the unconditional fairness assumption $\{\{\, set \,\}\}$ ensures that the values 0 and 1 are output infinitely often. ■

3.5.3 Fairness and Safety

While fairness assumptions may be necessary to verify liveness properties, they are irrelevant for verifying safety properties, provided that they can always be ensured by means of an appropriate scheduling strategy. Such fairness assumptions are called *realizable* fairness assumptions. A fairness assumption cannot be realized in a transition system whenever there exists a reachable state from where *no* fair path begins. In this case, it is impossible to design a scheduler that resolves the nondeterminism such that only fair paths remain.

Example 3.53. A Nonrealizable Fairness Assumption

Consider the transition system depicted in Figure 3.13, and suppose the unconditional fairness assumption $\{\{\, \alpha \,\}\}$ is imposed. As the α-transition can only be taken once, it is evident that the transition system can never guarantee this form of fairness. As there is a reachable state from which no unconditional fair path exists, this fairness assumption is nonrealizable. ■

Definition 3.54. Realizable Fairness Assumption

Let TS be a transition system with the set of actions Act and $\mathcal{F}$ a fairness assumption for Act. $\mathcal{F}$ is called *realizable* for TS if for every reachable state s: $\mathit{FairPaths}_{\mathcal{F}}(s) \neq \varnothing$. ■

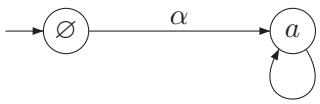


Figure 3.13: Unconditional fairness.

Stated in words, a fairness assumption is realizable in a transition system TS whenever in any reachable state at least one fair execution is possible. This entails that every initial finite execution fragment of TS can be completed to a fair execution. Note that there is no requirement on the unreachable states.

The following theorem shows the irrelevance of realizable fairness assumptions for the verification of safety properties. The *suffix property* of fairness assumptions is essential for its proof. This means the following. If

$$\rho \;=\; s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots$$

is an (infinite) execution fragment, then ρ is fair if and only if every suffix

$$s_j \xrightarrow{\alpha_{j+1}} s_{j+1} \xrightarrow{\alpha_{j+2}} s_{j+2} \xrightarrow{\alpha_{j+3}} \ldots$$

of ρ is fair too. Conversely, every fair execution fragment ρ (as above) starting in state s_0 can be preceded by an arbitrary finite execution fragment

$$s'_0 \xrightarrow{\beta_1} s'_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_n} s'_n = s_0$$

ending in s_0. Proceeding in s_0 by execution ρ yields the fair execution fragment:

$$\underbrace{s'_0 \xrightarrow{\beta_1} s'_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_n} s'_n}_{\text{arbitrary starting fragment}} = \underbrace{s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots}_{\text{fair continuation}}$$

Theorem 3.55. Realizable Fairness is Irrelevant for Safety Properties

Let TS be a transition system with set of propositions AP, $\mathcal{F}$ a realizable fairness assumption for TS, and P_{safe} a safety property over AP. Then:

$$TS \models P_{safe} \quad \text{if and only if} \quad TS \models_{\mathcal{F}} P_{safe}.$$

Proof: "⇒": Assume $TS \models P_{safe}$. Then, by definition of $\models$ and the fact that the fair traces of TS are a subset of the traces of TS, we have

$$\mathit{FairTraces}_{\mathcal{F}}(TS) \;\subseteq\; \mathit{Traces}(TS) \;\subseteq\; P_{safe}.$$

Thus, by definition of $\models_{\mathcal{F}}$ it follows that $TS \models_{\mathcal{F}} P_{safe}$.

"$\Leftarrow$": Assume $TS \models_{\mathcal{F}} P_{safe}$. It is to be shown $TS \models P_{safe}$, i.e., $\mathit{Traces}(TS) \subseteq P_{safe}$. This is done by contraposition. Let $\sigma \in \mathit{Traces}(TS)$ and assume $\sigma \notin P_{safe}$. As $\sigma \notin P_{safe}$, there is a bad prefix of σ, $\widehat{\sigma}$ say, for P_{safe}. Hence, the set of properties that has $\widehat{\sigma}$ as a prefix, i.e.,

$$P \;=\; \Big\{\sigma' \in \Big(2^{AP}\Big)^{\omega} \;\mid\; \widehat{\sigma} \in \mathit{pref}(\sigma')\Big\},$$

satisfies $P \cap P_{safe} = \varnothing$. Further, let $\widehat{\pi} = s_0\, s_1 \ldots s_n$ be a finite path fragment of TS with

$$\mathit{trace}(\widehat{\pi}) \;=\; \widehat{\sigma}.$$

Since $\mathcal{F}$ is a realizable fairness assumption for TS and $s_n \in \mathit{Reach}(TS)$, there is an $\mathcal{F}$-fair path starting in s_n. Let

$$s_n\, s_{n+1}\, s_{n+2} \ldots \in \mathit{FairPaths}_{\mathcal{F}}(s_n).$$

The path $\pi = s_0 \ldots s_n\, s_{n+1}\, s_{n+2} \ldots$ is in $\mathit{FairPaths}_{\mathcal{F}}(TS)$ and thus,

$$\mathit{trace}(\pi) \;=\; L(s_0) \ldots L(s_n)\, L(s_{n+1})\, L(s_{n+2}) \ldots \;\in\; \mathit{FairTraces}_{\mathcal{F}}(TS) \;\subseteq\; P_{safe}.$$

On the other hand, $\widehat{\sigma} = L(s_0) \ldots L(s_n)$ is a prefix of $\mathit{trace}(\pi)$. Thus, $\mathit{trace}(\pi) \in P$. This contradicts $P \cap P_{safe} = \varnothing$. ■

Theorem 3.55 does not hold if arbitrary (i.e., possibly nonrealizable) fairness assumptions are permitted. This is illustrated by the following example.

Example 3.56. Nonrealizable Fairness may harm Safety Properties

Consider the transition system TS in Figure 3.14 and suppose the unconditional fairness assumption $\mathcal{F} = \{\{\alpha\}\}$ is imposed. $\mathcal{F}$ is not realizable for TS, as the noninitial state (referred to as state s_1), is reachable, but has no $\mathcal{F}$-fair execution. Obviously, TS has only one fair path (namely the path that never leaves the initial state s_0). In contrast, paths of the form $s_0 \ldots s_0\, s_1\, s_1\, s_1 \ldots$ are not fair, since α is only executed finitely often. Accordingly, we have that

$$TS \models_{\mathcal{F}} \text{"never } a\text{"} \quad \text{but} \quad TS \not\models \text{"never } a\text{"}.$$

■

3.6 Summary

- The set of reachable states of a transition system TS can be determined by a search algorithm on the state graph of TS.

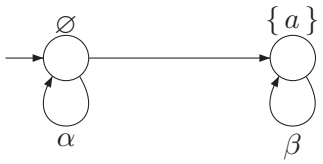


Figure 3.14: Unconditional fairness may be relevant for safety properties.

- A trace is a sequence of sets (!) of atomic propositions. The traces of a transition system TS are obtained from projecting the paths to the sequence of state labels.
- A linear-time (LT, for short) property is a set of infinite words over the alphabet 2^{AP}.
- Two transition systems are trace-equivalent (i.e., they exhibit the same traces) if and only if they satisfy the same LT properties.
- An invariant is an LT property that is purely state-based and requires a propositional logic formula Φ to hold for all reachable states. Invariants can be checked using a depth-first search where the depth-first search stack can be used to provide a counterexample in case an invariant is refuted.
- Safety properties are generalizations of invariants. They constrain the finite behaviors. The formal definition of safety properties can be provided by means of their bad prefixes in the sense that each trace that refutes a safety property has a finite prefix, the bad prefix, that causes this.
- Two transition systems exhibit the same finite traces if and only if they satisfy the same safety properties.
- A liveness property is an LT property if it does not rule out any finite behavior. It constrains infinite behavior.
- Any LT property is equivalent to an LT property that is a conjunction of a safety and a liveness property.
- Fairness assumptions serve to rule out traces that are considered to be unrealistic. They consist of unconditional, strong, and weak fairness constraints, i.e., constraints on the actions that occur along infinite executions.
- Fairness assumptions are often necessary to establish liveness properties, but they are—provided they are realizable—irrelevant for safety properties.

3.7 Bibliographic Notes

The dining philosophers example discussed in Example 3.2 has been developed by Dijkstra [128] in the early seventies to illustrate the intricacies of concurrency. Since then it has become one of the standard examples for reasoning about parallel systems.

The depth-first search algorithm that we used as a basis for the invariance checking algorithm goes back to Tarjan [387]. Further details about graph traversal algorithms can be found in any textbook on algorithms and data structures, e.g. [100], or on graph algorithms [188].

Traces. Traces have been introduced by Hoare [202] to describe the linear-time behavior of transition systems and have been used as the initial semantical model for the process algebra CSP. Trace theory has further been developed by, among others, van de Snepscheut [403] and Rem [354] and has successfully been used to design and analyze fine-grained parallel programs that occur in, e.g., asynchronous hardware circuits. Several extensions to traces and their induced equivalences have been proposed, such as failures [65] where a trace is equipped with information about which actions are rejected after execution of such trace. The FDR model checker [356] supports the automated checking of failure-divergence refinement and the checking of safety properties. A comprehensive survey of these refined notions of trace equivalence and trace inclusion has recently been given by Bruda [68].

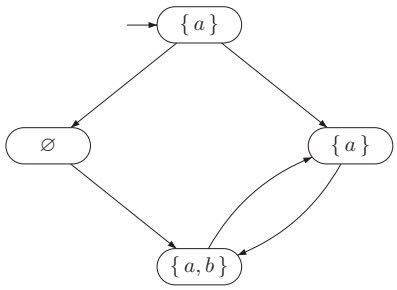
Safety and liveness. The specification of linear-time properties using sets of infinite sequences of states (and their topological characterizations) goes back to Alpern and Schneider [5, 6, 7]. An earlier approach by Gerth [164] was based on finite sequences. Lamport [257] categorized properties as either safety, liveness, or properties that are neither. Alternative characterizations have been provided by Rem [355] and Gumm [178]. Subclasses of liveness and safety properties in the linear-time framework have been identified by Sistla [371], and Chang, Manna, and Pnueli [80]. Other definitions of liveness properties have been provided by Dederichs and Weber [119] and Naumovich and Clarke [312] (for linear-time properties), and Manolios and Trefler [285, 286] (for branching-time properties). A survey of safety and liveness has been given by Kindler [239].

Fairness. Fairness has implicitly been introduced by Dijkstra [126, 127] by assuming that one should abstract from the speed of processors and that each process gets its turn once it is initiated. Park [321] studied the notion of fairness in providing a semantics to data-flow languages. Weak and strong fairness have been introduced by Lehmann, Pnueli, and Stavi [267] in the context of shared variable concurrent programs. Queille and Sifakis [348] consider fairness for transition systems. An overview of the fairness notions has been provided by Kwiatkowska [252]. An extensive treatment of fairness can be found

in the monograph by Francez [155]. A recent characterization of fairness in terms of topology, language theory, and game theory has been provided by Völzer, Varacca, and Kindler [415].

3.8 Exercises

Exercise 3.1. Give the traces on the set of atomic propositions $\{a, b\}$ of the following transition system:



Exercise 3.2. On page 97, a transformation is described of a transition system TS with possible terminal states into an "equivalent" transition system TS^* without terminal states. Questions:

(a) Give a formal definition of this transformation $TS \mapsto TS^*$

(b) Prove that the transformation preserves trace-equivalence, i.e., show that if TS_1, TS_2 are transition systems (possibly with terminal states) such that $\mathit{Traces}(TS_1) = \mathit{Traces}(TS_2)$, then $\mathit{Traces}(TS_1^*) = \mathit{Traces}(TS_2^*)$.[8]

Exercise 3.3. Give an algorithm (in pseudocode) for invariant checking such that in case the invariant is refuted, a *minimal* counterexample, i.e., a counterexample of minimal length, is provided as an error indication.

Exercise 3.4. Recall the definition of AP-deterministic transition systems (Definition 2.5 on page 24). Let TS and TS' be transition systems with the same set of atomic propositions AP. Prove the following relationship between trace inclusion and finite trace inclusion:

(a) For AP-deterministic TS and TS':

$$\mathit{Traces}(TS) = \mathit{Traces}(TS') \text{ if and only if } \mathit{Traces}_{\mathit{fin}}(TS) = \mathit{Traces}_{\mathit{fin}}(TS').$$

[8] If TS is a transition system with terminal states, then $\mathit{Traces}(TS)$ is defined as the set of all words $\mathit{trace}(\pi)$ where π is an initial, maximal path fragment in TS.

(b) Give concrete examples of TS and TS' where at least one of the transition systems is not AP-deterministic, but

$$\mathit{Traces}(TS) \not\subseteq \mathit{Traces}(TS') \quad \text{and} \quad \mathit{Traces}_{\mathit{fin}}(TS) = \mathit{Traces}_{\mathit{fin}}(TS').$$

Exercise 3.5. Consider the set AP of atomic propositions defined by $AP = \{ x = 0, x > 1 \}$ and consider a nonterminating sequential computer program P that manipulates the variable x. Formulate the following informally stated properties as LT properties:

(a) false

(b) initially x is equal to zero

(c) initially x differs from zero

(d) initially x is equal to zero, but at some point x exceeds one

(e) x exceeds one only finitely many times

(f) x exceeds one infinitely often

(g) the value of x alternates between zero and two

(h) true

(This exercise has been adopted from [355].) Determine which of the provided LT properties are safety properties. Justify your answers.

Exercise 3.6. Consider the set $AP = \{ A, B \}$ of atomic propositions. Formulate the following properties as LT properties and characterize each of them as being either an invariance, safety property, or liveness property, or none of these.

(a) A should never occur,

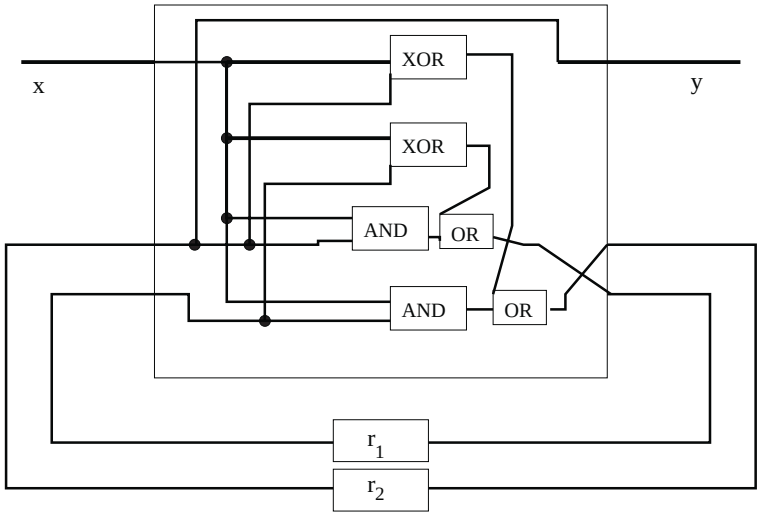
(b) A should occur exactly once,

(c) A and B alternate infinitely often,

(d) A should eventually be followed by B.

(This exercise has been inspired by [312].)

Exercise 3.7. Consider the following sequential hardware circuit:



The circuit has input variable x, output variable y, and registers r_1 and r_2 with initial values $r_1 = 0$ and $r_2 = 1$. The set AP of atomic propositions equals $\{ x, r_1, r_2, y \}$. Besides, consider the following informally formulated LT properties over AP:

P_1 : Whenever the input x is continuously high (i.e., x=1), then the output y is infinitely often high.

P_2 : Whenever currently r_2=0, then it will never be the case that after the next input, r_1=1.

P_3 : It is never the case that two successive outputs are high.

P_4 : The configuration with x=1 and r_1=0 never occurs.

Questions:

(a) Give for each of these properties an example of an infinite word that belongs to P_i. Do the same for the property $\left(2^{AP}\right)^{\omega} \setminus P_i$, i.e., the complement of P_i.

(b) Determine which properties are satisfied by the hardware circuit that is given above.

(c) Determine which of the properties are safety properties. Indicate which properties are invariants.

  (i) For each safety property P_i, determine the (regular) language of bad prefixes.

  (ii) For each invariant, provide the propositional logic formula that specifies the property that should be fulfilled by each state.

Exercise 3.8. Let LT properties P and P' be equivalent, notation $P \cong P'$, if and only if $\mathit{pref}(P) = \mathit{pref}(P')$. Prove or disprove: $P \cong P'$ if and only if $\mathit{closure}(P) = \mathit{closure}(P')$.

Exercise 3.9. Show that for any transition system TS, the set $\mathit{closure}(\mathit{Traces}(TS))$ is a safety property such that $TS \models \mathit{closure}(\mathit{Traces}(TS))$.

Exercise 3.10. Let P be an LT property. Prove: $\mathit{pref}(\mathit{closure}(P)) = \mathit{pref}(P)$.

Exercise 3.11. Let P and P' be liveness properties over AP. Prove or disprove the following claims:

(a) $P \cup P'$ is a liveness property,

(b) $P \cap P'$ is a liveness property.

Answer the same question for P and P' being safety properties.

Exercise 3.12. Prove Lemma 3.38 on page 125.

Exercise 3.13. Let $AP = \{ a, b \}$ and let P be the LT property of all infinite words $\sigma = A_0 A_1 A_2 \ldots \in \left(2^{AP}\right)^{\omega}$ such that there exists $n \geqslant 0$ with $a \in A_i$ for $0 \leqslant i < n$, $\{ a, b \} = A_n$ and $b \in A_j$ for infinitely many $j \geqslant 0$. Provide a decomposition $P = P_{safe} \cap P_{live}$ into a safety and a liveness property.

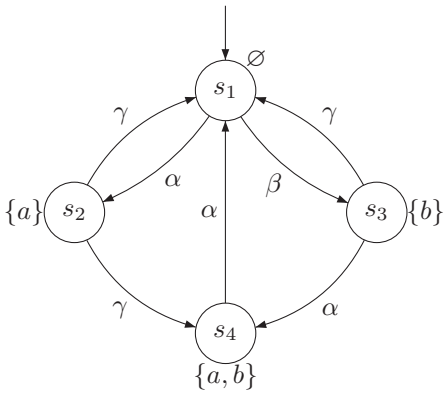
Exercise 3.14. Let TS_{Sem} and TS_{Pet} be the transition systems for the semaphore-based mutual exclusion algorithm (Example 2.24 on page 43) and Peterson's algorithm (Example 2.25 on page 45), respectively. Let $AP = \{ \mathit{wait}_i, \mathit{crit}_i \mid i = 1, 2 \}$. Prove or disprove:

$$\mathit{Traces}(TS_{Sem}) = \mathit{Traces}(TS_{Pet}).$$

If the property does not hold, provide an example trace of one transition system that is not a trace of the other one.

Exercise 3.15. Consider the transition system TS outlined on the right and the sets of actions $B_1 = \{ \alpha \}$, $B_2 = \{ \alpha, \beta \}$, and $B_3 = \{ \beta \}$. Further, let E_b, E_a and E' be the following LT properties:

- E_b = the set of all words $A_0A_1\cdots \in \left(2^{\{a,b\}}\right)^\omega$ with $A_i \in \{\{a,b\},\{b\}\}$ for infinitely many i (i.e., infinitely often b).
- E_a = the set of all words $A_0A_1\cdots \in \left(2^{\{a,b\}}\right)^\omega$ with $A_i \in \{\{a,b\},\{a\}\}$ for infinitely many i (i.e., infinitely often a).
- E' = set of all words $A_0A_1\cdots \in \left(2^{\{a,b\}}\right)^\omega$ for which there does not exist an $i \in \mathbb{N}$ s.t. $A_i = \{a\}$, $A_{i+1} = \{a,b\}$ and $A_{i+2} = \varnothing$.



Questions:

(a) For which sets of actions B_i ($i \in \{1,2,3\}$) and LT properties $E \in \{E_a, E_b, E'\}$ it holds that $TS \models_{\mathcal{F}_i} E$? Here, $\mathcal{F}_i$ is a strong fairness condition with respect to B_i that does not impose any unconditional or weak fairness conditions (i.e., $\mathcal{F}_i = (\varnothing, \{B_i\}, \varnothing)$).

(b) Answer the same question in the case of weak fairness (instead of strong fairness, i.e., $\mathcal{F}_i = (\varnothing, \varnothing, \{B_i\})$).

Exercise 3.16. Let TS_i (for i=1, 2) be the transition system $(S_i, Act, \rightarrow_i, I_i, AP_i, L_i)$ and $\mathcal{F} = (\mathcal{F}_{ucond}, \mathcal{F}_{strong}, \mathcal{F}_{weak})$ be a fairness assumption with $\mathcal{F}_{ucond} = \varnothing$. Prove or disprove (i.e., give a counterexample for) the following claims:

(a) $Traces(TS_1) \subseteq Traces(TS_1 \parallel TS_2)$ where $Syn \subseteq Act$

(b) $Traces(TS_1) \subseteq Traces(TS_1 \,|||\, TS_2)$

(c) $Traces(TS_1 \parallel TS_2) \subseteq Traces(TS_1)$ where $Syn \subseteq Act$

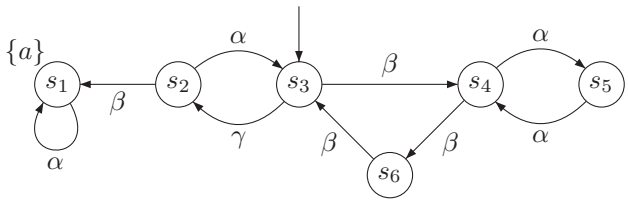
(d) $Traces(TS_1) \subseteq Traces(TS_2) \;\Rightarrow\; FairTraces_{\mathcal{F}}(TS_1) \subseteq FairTraces_{\mathcal{F}}(TS_2)$

(e) For liveness property P with $TS_2 \models_{\mathcal{F}} P$ we have

$$Traces(TS_1) \subseteq Traces(TS_2) \quad\Rightarrow\quad TS_1 \models_{\mathcal{F}} P.$$

Assume that in items (a) through (c), we have $AP_2 = \varnothing$ and that $TS_1 \parallel TS_2$ and $TS_1 \,|||\, TS_2$, respectively, have $AP = AP_1$ as atomic propositions and $L(\langle s, s'\rangle) = L_1(s)$ as labeling function. In items (d) and (e) you may assume that $AP_1 = AP_2$.

Exercise 3.17. Consider the following transition system TS with the set of atomic propositions $\{a\}$:

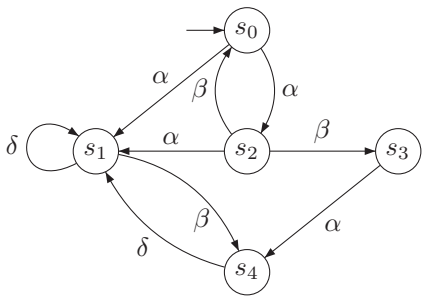


Let the fairness assumption

$$\mathcal{F} = (\varnothing, \{\{\alpha\}, \{\beta\}\}, \{\{\beta\}\}).$$

Determine whether $TS \models_{\mathcal{F}}$ "eventually a". Justify your answer!

Exercise 3.18. Consider the following transition system TS (without atomic propositions):



Decide which of the following fairness assumptions $\mathcal{F}_i$ are realizable for TS. Justify your answers!

(a) $\mathcal{F}_1 = (\{\{\alpha\}\}, \{\{\delta\}\}, \{\{\alpha, \beta\}\})$

(b) $\mathcal{F}_2 = (\{\{\delta, \alpha\}\}, \{\{\alpha, \beta\}\}, \{\{\delta\}\})$

(c) $\mathcal{F}_3 = (\{\{\alpha, \delta\}, \{\beta\}\}, \{\{\alpha, \beta\}\}, \{\{\delta\}\})$

Exercise 3.19. Let $AP = \{a, b\}$.

(a) P_1 denotes the LT property that consists of all infinite words $\sigma = A_0 A_1 A_2 \ldots \in \left(2^{AP}\right)^\omega$ such that there exists $n \geqslant 0$ with

$$\forall j < n.\ A_j = \varnothing \quad \wedge \quad A_n = \{a\} \quad \wedge \quad \forall k > n.\ (A_k = \{a\} \Rightarrow A_{k+1} = \{b\}).$$

(i) Give an ω–regular expression for P_1.

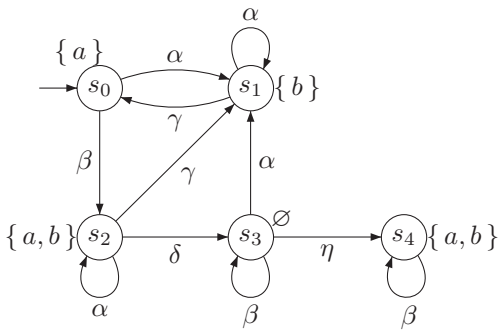
(ii) Apply the decomposition theorem and give expressions for P_{safe} and P_{live}.

(iii) Justify that P_{live} is a liveness and that P_{safe} is a safety property.

(b) Let P_2 denote the set of traces of the form $\sigma = A_0 A_1 A_2 \ldots \in \big(2^{AP}\big)^\omega$ such that

$$\overset{\infty}{\exists}\, k.\ A_k = \{\, a, b \,\} \quad \wedge \quad \exists n \geqslant 0.\ \forall k > n.\ \big(a \in A_k \Rightarrow b \in A_{k+1}\big).$$

Consider the following transition system TS:



Consider the following fairness assumptions:

(a) $\mathcal{F}_1 = \Big(\big\{\{\alpha\}\big\}, \big\{\{\beta\}, \{\delta, \gamma\}, \{\eta\}\big\}, \varnothing\Big)$. Decide whether $TS \models_{\mathcal{F}_1} P_2$.

(b) $\mathcal{F}_2 = \Big(\big\{\{\alpha\}\big\}, \big\{\{\beta\}, \{\gamma\}\big\}, \big\{\{\eta\}\big\}\Big)$. Decide whether $TS \models_{\mathcal{F}_2} P_2$.

Justify your answers.

Exercise 3.20. Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system without terminal states and let $A_1, \ldots, A_k,\ A'_1, \ldots, A'_l \subseteq Act$.

(a) Let $\mathcal{F}$ be the fairness assumption $\mathcal{F} = (\varnothing, \mathcal{F}_{strong}, \mathcal{F}_{weak})$ where

$$\mathcal{F}_{strong} = \{A_1, \ldots, A_k\} \text{ and } \mathcal{F}_{weak} = \{A'_1, \ldots, A'_l\}.$$

Provide a sketch of a scheduling algorithm that resolves the nondeterminism in TS in an $\mathcal{F}$-fair way.

(b) Let $\mathcal{F}_{ucond} = \{A_1, \ldots, A_k\}$, viewed as an unconditional fairness assumption for TS. Design a (scheduling) algorithm that checks whether $\mathcal{F}_{ucond}$ for TS is realizable, and if so, generates an $\mathcal{F}_{ucond}$-fair execution for TS.

Chapter 4

Regular Properties

This chapter treats some elementary algorithms to verify important classes of safety properties, liveness properties, and a wide range of other linear-time properties. We first consider regular safety properties, i.e., safety properties whose bad prefixes constitute a regular language, and hence can be recognized by a finite automaton. The algorithm to check a safety property P_{safe} for a given finite transition system TS relies on a reduction to the invariant-checking problem in a certain product construction of TS with a finite automaton that recognizes the bad prefixes of P_{safe}.

We then generalize this automaton-based verification algorithm to a larger class of linear-time properties, the so-called ω-regular properties. This class of properties covers regular safety properties, but also many other relevant properties such as various liveness properties. ω-regular properties can be represented by so-called Büchi automata, a variant of finite automata that accept infinite (rather than finite) words. Büchi automata will be the key concept to verify ω-regular properties via a reduction to persistence checking. The latter is a variant of invariant checking and aims to show that a certain state-condition holds continuously from some moment on.

4.1 Automata on Finite Words

Definition 4.1. Nondeterministic Finite Automaton (NFA)

A *nondeterministic finite automaton* (NFA) $\mathcal{A}$ is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ where

- Q is a finite set of states,
- Σ is an alphabet,
- $\delta : Q \times \Sigma \to 2^Q$ is a transition function,
- $Q_0 \subseteq Q$ is a set of initial states, and
- $F \subseteq Q$ is a set of *accept* (or: final) states.

The size of $\mathcal{A}$, denoted $|\mathcal{A}|$, is the number of states and transitions in $\mathcal{A}$, i.e.,

$$|\mathcal{A}| \;=\; |Q| + \sum_{q \in Q} \sum_{A \in \Sigma} |\delta(q, A)|.$$

■

Σ defines the symbols on which the automaton is defined. The (possibly empty) set Q_0 defines the states in which the automaton may start. The transition function δ can be identified with the relation $\to \subseteq Q \times \Sigma \times Q$ given by

$$q \xrightarrow{A} q' \text{ iff } q' \in \delta(q, A).$$

Thus, often the notion of transition relation (rather than transition function) is used for δ. Intuitively, $q \xrightarrow{A} q'$ denotes that the automaton can move from state q to state q' when reading the input symbol A.

Example 4.2. An Example of a Finite-State Automaton

An example of an NFA is depicted in Figure 4.1. Here, $Q = \{ q_0, q_1, q_2 \}$, $\Sigma = \{ A, B \}$, $Q_0 = \{ q_0 \}$, $F = \{ q_2 \}$, and the transition function δ is defined by

$$\begin{array}{lcllcl} \delta(q_0, A) & = & \{ q_0 \} & \delta(q_0, B) & = & \{ q_0, q_1 \} \\ \delta(q_1, A) & = & \{ q_2 \} & \delta(q_1, B) & = & \{ q_2 \} \\ \delta(q_2, A) & = & \varnothing & \delta(q_2, B) & = & \varnothing \end{array}$$

This corresponds to the transitions $q_0 \xrightarrow{A} q_0$, $q_0 \xrightarrow{B} q_0$, $q_0 \xrightarrow{B} q_1$, $q_1 \xrightarrow{A} q_2$, and $q_1 \xrightarrow{B} q_2$. The drawing conventions for an NFA are the same as for labeled transition systems. Accept states are distinguished from other states by drawing them with a double circle. ■

The intuitive operational behavior of an NFA is as follows. The automaton starts in one of the states in Q_0, and then is fed with an input word $w \in \Sigma^*$. The automaton reads this

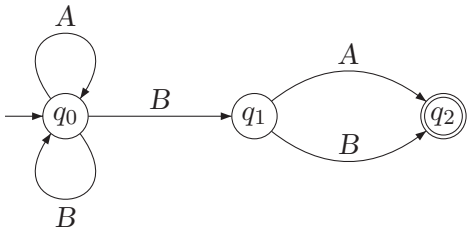


Figure 4.1: An example of a finite-state automaton.

word character by character from the left to the right. (The reader may assume that the input word is provided on a tape from which the automaton reads the input symbols from the left to the right by moving a cursor from the current position i to the next position $i{+}1$ when reading the ith input symbol. However, the automaton can neither write on the tape nor move the cursor in another way than one position to the right.) After reading an input symbol, the automaton changes its state according to the transition relation δ. That is, if the current input symbol A is read in the state q, the automaton chooses one of the possible transitions $q \xrightarrow{A} q'$ (i.e., one state $q' \in \delta(q, A)$) and moves to q' where the next input symbol will be consumed. If $\delta(q, A)$ contains two or more states, then the decision for the next state is made nondeterministically. An NFA cannot perform any transition when its current state q does not have an outgoing transition that is labeled with the current input symbol A. In that case, i.e., if $\delta(q, A) = \varnothing$, the automaton is stuck and no further progress can be made. The input word is said to be *rejected*. When the complete input word has been read, the automaton halts. It *accepts* whenever the current state is an accept state, and it *rejects* otherwise.

This intuitive explanation of the possible behaviors of an NFA for a given input word $w = A_1 \dots A_n$ is formalized by means of runs for w (see Definition 4.3 below). For any input word w there might be several possible behaviors (runs); some of them might be accepting, some of them might be rejecting. Word w is accepted by $\mathcal{A}$ if *at least one* of its runs is accepting, i.e., succeeds in reading the whole word and ends in a final state. This relies on the typical nondeterministic acceptor criterion which assumes an oracle for resolving the nondeterminism such that, whenever possible, an accepting run will be generated.

Definition 4.3. Runs, Accepted Language of an NFA

Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be an NFA and $w = A_1 \dots A_n \in \Sigma^*$ a finite word. A *run* for w in $\mathcal{A}$ is a finite sequence of states $q_0\, q_1\, \dots\, q_n$ such that

- $q_0 \in Q_0$ and

- $q_i \xrightarrow{A_{i+1}} q_{i+1}$ for all $0 \leqslant i < n$.

Run $q_0\, q_1 \ldots q_n$ is called *accepting* if $q_n \in F$. A finite word $w \in \Sigma^*$ is called *accepted* by $\mathcal{A}$ if there exists an accepting run for w. The *accepted language* of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of finite words in Σ^* accepted by $\mathcal{A}$, i.e.,

$$\mathcal{L}(\mathcal{A}) = \{\, w \in \Sigma^* \mid \text{ there exists an accepting run for } w \text{ in } \mathcal{A} \,\}.$$

■

Example 4.4. Runs and Accepted Words

Example runs of the automaton $\mathcal{A}$ in Figure 4.1 are q_0 for the empty word ε, $q_0\, q_1$ for the word consisting of the symbol B, $q_0\, q_0\, q_0\, q_0$ for, e.g., the words ABA and BBA and $q_0\, q_1\, q_2$ for the words BA, and BB. Accepting runs are runs that finish in the final state q_2. For instance, the runs $q_0\, q_1\, q_2$ for BA and BB and $q_0\, q_0\, q_1\, q_2$ for the words ABB, ABA, BBA, and BBB are accepting. Thus, these words belong to $\mathcal{L}(\mathcal{A})$. The word AAA is not accepted by $\mathcal{A}$ since it only has single run, namely $q_0\, q_0\, q_0\, q_0$, which is not accepting.

The accepted language $\mathcal{L}(\mathcal{A})$ is given by the regular expression $(A + B)^* B(A + B)$. Thus, $\mathcal{L}(\mathcal{A})$ is the set of words over $\{\, A, B \,\}$ where the last but one symbol is B. ■

The special cases $Q_0 = \varnothing$ or $F = \varnothing$ are allowed. In both cases, $\mathcal{L}(\mathcal{A}) = \varnothing$. If $F = \varnothing$, then there are no accepting runs. If there are no initial states, then there are no runs at all. Intuitively, the automaton rejects any input word immediately.

An equivalent alternative characterization of the accepted language of an NFA $\mathcal{A}$ is as follows. Let $\mathcal{A}$ be an NFA as above. We extend the transition function δ to the function $\delta^* : Q \times \Sigma^* \to 2^Q$ as follows: $\delta^*(q, \varepsilon) = \{\, q \,\}$, $\delta^*(q, A) = \delta(q, A)$, and

$$\delta^*(q, A_1 A_2 \ldots A_n) = \bigcup_{p \in \delta(q, A_1)} \delta^*(p, A_2 \ldots A_n).$$

Stated in words, $\delta^*(q, w)$ is the set of states that are reachable from q for the input word w. In particular, $\bigcup_{q_0 \in Q_0} \delta^*(q_0, w)$ is the set of all states where a run for w in $\mathcal{A}$ can end. If one of these states is final, then w has an accepting run. Vice versa, if $w \notin \mathcal{L}(\mathcal{A})$, then none of these states is final. Hence, we have the following alternative characterization of the accepted language of an NFA by means of the extended transition function δ^*:

Lemma 4.5. Alternative Characterization of the Accepted Language

Let $\mathcal{A}$ be an NFA. Then:

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \cap F \neq \varnothing \text{ for some } q_0 \in Q_0\}.$$

It can be shown that the language accepted by an NFA constitutes a regular language. In fact, there are algorithms that for a given NFA $\mathcal{A}$ generate a regular expression for the language $\mathcal{L}(\mathcal{A})$. Vice versa, for any regular expression E, an NFA can be constructed that accepts $\mathcal{L}(\mathsf{E})$. Hence, the class of regular languages agrees with the class of languages accepted by an NFA.

An example of a language that is nonregular (but context-free) is $\{ A^n B^n \mid n \geqslant 0 \}$. There does not exist an NFA that accepts it. The intuitive argument for this is that one needs to be able to count the number of A's so as to be able to determine the number of B's that are to follow.

Since NFAs serve to represent (regular) languages we may identify those NFA that accept the same language:

Definition 4.6. Equivalence of NFAs

Let $\mathcal{A}$ and $\mathcal{A}'$ be NFAs with the same alphabet. $\mathcal{A}$ and $\mathcal{A}'$ are called *equivalent* if $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$. ■

A fundamental issue in automata theory is to decide for a given NFA $\mathcal{A}$ whether its accepted language is *empty*, i.e., whether $\mathcal{L}(\mathcal{A}) = \varnothing$. This is known as the *emptiness problem*. From the acceptance condition, it follows directly that $\mathcal{L}(\mathcal{A})$ is nonempty if and only if there is at least one run that ends in some final state. Thus, nonemptiness of $\mathcal{L}(\mathcal{A})$ is equivalent to the existence of an accept state $q \in F$ which is reachable from an initial state $q_0 \in Q_0$. This can easily be determined in time $\mathcal{O}(|\mathcal{A}|)$ using a depth-first search traversal that encounters all states that are reachable from the initial states and checks whether one of them is final. For state $q \in Q$, let $\mathit{Reach}(q) = \bigcup_{w \in \Sigma^*} \delta^*(q, w)$; that is, $\mathit{Reach}(q)$ is the set of states q' that are reachable via an arbitrary run starting in state q.

Theorem 4.7. Language Emptiness is Equivalent to Reachability

Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be an NFA. Then, $\mathcal{L}(\mathcal{A}) \neq \varnothing$ if and only if there exists $q_0 \in Q_0$ and $q \in F$ such that $q \in \mathit{Reach}(q_0)$.

Regular languages exhibit some interesting closure properties, e.g., the union of two regular languages is regular. The same applies to concatenation and Kleene star (finite repetition). This is immediate from the definition of regular languages as those languages that can be generated by regular expressions. They are also closed under intersection and complementation, i.e., if $\mathcal{L}$, $\mathcal{L}_1, \mathcal{L}_2$ are regular languages over the alphabet Σ, then so are $\overline{\mathcal{L}} = \Sigma^* \setminus \mathcal{L}$ and $\mathcal{L}_1 \cap \mathcal{L}_2$.

Let us briefly sketch the proofs for this. In both cases, we may proceed on the basis of finite automata and assume a representation of the given regular languages by NFA $\mathcal{A}$, $\mathcal{A}_1$, and $\mathcal{A}_2$ with the input alphabet Σ that accept the regular languages $\mathcal{L}$, $\mathcal{L}_1$, and $\mathcal{L}_2$, respectively. Intersection can be realized by a product construction $\mathcal{A}_1 \otimes \mathcal{A}_2$ which can be viewed as a parallel composition with synchronization over all symbols $A \in \Sigma$. In fact, the formal definition of $\otimes$ is roughly the same as the synchronization operator $\|$; see Definition 2.26 on page 48. The idea is simply that for the given input word, we run the two automata in parallel and reject as soon as one automaton cannot read the current input symbol, but accept if the input word is fully consumed and both automata accept (i.e., are in a final state).

Definition 4.8. Synchronous Product of NFAs

For NFA $\mathcal{A}_i = (Q_i, \Sigma, \delta_i, Q_{0,i}, F_i)$, with i=1, 2, the *product automaton*

$$\mathcal{A}_1 \otimes \mathcal{A}_2 \;=\; (Q_1 \times Q_2, \Sigma, \delta, Q_{0,1} \times Q_{0,2}, F_1 \times F_2)$$

where δ is defined by

$$\frac{q_1 \xrightarrow{A}_1 q_1' \;\wedge\; q_2 \xrightarrow{A}_2 q_2'}{(q_1, q_2) \xrightarrow{A} (q_1', q_2')}.$$

■

It follows that this product construction of automata corresponds indeed to the intersection of their accepting languages, i.e., $\mathcal{L}(\mathcal{A}_1 \otimes \mathcal{A}_2) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$.

Let us now consider the complementation operator. Given an NFA $\mathcal{A}$ with the input alphabet Σ, we aim to construct an NFA for the complement language $\Sigma^* \setminus \mathcal{L}(\mathcal{A})$. The main step to do so is first to construct an equivalent *deterministic* finite automaton $\mathcal{A}_{det}$ which can be complemented in a quite simple way.

Definition 4.9. Deterministic Finite Automaton (DFA)

Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be an NFA. $\mathcal{A}$ is called *deterministic* if $|Q_0| \leqslant 1$ and $|\delta(q, A)| \leqslant 1$ for all states $q \in Q$ and all symbols $A \in \Sigma$. We will use the abbreviation DFA for a deterministic finite automaton.

DFA $\mathcal{A}$ is called *total* if $|Q_0| = 1$ and $|\delta(q, A)| = 1$ for all $q \in Q$ and all $A \in \Sigma$. ■

Stated in words, an NFA is deterministic if it has at most a single initial state and if for each symbol A the successor state of each state q is either uniquely defined (if $|\delta(q, A)| = 1$) or undefined (if $\delta(q, A) = \varnothing$). Total DFAs provide unique successor states, and thus, unique runs for each input word. Any DFA can be turned into an equivalent total DFA by simply adding a nonfinal trap state, q_{trap} say, that is equipped with a self-loop for any symbol $A \in \Sigma$. From any state $q \neq q_{trap}$, there is a transition to q_{trap} for any symbol A for which q has no A-successor in the given nontotal DFA.

Total DFA are often written in the form $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ where q_0 stands for the unique initial state and δ is a (total) transition function $\delta : Q \times \Sigma \to Q$. Also, the extended transition function δ^* of a total DFA can be viewed as a total function $\delta^* : Q \times \Sigma^* \to Q$, which for given state q and finite word w returns the unique state $p = \delta^*(q, w)$ that is reached from state q for the input word w. In particular, the accepted language of a total DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ is given by

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}.$$

The observation that total DFAs have *exactly one run* for each input word allows complementing a total DFA $\mathcal{A}$ by simply declaring all states to be final that are nonfinal in $\mathcal{A}$ and vice versa. Formally, if $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ is a total DFA then $\overline{\mathcal{A}} = (Q, \Sigma, \delta, q_0, Q \setminus F)$ is a total DFA with $\mathcal{L}(\overline{\mathcal{A}}) = \Sigma^* \setminus \mathcal{L}(\mathcal{A})$. Note that the operator $\mathcal{A} \mapsto \overline{\mathcal{A}}$ applied to a nontotal DFA (or NFA with proper nondeterministic choices) fails to provide an automaton for the complement language (why?).

It remains to explain how to construct for a given NFA $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ an equivalent total DFA $\mathcal{A}_{det}$. This can be done by a *powerset construction*, also often called a *subset construction*, since the states of $\mathcal{A}_{det}$ are the subsets of Q. This allows $\mathcal{A}_{det}$ to simulate $\mathcal{A}$ by moving the prefixes $A_1 \dots A_i$ of the given input word $w = A_1 \dots A_n \in \Sigma$ to the set of states that are reachable in $\mathcal{A}$ for $A_1 \dots A_i$. That is, $\mathcal{A}_{det}$ starts in Q_0, the set of initial states in $\mathcal{A}$. If $\mathcal{A}_{det}$ is in state Q' (which is a subset of $\mathcal{A}$'s state space Q), then $\mathcal{A}_{det}$ moves the input symbol A to $Q'' = \bigcup_{q \in Q'} \delta(q, A)$. If the input word has been consumed and $\mathcal{A}_{det}$ is in a state Q' that contains a state in $\mathcal{A}$'s set of accept states, then $\mathcal{A}_{det}$ accepts. The latter means that there exists an accepting run in $\mathcal{A}$ for the given input word w that ends in an accept state, and hence, $w \in \mathcal{L}(\mathcal{A})$. The formal definition of $\mathcal{A}_{det}$ is $\mathcal{A}_{det} = (2^Q, \Sigma, \delta_{det}, Q_0, F_{det})$ where

$$F_{det} = \{Q' \subseteq Q \mid Q' \cap F \neq \varnothing\}$$

and where the total transition function $\delta_{det} : 2^Q \times \Sigma \to 2^Q$ is defined by

$$\delta_{det}(Q', A) = \bigcup_{q \in Q'} \delta(q, A).$$

Clearly, $\mathcal{A}_{det}$ is a total DFA and, for all finite words $w \in \Sigma^*$, we have

$$\delta^*_{det}(Q_0, w) = \bigcup_{q_0 \in Q_0} \delta^*(q_0, w).$$

Thus, by Lemma 4.5, $\mathcal{L}(\mathcal{A}_{det}) = \mathcal{L}(\mathcal{A})$.

Example 4.10. Determinizing a Nondeterministic Finite Automaton

Consider the NFA depicted in Figure 4.1 on page 153. This automaton is not deterministic as on input symbol B in state q_0 the next state is not uniquely determined. The total DFA that is obtained through the powerset construction is depicted in Figure 4.2. ■

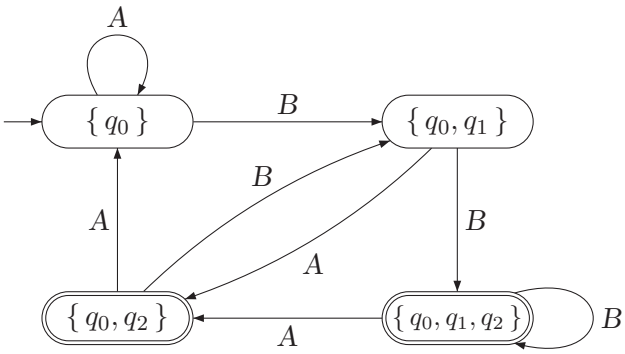


Figure 4.2: A DFA accepting $\mathcal{L}((A + B)^*B(A + B))$.

The powerset construction yields a total DFA that is exponentially larger than the original NFA. In fact, although DFAs and NFAs have the same power (both are equivalent formalisms for regular languages), NFAs can be much more efficient. The regular language given by the regular expression $\mathsf{E}_k = (A + B)^*B(A + B)^k$ (where k is a natural number) is accepted by an NFA with k+2 states (namely?), but it can be shown that there is no equivalent DFA with less than 2^k states. The intuitive argument for the latter is that each DFA for $\mathcal{L}(\mathsf{E}_k)$ needs to "remember" the positions of the symbol B among the last k input symbols which yields $\Omega(2^k)$ states.

We finally mention that for any regular language $\mathcal{L}$ there is a unique DFA $\mathcal{A}$ with $\mathcal{L} = \mathcal{L}(\mathcal{A})$ where the number of states is minimal under all DFAs for $\mathcal{L}$. Uniqueness is understood up to isomorphism, i.e., renaming of the states. (This does not hold for NFA. Why?) There is an algorithm to minimize a given DFA with N states into its equivalent minimal DFA which is based on partition refinement and takes $\mathcal{O}(N \cdot \log N)$ time in the worst case. The concepts of this minimization algorithm are outside the scope of this monograph and can be found in any textbook on automata theory. However, in Chapter 7 a very similar partitioning-refinement algorithm will be presented for bisimulation minimization.

4.2 Model-Checking Regular Safety Properties

In this section, it will be shown how NFAs can be used to check the validity of an important class of safety properties. The main characteristic of these safety properties is that all their bad prefixes constitute a regular language. The bad prefixes of these so-called *regular* safety properties can thus be recognized by an NFA. The main result of this section is that checking a regular safety property on a finite transition system can be reduced to invariant checking on the product of TS and an NFA $\mathcal{A}$ for the bad prefixes. Stated differently, if one wants to check whether a regular safety property holds for TS, it suffices to perform a reachability analysis in the product $TS \otimes \mathcal{A}$ to check a corresponding invariant on $TS \otimes \mathcal{A}$.

4.2.1 Regular Safety Properties

Recall that safety properties are LT properties, i.e., sets of infinite words over 2^{AP}, such that every trace that violates a safety property has a bad prefix that causes a refutation (cf. Definition 3.22 on page 112). Bad prefixes are finite, and thus the set of bad prefixes constitutes a language of finite words over the alphabet $\Sigma = 2^{AP}$. That is, the input symbols $A \in \Sigma$ of the NFA are now sets of atomic propositions. For instance, if $AP = \{ a, b \}$, then $\Sigma = \{ A_1, A_2, A_3, A_4 \}$ consists of the four input symbols $A_1 = \{\}$, $A_2 = \{ a \}$, $A_3 = \{ b \}$, and $A_4 = \{ a, b \}$.[1]

Definition 4.11. Regular Safety Property

Safety property P_{safe} over AP is called *regular* if its set of bad prefixes constitutes a regular language over 2^{AP}. ■

Every invariant is a regular safety property. If Φ is the state condition (propositional formula) of the invariant that should be satisfied by all reachable states, then the language of bad prefixes consists of the words $A_0 A_1 \ldots A_n$ such that $A_i \not\models \Phi$ for some $0 \leqslant i \leqslant n$. Such languages are regular, since they can be characterized by the (casually written) regular notation

$$\Phi^* (\neg \Phi) \, \mathrm{true}^*.$$

Here, Φ stands for the set of all $A \subseteq AP$ with $A \models \Phi$, $\neg\Phi$ for the set of all $A \subseteq AP$ with $A \not\models \Phi$, while true means the set of all subsets A of AP. For instance, if $AP = \{ a, b \}$ and $\Phi = a \vee \neg b$, then

[1] The symbol $\{\}$ denotes the empty subset of AP which serves as symbol in the alphabet $\Sigma = 2^{AP}$. It must be distinguished from the regular expression $\varnothing$ representing the empty language.

- Φ stands for the regular expression $\{\} + \{\, a \,\} + \{\, a, b \,\}$,
- $\neg\Phi$ stands for the regular expression consisting of the symbol $\{\, b \,\}$,
- true stands for the regular expression $\{\} + \{\, a \,\} + \{\, b \,\} + \{\, a, b \,\}$.

The bad prefixes of the invariant over condition $a \vee \neg b$ are given by the regular expression:

$$\mathsf{E} = \underbrace{(\{\} + \{\, a \,\} + \{\, a, b \,\})^*}_{\Phi^*} \underbrace{\{\, b \,\}}_{\neg\Phi} \underbrace{(\{\} + \{\, a \,\} + \{\, b \,\} + \{\, a, b \,\})^*}_{\text{true}^*}.$$

Thus, $\mathcal{L}(\mathsf{E})$ consists of all words $A_1 \ldots A_n$ such that $A_i = \{\, b \,\}$ for some $1 \leqslant i \leqslant n$. Note that, for $A \subseteq AP = \{\, a, b \,\}$, we have $A \not\models a \vee \neg b$ if and only if $A = \{\, b \,\}$. Hence, $\mathcal{L}(\mathsf{E})$ agrees with the set of bad prefixes for the invariant induced by the condition Φ.

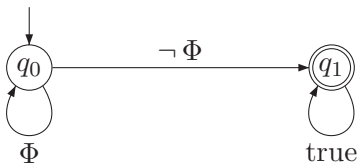


Figure 4.3: NFA accepting all bad prefixes of the invariant over the condition Φ.

In fact, for any invariant P_{inv}, the language of all bad prefixes can be represented by an NFA with two states, as shown in Figure 4.3. Here and in the sequel, we use symbolic notations in the pictures for NFAs over the alphabet 2^{AP}. We use propositional formulae over AP as labels for the edges. Thus, an edge leading from state q to state q' labeled with formula Ψ means that there are transitions $q \xrightarrow{A} q'$ for *all* $A \subseteq AP$ where $A \models \Psi$. E.g., if $AP = \{\, a, b \,\}$ and $\Phi = a \vee \neg b$, then Figure 4.3 is a representation for an NFA with two states q_0, q_1 and the transitions

$$q_0 \xrightarrow{\{\}} q_0, \quad q_0 \xrightarrow{\{a\}} q_0, \quad q_0 \xrightarrow{\{a,b\}} q_0, \quad q_0 \xrightarrow{\{b\}} q_1$$

and

$$q_1 \xrightarrow{\{\}} q_1, \quad q_1 \xrightarrow{\{a\}} q_1, \quad q_1 \xrightarrow{\{b\}} q_1, \quad q_1 \xrightarrow{\{a,b\}} q_1.$$

For the invariant over $AP = \{\, a, b \,\}$ induced by the condition $\Phi = a \vee \neg b$, the minimal bad prefixes are described by the regular expression $(\{\} + \{\, a \,\} + \{\, a, b \,\})^* \{\, b \,\}$. Hence, the minimal bad prefixes constitute a regular language too. An automaton that recognizes all minimal bad prefixes, which are given by the regular expression $\Phi^*(\neg\Phi)$, is obtained from Figure 4.3 by omitting the self-loop of state q_1. In fact, for the definition of regular safety properties it is irrelevant whether the regularity of the set of all bad prefixes or of the set of all minimal bad prefixes is required:

Lemma 4.12. Criterion for Regularity of Safety Properties

The safety property P_{safe} is regular if and only if the set of minimal bad prefixes for P_{safe} is regular.

Proof: "if": Let $\mathcal{A} = (Q, 2^{AP}, \delta, Q_0, F)$ be an NFA for $\mathit{MinBadPref}(P_{safe})$. Then, an NFA for $\mathit{BadPref}(P_{safe})$ is obtained by adding self-loops $q \xrightarrow{A} q$ for all states $q \in F$ and all $A \subseteq AP$. It is easy to check that the modified NFA accepts the language consisting of all bad prefixes for P_{safe}. Thus, $\mathit{BadPref}(P_{safe})$ is regular, which yields the claim.

"only if": Let $\mathcal{A} = (Q, 2^{AP}, \delta, Q_0, F)$ be a DFA for $\mathit{BadPref}(P_{safe})$. For $\mathit{MinBadPref}(P_{safe})$, a DFA is obtained by removing all outgoing transitions from the accept states in $\mathcal{A}$. Let $\mathcal{A}'$ be the modified DFA and let us check that $\mathcal{L}(\mathcal{A}') = \mathit{MinBadPref}(P_{safe})$.

If we are given a word $w = A_1 \dots A_n \in \mathcal{L}(\mathcal{A}')$, then $w \in \mathcal{L}(\mathcal{A})$ since the run $q_0 q_1 \dots q_n$ in $\mathcal{A}'$ for w is also an accepting run in $\mathcal{A}$. Therefore, w is a bad prefix for P_{safe}. Distinguish two cases.

Assume w is not a minimal bad prefix. Then there exists a proper prefix $A_1 \dots A_i$ of w that is a bad prefix for P_{safe}. Thus, $A_1 \dots A_i \in \mathcal{L}(\mathcal{A})$. Since $\mathcal{A}$ is deterministic, $q_0 q_1 \dots q_i$ is the (unique) run for $A_1 \dots A_i$ in $\mathcal{A}$ and $q_i \in F$. Since $i < n$ and q_i has no outgoing transitions in $\mathcal{A}'$, $q_0 \dots q_i \dots q_n$ cannot be a run for $A_1 \dots A_i \dots A_n$ in $\mathcal{A}'$. This contradicts the assumption and shows that $A_1 \dots A_n$ is a minimal bad prefix for P_{safe}.

Vice versa, if w is a minimal bad prefix for P_{safe}, then

(1) $A_1 \dots A_n \in \mathit{BadPref}(P_{safe}) = \mathcal{L}(\mathcal{A})$ and

(2) $A_1 \dots A_i \notin \mathit{BadPref}(P_{safe}) = \mathcal{L}(\mathcal{A})$ for all $1 \leqslant i < n$.

Let $q_0 \dots q_n$ be the unique run for w in $\mathcal{A}$. Then, (2) yields $q_i \notin F$ for $1 \leqslant i < n$, while $q_n \in F$ by (1). Thus, $q_0 \dots q_n$ is an accepting run for w in $\mathcal{A}'$ which yields $w \in \mathcal{L}(\mathcal{A}')$. ■

Example 4.13. Regular Safety Property for Mutual Exclusion Algorithms

Consider a mutual exclusion algorithm such as the semaphore-based one or Peterson's algorithm. The bad prefixes of the safety property P_{mutex} ("there is always at most one process in its critical section") constitute the language of all finite words $A_0 A_1 \dots A_n$ such that

$$\{ \mathit{crit}_1, \mathit{crit}_2 \} \subseteq A_i$$

for some index i with $0 \leqslant i \leqslant n$. If $i{=}n$ is the smallest such index, i.e., $\{ crit_1, crit_2 \} \subseteq A_n$ and $\{ crit_1, crit_2 \} \not\subseteq A_j$ for $0 \leqslant j < n$, then $A_0 \ldots A_n$ is a minimal bad prefix. The language of all (minimal) bad prefixes is regular. An NFA recognizing all minimal bad prefixes is depicted in Figure 4.4. ■

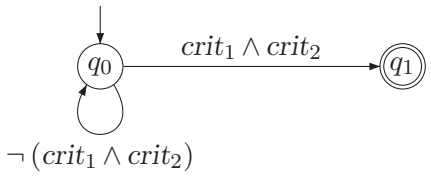


Figure 4.4: Minimal bad prefixes that refute the mutual exclusion property.

Example 4.14. Regular Safety Property for the Traffic Light

Consider a traffic light with three possible colors: red, yellow and green. The property "a red phase must be preceded immediately by a yellow phase" is specified by the set of infinite words $\sigma = A_0\, A_1 \ldots$ with $A_i \subseteq \{ red, yellow \}$ such that for all $i \geqslant 0$ we have that

$$red \in A_i \text{ implies } i > 0 \text{ and } yellow \in A_{i-1}.$$

The bad prefixes are finite words that violate this condition. Examples of bad prefixes that are minimal are

$$\{\}\,\{\}\,\{ red \} \quad \text{and} \quad \{\}\,\{ red \}.$$

In general, the minimal bad prefixes are words of the form $A_0\, A_1 \ldots A_n$ such that $n > 0$, $red \in A_n$, and $yellow \notin A_{n-1}$. The NFA in Figure 4.5 accepts these minimal bad prefixes. Recall the meaning of the edge labels in the pictorial representations of an NFA over the

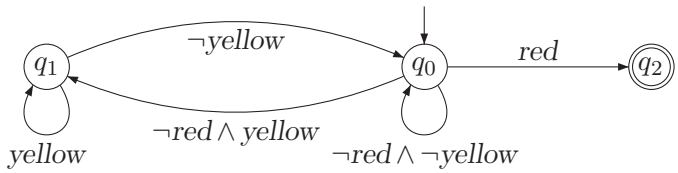


Figure 4.5: Minimal bad prefixes in which red is not preceded by yellow.

alphabet $\Sigma = 2^{AP}$ where $AP = \{ yellow, red \}$. For instance, the edge-label *yellow* in the self-loop of state q_1 denotes a formula, namely the positive literal $yellow \in AP$. This stands for all sets $A \subseteq AP = \{ yellow, red \}$ where the literal *yellow* holds, that is, the sets $\{ yellow \}$ and $\{ yellow, red \}$. Hence, the self-loop of q_1 in the picture stands for the two

transitions:

$$q_1 \xrightarrow{\{yellow\}} q_1 \quad \text{and} \quad q_1 \xrightarrow{\{yellow,red\}} q_1.$$

Similarly, the edge label $\neg yellow$ in the transition from q_1 to q_0 stands for the negative literal $\neg yellow$, and thus represents the transitions:

$$q_1 \xrightarrow{\{red\}} q_0 \quad \text{and} \quad q_1 \xrightarrow{\{\}} q_0.$$

In the same way the label *red* of the transition from q_0 to q_2 represents two transitions (with the labels $\{ red \}$ and $\{ red, yellow \}$), while the edge labels $\neg red \wedge yellow$ and $\neg red \wedge \neg yellow$ denote only a single symbol in 2^{AP}, namely $\{ yellow \}$ and $\{\}$, respectively. ■

Example 4.15. A Nonregular Safety Property

Not all safety properties are regular. As an example of a nonregular safety property, consider:

"The number of inserted coins is always at least the number of dispensed drinks."

(See also Example 3.24 on page 113). Let the set of propositions be $\{ pay, drink \}$. Minimal bad prefixes for this safety property constitute the language

$$\{ pay^n \, drink^{n+1} \mid n \geqslant 0 \}$$

which is not a regular, but a context-free language. Such safety properties fall outside the scope of the following verification algorithm. ■

4.2.2 Verifying Regular Safety Properties

Let P_{safe} be a *regular* safety property over the atomic propositions AP and $\mathcal{A}$ an NFA recognizing the (minimal) bad prefixes of P_{safe}. (Recall that by Lemma 4.12 on page 161 it is irrelevant whether $\mathcal{A}$ accepts all bad prefixes for P_{safe} or only the minimal ones.) For technical reasons, we assume that $\varepsilon \notin \mathcal{L}(\mathcal{A})$. In fact, this is not a severe restriction since otherwise all finite words over 2^{AP} are bad prefixes, and hence, $P_{safe} = \varnothing$. In this case, $TS \models P_{safe}$ if and only if TS has no initial state.

Furthermore, let TS be a *finite* transition system without terminal states with corresponding set of propositions AP. In this section, we aim to establish an algorithmic method for verifying whether TS satisfies *regular* safety property P_{safe}, i.e., to check whether

$TS \models P_{safe}$ holds. According to Lemma 3.25 on page 114 we have

$$TS \models P_{safe} \quad \text{if and only if} \quad \mathit{Traces}_{\mathit{fin}}(TS) \cap \mathit{BadPref}(P_{safe}) = \varnothing$$

$$\text{if and only if} \quad \mathit{Traces}_{\mathit{fin}}(TS) \cap \mathcal{L}(\mathcal{A}) = \varnothing.$$

Thus, it suffices to check whether $\mathit{Traces}_{\mathit{fin}}(TS) \cap \mathcal{L}(\mathcal{A}) = \varnothing$ to establish $TS \models P_{safe}$.

To do so, we adopt a similar strategy as for checking whether two NFAs intersect. Recall that in order to check whether the NFAs $\mathcal{A}_1$ and $\mathcal{A}_2$ do intersect, it suffices to consider their product automaton, so

$$\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2) \;=\; \varnothing \quad \text{if and only if} \quad \mathcal{L}(\mathcal{A}_1 \otimes \mathcal{A}_2) \;=\; \varnothing.$$

The question whether two automata do intersect is thus reduced to a simple reachability problem in the product automaton.

This is now exploited as follows. In order to check whether $\mathit{Traces}_{\mathit{fin}}(TS) \cap \mathcal{L}(\mathcal{A}) = \varnothing$, we first build a product of transition system TS and NFA $\mathcal{A}$ in the same vein as the synchronous product of NFA. This yields the transition system $TS \otimes \mathcal{A}$. For this transition system, an invariant can be given using a propositional logic formula Φ –derived from the accept states of $\mathcal{A}$– such that $\mathit{Traces}_{\mathit{fin}}(TS) \cap \mathcal{L}(\mathcal{A}) = \varnothing$ if and only if $TS \otimes \mathcal{A} \models$ "always Φ". In this way, the verification of a regular safety property is reduced to invariant checking. Recall that for checking invariants, Algorithm 4 (see page 110) can be exploited.

We start by formally defining the product between a transition system TS and an NFA $\mathcal{A}$, denoted $TS \otimes \mathcal{A}$. Let $TS = (S, Act, \rightarrow, I, AP, L)$ and $\mathcal{A} = (Q, 2^{AP}, \delta, Q_0, F)$ with $Q_0 \cap F = \varnothing$. Recall that the alphabet of $\mathcal{A}$ consists of sets of atomic proposition in TS. Transition system $TS \otimes \mathcal{A}$ has state space $S \times Q$ and a transition relation such that each path fragment $\pi = s_0\, s_1 \ldots s_n$ in TS can be extended to a path fragment

$$\langle s_0, q_1 \rangle \, \langle s_1, q_2 \rangle \ldots \langle s_n, q_{n+1} \rangle$$

in $TS \otimes \mathcal{A}$ which has an initial state $q_0 \in Q_0$ for which

$$q_0 \xrightarrow{L(s_0)} q_1 \xrightarrow{L(s_1)} q_2 \xrightarrow{L(s_2)} \ldots \xrightarrow{L(s_n)} q_{n+1}$$

is a run—not necessarily accepting—of NFA $\mathcal{A}$ that generates the word

$$\mathit{trace}(\pi) \;=\; L(s_0)\, L(s_1) \ldots L(s_n).$$

Finally, labels of states are state names of $\mathcal{A}$. These considerations lead to the following definition:

Definition 4.16. Product of Transition System and NFA

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system without terminal states and $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ an NFA with the alphabet $\Sigma = 2^{AP}$ and $Q_0 \cap F = \varnothing$. The product transition system $TS \otimes \mathcal{A}$ is defined as follows:

$$TS \otimes \mathcal{A} \;=\; (S', Act, \rightarrow', I', AP', L')$$

where

- $S' = S \times Q$,
- $\rightarrow'$ is the smallest relation defined by the rule

$$\frac{s \xrightarrow{\alpha} t \;\wedge\; q \xrightarrow{L(t)} p}{\langle s, q\rangle \xrightarrow{\alpha}{}' \langle t, p\rangle},$$

- $I' = \{\, \langle s_0, q\rangle \mid s_0 \in I \;\wedge\; \exists q_0 \in Q_0.\; q_0 \xrightarrow{L(s_0)} q \,\}$,
- $AP' = Q$, and
- $L' : S \times Q \to 2^Q$ is given by $L'(\langle s, q\rangle) \;=\; \{\, q \,\}$.

■

Remark 4.17. Terminal States

For the definition of LT properties (and thus of invariants) we have assumed transition systems to have no terminal states. It is, however, not guaranteed that $TS \otimes \mathcal{A}$ possesses this property, even if TS does. This stems from the fact that in NFA $\mathcal{A}$ there may be a state q, say, that has no direct successor states for some set A of atomic propositions, i.e., with $\delta(q, A) = \varnothing$. This technical problem can be treated by either requiring $\delta(q, A) \neq \varnothing$ for all states $q \in Q$ and $A \subseteq AP$ or by extending the notion of invariants to arbitrary transition systems. Note that imposing the requirement $\delta(q, A) \neq \varnothing$ is not a severe restriction, as any NFA can be easily transformed into an equivalent one that satisfies this property by introducing a state q_{trap} and adding transition $q \xrightarrow{A} q_{trap}$ to $\mathcal{A}$ whenever $\delta(q, A) = \varnothing$ or $q = q_{trap}$. We finally remark that for the algorithm for invariant checking, it is not of any relevance whether terminal states exist or not. ■

Example 4.18. A Product Automaton

The language of the minimal bad prefixes of the safety property "each red light phase

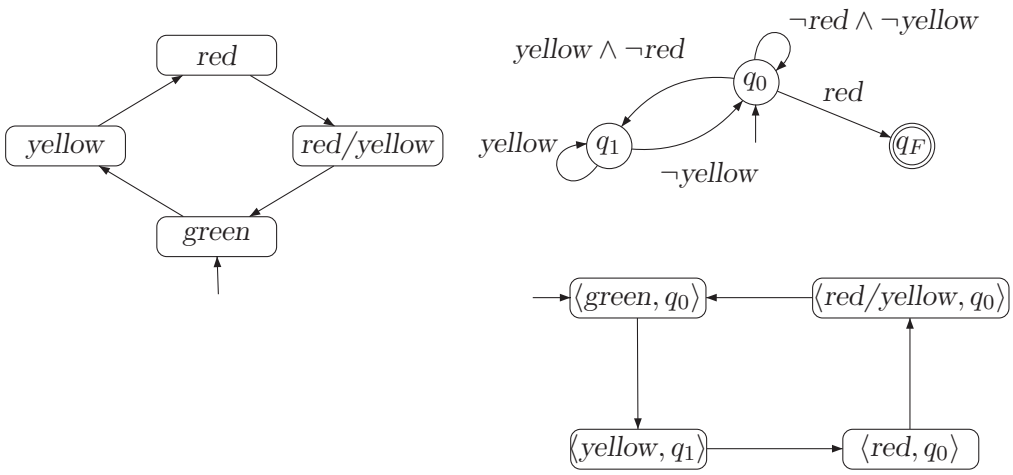


Figure 4.6: German traffic light (left upper figure), an NFA (right upper figure), and their product (lower figure).

is preceded by a yellow light phase" is accepted by the DFA $\mathcal{A}$ indicated in Example 4.14 (page 162). We consider a German traffic light, which besides the usual possible colors red, green, and yellow, has the possibility to indicate red and yellow simultaneously indicating "green light soon". The transition system *GermanTrLight* thus has four states with the usual transitions *red* $\rightarrow$ *red+yellow*, *red+yellow* $\rightarrow$ *green*, *green* $\rightarrow$ *yellow*, and *yellow* $\rightarrow$ *red*. Let $AP = \{ red, yellow \}$ indicating the corresponding light phases. The labeling is defined as follows: $L(red) = \{ red \}$, $L(yellow) = \{ yellow \}$, $L(green) = \varnothing = L(red{+}yellow)$. The product transition system $GermanTrLight \otimes \mathcal{A}$ consists of four reachable states (see Figure 4.6). As action labels are not relevant here, they are omitted. ■

The following theorem shows that the verification of a regular safety property can be reduced to checking an invariant in the product.

Let *TS* and $\mathcal{A}$ be as before. Let $P_{inv(\mathcal{A})}$ be the invariant over $AP' = 2^Q$ which is defined by the propositional formula

$$\bigwedge_{q \in F} \neg\, q.$$

In the sequel, we often write $\neg\, F$ as shorthand for $\bigwedge_{q \in F} \neg\, q$. Stated in words, $\neg F$ holds in all nonaccept states.

Theorem 4.19. Verification of Regular Safety Properties

For transition system TS over AP, NFA $\mathcal{A}$ with alphabet 2^{AP} as before, and regular safety property P_{safe} over AP such that $\mathcal{L}(\mathcal{A})$ equals the set of (minimal) bad prefixes of P_{safe}, the following statements are equivalent:

(a) $TS \models P_{safe}$

(b) $Traces_{fin}(TS) \cap \mathcal{L}(\mathcal{A}) = \varnothing$

(c) $TS \otimes \mathcal{A} \models P_{inv(A)}$

Proof: Let $TS = (S, Act, \rightarrow, I, AP, L)$ and $\mathcal{A} = (Q, 2^{AP}, \delta, Q_0, F)$.

The equivalence of (a) and (b) follows immediately by Lemma 3.25 (page 114). To establish the equivalence of (a), (b), and (c), we show

$$(c) \Longrightarrow (a): \; TS \not\models P_{safe} \quad \text{implies} \quad TS \otimes \mathcal{A} \not\models P_{inv(A)}$$

and

$$(b) \Longrightarrow (c): \; TS \otimes \mathcal{A} \not\models P_{inv(A)} \quad \text{implies} \quad Traces_{fin}(TS) \cap \mathcal{L}(\mathcal{A}) \neq \varnothing.$$

Proof of "(c) $\Longrightarrow$ (a)": If $TS \not\models P_{safe}$, then there is a finite initial path fragment $\widehat{\pi} = s_0\, s_1 \ldots s_n$ in TS with

$$trace(\widehat{\pi}) = L(s_0)\, L(s_1) \ldots L(s_n) \in \mathcal{L}(\mathcal{A}).$$

Since $trace(\widehat{\pi}) \in \mathcal{L}(\mathcal{A})$, there exists an accepting run $q_0\, q_1 \ldots q_{n+1}$ of $\mathcal{A}$ for $trace(\widehat{\pi})$. Accordingly

$$q_0 \in Q_0 \text{ and } q_i \xrightarrow{L(s_i)} q_{i+1} \text{ for all } 0 \leqslant i \leqslant n, \text{ and } q_{n+1} \in F.$$

Thus, $\langle s_0, q_1\rangle\, \langle s_1, q_2\rangle \ldots \langle s_n, q_{n+1}\rangle$ is an initial path fragment in $TS \otimes \mathcal{A}$ with

$$\langle s_n, q_{n+1}\rangle \not\models \neg F.$$

It thus follows that $TS \otimes \mathcal{A} \not\models P_{inv(A)}$.

Proof of "(b) $\Longrightarrow$ (c)": Let $TS \otimes \mathcal{A} \not\models P_{inv(A)}$. Then there exists an initial path fragment

$$\langle s_0, q_1\rangle \ldots \langle s_n, q_{n+1}\rangle$$

in $TS \otimes \mathcal{A}$ with $q_{n+1} \in F$, and $q_1, \ldots, q_n \notin F$. Besides, $s_0\, s_1 \ldots s_n$ is an initial path fragment in TS. Further,

$$q_i \xrightarrow{L(s_i)} q_{i+1} \text{ for all } 0 \leqslant i \leqslant n.$$

Since $\langle s_0, q_1 \rangle$ is an initial state of $TS \otimes \mathcal{A}$, there is an initial state q_0 in $\mathcal{A}$ such that $q_0 \xrightarrow{L(s_0)} q_1$. Sequence $q_0\, q_1 \ldots q_{n+1}$ is thus an accepting run for $\mathit{trace}(s_0\, s_1 \ldots s_n)$. Thus,

$$\mathit{trace}(s_0\, s_1 \ldots s_n) \in \mathit{Traces}_{\mathit{fin}}(TS) \cap \mathcal{L}(\mathcal{A})$$

which yields $\mathit{Traces}_{\mathit{fin}}(TS) \cap \mathcal{L}(\mathcal{A}) \neq \varnothing$.

■

Stated in words, Theorem 4.19 yields that in order to check the transition system TS versus the regular safety property P_{safe}, it suffices to check whether no state $\langle s, q \rangle$ in $TS \otimes \mathcal{A}$ is reachable where the $\mathcal{A}$-component q is an accept state in $\mathcal{A}$. This invariant "visit never an accept state in $\mathcal{A}$" (formally given by the invariant condition $\Phi = \neg F$) can be checked using a depth-first search approach as described in detail in Algorithm 4 (page 110). Note that in case the safety property is refuted, the invariant checking algorithm provides a counterexample. This counterexample is in fact a finite path fragment $\langle s_0, q_1 \rangle \langle s_1, q_2 \rangle \ldots \langle s_n, q_{n+1} \rangle$ in the transition system $TS \otimes \mathcal{A}$ that leads to an accept state. The projection to the states in TS yields an initial finite path fragment $s_0\, s_1 \ldots s_n$ in TS where the induced trace $\mathit{trace}(s_0 s_1 \ldots s_n) \in (2^{AP})^*$ is accepted by $\mathcal{A}$ (since it has an accepting run of the form $q_0\, q_1 \ldots q_{n+1}$). Thus, $\mathit{trace}(s_0\, s_1 \ldots s_n)$ is a bad prefix for P_{safe}. Hence, $s_0\, s_1 \ldots s_n$ yields a useful error indication since $\mathit{trace}(\pi) \notin P_{\mathit{safe}}$ for all paths π in TS that start with the prefix $s_0\, s_1 \ldots s_n$.

Corollary 4.20.

Let TS, $\mathcal{A}$, and P_{safe} be as in Theorem 4.19. Then, for each initial path fragment $\langle s_0, q_1 \rangle \ldots \langle s_n, q_{n+1} \rangle$ of $TS \otimes \mathcal{A}$:

$$q_1, \ldots, q_n \notin F \text{ and } q_{n+1} \in F \quad \text{implies} \quad \mathit{trace}(s_0\, s_1 \ldots s_n) \in \mathcal{L}(\mathcal{A}).$$

As a result, the skeleton in Algorithm 5 can be used to check a regular safety property against a transition system and to report a counterexample (i.e., finite initial path fragment in TS inducing a bad prefix) as diagnostic feedback if the safety property does not hold for TS.

Example 4.21. Checking a Regular Safety Property for the Traffic Light

Consider again the German traffic light system and the regular safety property P_{safe}

Algorithm 5 Model-checking algorithm for regular safety properties

Input: finite transition system TS and regular safety property P_{safe}
Output: true if $TS \models P_{safe}$. Otherwise false plus a counterexample for P_{safe}.

Let NFA $\mathcal{A}$ (with accept states F) be such that $\mathcal{L}(\mathcal{A})$ = bad prefixes of P_{safe}
Construct the product transition system $TS \otimes \mathcal{A}$
Check the invariant $P_{inv(\mathcal{A})}$ with proposition $\neg F = \bigwedge_{q \in F} \neg q$ on $TS \otimes \mathcal{A}$.

if $TS \otimes \mathcal{A} \models P_{inv(\mathcal{A})}$ **then**
  return true
else
  Determine an initial path fragment $\langle s_0, q_1 \rangle \ldots \langle s_n, q_{n+1} \rangle$ of $TS \otimes \mathcal{A}$ with $q_{n+1} \in F$
  return (false, $s_0\, s_1 \ldots s_n$)
fi

that each red light phase should be immediately preceded by a yellow light phase. The transition system of the traffic light, the NFA accepting the bad prefixes of the safety property, as well as their product automaton, are depicted in Figure 4.6 (page 166). To check the validity of P_{safe}, only the second component of the states $\langle s, q \rangle$ is relevant. The fact that no state of the form $\langle \ldots, q_F \rangle$ is reachable ensures the invariant $\neg q_F$ to hold in all reachable states. Thus $GermanTrLight \models P_{safe}$.

If the traffic light is modified such that the state "red" is the initial state (instead of "green"), then we obtain a transition system that violates P_{safe}. Actually, in this case the invariant $\neg q_F$ is already violated in the initial state of the resulting product transition system that has the following form:

$$\langle red, \delta(q_0, \{red\}) \rangle \;=\; \langle red, q_F \rangle.$$

■

We conclude this part by considering the worst-case time and space complexity of the automata-based algorithm for checking regular safety properties.

Theorem 4.22. Complexity of Verifying Regular Safety Properties

The time and space complexity of Algorithm 5 is in $\mathcal{O}(|TS| \cdot |\mathcal{A}|)$ *where* $|TS|$ *and* $|\mathcal{A}|$ *denote the number of states and transitions in* TS *and* $\mathcal{A}$*, respectively.*

Assuming an generation of the reachable states of TS from a syntactic description of the processes, the above bound also holds if $|TS|$ denotes the size of the reachable fragment of TS.

Proof: Follows directly from the fact that the number of states in the product automaton $TS \otimes \mathcal{A}$ is in $\mathcal{O}(|S| \cdot |Q|)$ (where S and Q denote the state space of TS and $\mathcal{A}$, respectively) and the fact that the time and space complexity of invariant checking is linear in the number of states and transitions of the transition system $TS \otimes \mathcal{A}$. (Thus, we can even establish the bound $\mathcal{O}(|S| \cdot |Q| + | \rightarrow | \cdot |\delta|)$ for the runtime where $| \rightarrow |$ denotes the number of transitions in TS and $|\delta|$ the number of transitions in $\mathcal{A}$.) ■

4.3 Automata on Infinite Words

Finite-state automata accept finite words, i.e., sequences of symbols of finite length, and yield the basis for checking regular safety properties. In this and the following sections, these ideas are generalized toward a more general class of LT properties. These include regular safety, various liveness properties, but also many other properties that are relevant to formalize the requirements for "realistic" systems. The rough idea is to consider variants of NFAs, called nondeterministic Büchi automata (NBAs), which serve as acceptors for languages of infinite words. It will be established that if we are given a nondeterministic Büchi automaton $\mathcal{A}$ that specifies the "bad traces" (i.e., that accepts the complement of the LT property P to be verified), then a graph analysis in the product of the given transition system TS and the automaton $\mathcal{A}$ suffices to either establish or disprove $TS \models P$. Whereas for regular safety properties a reduction to invariant checking (i.e., a depth-first search) is possible, the graph algorithms needed here serve to check a so-called persistence property. Such properties state that eventually for ever a certain proposition holds.

We first introduce the "ω-counterpart" to regular languages, both by introducing ω-regular expressions (Section 4.3.1) and nondeterministic Büchi automata (see Section 4.3.2). Variants of nondeterministic Büchi automata will be discussed in Sections 4.3.3 and 4.3.4.

4.3.1 ω-Regular Languages and Properties

Infinite words over the alphabet Σ are infinite sequences $A_0\, A_1\, A_2 \ldots$ of symbols $A_i \in \Sigma$. Σ^ω denotes the set of all infinite words over Σ. As before, the Greek letter σ will be used for infinite words, while w, v, u range over finite words. Any subset of Σ^ω is called a language of infinite words, sometimes also called an ω-language. In the sequel, the notion of a language will be used for any subset of $\Sigma^* \cup \Sigma^\omega$. Languages will be denoted by the symbol $\mathcal{L}$.

To reason about languages of infinite words, the basic operations of regular expressions (union, concatenation, and finite repetition) are extended by *infinite* repetition, denoted by the Greek letter ω.[2] For instance, the infinite repetition of the finite word AB yields the infinite word $ABABABABAB\ldots$ (ad infinitum) and is denoted by $(AB)^\omega$. For the special case of the empty word, we have $\varepsilon^\omega = \varepsilon$. For an infinite word, infinite repetition has no effect, that is, $\sigma^\omega = \sigma$ if $\sigma \in \Sigma^\omega$. Note that the finite repetition of a word results in a language of finite words, i.e., a subset of Σ^*, whereas infinite repetition of a (finite or infinite) word results in a single word.

Infinite repetition can be lifted to languages as follows. For language $\mathcal{L} \subseteq \Sigma^*$, let $\mathcal{L}^\omega$ be the set of words in $\Sigma^* \cup \Sigma^\omega$ that arise from the infinite concatenation of (arbitrary) words in Σ, i.e.,

$$\mathcal{L}^\omega \;=\; \{w_1 w_2 w_3 \ldots \mid w_i \in \mathcal{L}, i \geqslant 1\}.$$

The result is an ω-language, provided that $\mathcal{L} \subseteq \Sigma^+$, i.e., $\mathcal{L}$ does not contain the empty word ε. However, in the sequel, we only need the ω-operator applied to languages of finite words that do not contain the empty word. In this case, i.e., for $\mathcal{L} \subseteq \Sigma^+$, we have $\mathcal{L}^\omega \subseteq \Sigma^\omega$.

In the following definition, the concatenation operator $\mathcal{L}_1.\mathcal{L}_2$ is used that combines a language $\mathcal{L}_1$ of finite words with a language $\mathcal{L}_2$ of infinite words. It is defined by $\mathcal{L}_1.\mathcal{L}_2 = \{w\sigma \mid w \in \mathcal{L}_1, \sigma \in \mathcal{L}_2\}$.

Definition 4.23. ω-Regular Expression

An *ω-regular expression* G over the alphabet Σ has the form

$$\mathsf{G} = \mathsf{E}_1.\mathsf{F}_1^\omega + \ldots + \mathsf{E}_n.\mathsf{F}_n^\omega$$

where $n \geqslant 1$ and $\mathsf{E}_1, \ldots, \mathsf{E}_n, \mathsf{F}_1, \ldots, \mathsf{F}_n$ are regular expressions over Σ such that $\varepsilon \notin \mathcal{L}(\mathsf{F}_i)$, for all $1 \leqslant i \leqslant n$.

The semantics of the ω-regular expression G is a language of infinite words, defined by

$$\mathcal{L}_\omega(\mathsf{G}) \;=\; \mathcal{L}(\mathsf{E}_1).\mathcal{L}(\mathsf{F}_1)^\omega \cup \ldots \cup \mathcal{L}(\mathsf{E}_n).\mathcal{L}(\mathsf{F}_n)^\omega$$

where $\mathcal{L}(\mathsf{E}) \subseteq \Sigma^*$ denotes the language (of finite words) induced by the regular expression E (see page 914).

Two ω-regular expressions G_1 and G_2 are *equivalent*, denoted $\mathsf{G}_1 \equiv \mathsf{G}_2$, if $\mathcal{L}_\omega(\mathsf{G}_1) = \mathcal{L}_\omega(\mathsf{G}_2)$. ■

[2] The symbol ω denotes the first infinite ordinal. It already appeared in the notation Σ^ω for the set of infinite words over the alphabet Σ.

Examples for ω-regular expressions over the alphabet $\Sigma = \{ A, B, C \}$ are

$$(A + B)^* A(AAB + C)^\omega \quad \text{or} \quad A(B + C)^* A^\omega + B(A + C)^\omega.$$

If E is a regular expression with $\varepsilon \notin \mathcal{L}(\mathsf{E})$, then also E^ω can be viewed as an ω-regular expression since it can be identified with $\mathsf{E}.\mathsf{E}^\omega$ or $\varepsilon.\mathsf{E}^\omega$. Note that we have $\mathcal{L}(\mathsf{E})^\omega = \mathcal{L}(\mathsf{E}.\mathsf{E}^\omega) = \mathcal{L}(\varepsilon.\mathsf{E}^\omega)$.

Definition 4.24. ω-Regular Language

A language $\mathcal{L} \subseteq \Sigma^\omega$ is called ω-regular if $\mathcal{L} = \mathcal{L}_\omega(\mathsf{G})$ for some ω-regular expression G over Σ. ■

For instance, the language consisting of all infinite words over $\{A, B\}$ that contain infinitely many A's is ω-regular since it is given by the ω-regular expression $(B^*A)^\omega$. The language consisting of all infinite words over $\{A, B\}$ that contain only finitely many A's is ω-regular too. A corresponding ω-regular expression is $(A+B)^*B^\omega$. The empty set is ω-regular since it is obtained, e.g., by the ω-regular expression $\varnothing^\omega$. More generally, if $\mathcal{L} \subseteq \Sigma^*$ is regular and $\mathcal{L}'$ is ω-regular, then $\mathcal{L}^\omega$ and $\mathcal{L}.\mathcal{L}'$ are ω-regular.

ω-Regular languages possess several closure properties: they are closed under union, intersection, and complementation. The argument for union is obvious from the definition by ω-regular expressions. The proof for the intersection will be provided later; see Corollary 4.60 on page 198. The more advanced proof for complementation is not provided in this monograph. We refer the interested reader to [174] that covers also other properties of ω-regular languages and various other automata models.

The concepts of ω-regular languages play an important role in verification since most relevant LT properties are ω-regular:

Definition 4.25. ω-Regular Properties

LT property P over AP is called *ω-regular* if P is an ω-regular language over the alphabet 2^{AP}. ■

For instance, for $AP = \{ a, b \}$, the invariant P_{inv} induced by the proposition $\Phi = a \vee \neg b$ is an ω-regular property since

$$\begin{aligned} P_{inv} &= \Big\{ A_0 A_1 A_2 \ldots \in (2^{AP})^\omega \mid \forall i \geqslant 0. \, (a \in A_i \text{ or } b \notin A_i) \Big\} \\ &= \Big\{ A_0 A_1 A_2 \ldots \in (2^{AP})^\omega \mid \forall i \geqslant 0. \, (A_i \in \{\{\}, \{a\}, \{a, b\}\} \Big\} \end{aligned}$$

is given by the ω-regular expression $\mathsf{E} = (\{\} + \{a\} + \{a,b\})^\omega$ over the alphabet $\Sigma = 2^{AP} = \{\{\},\{a\},\{b\},\{a,b\}\}$. In fact, any invariant over AP is ω-regular (the set AP of atomic propositions is arbitrary) as it can be described by the ω-regular expression Φ^ω where Φ denotes the underlying propositional formula (that has to hold for all reachable states) and is identified with the regular expression given by the sum of all $A \subseteq AP$ with $A \models \Phi$.

Also, any regular safety property P_{safe} is an ω-regular property. This follows from the fact that the complement language

$$(2^{AP})^\omega \setminus P_{safe} \;=\; \underbrace{\mathit{BadPref}(P_{safe})}_{\text{regular}}.(2^{AP})^\omega$$

is an ω-regular language. The result that ω-regular languages are closed under complementation (stated above, in the end of Section 4.3.1 on page 172) yields the claim.

Example 4.26. Mutual Exclusion

Another example of an ω-regular property is the property given by the informal statement "process $\mathcal{P}$ visits its critical section infinitely often" which, for $AP = \{\, \mathit{wait}, \mathit{crit} \,\}$, can be formalized by the ω-regular expression:

$$((\underbrace{\{\} + \{\, \mathit{wait} \,\}}_{\text{negative literal } \neg \mathit{crit}})^*.(\underbrace{\{\, \mathit{crit} \,\} + \{\, \mathit{wait}, \mathit{crit} \,\}}_{\text{positive literal } \mathit{crit}}))^\omega.$$

When allowing a somewhat sloppy notation using propositional formulae, the above expression may be rewritten into $((\neg \mathit{crit})^*.\mathit{crit})^\omega$.

Starvation freedom in the sense of "whenever process $\mathcal{P}$ is waiting then it will enter its critical section eventually later" is an ω-regular property as it can be described by

$$((\neg \mathit{wait})^*.\mathit{wait}.\mathrm{true}^*.\mathit{crit})^\omega \;+\; ((\neg \mathit{wait})^*.\mathit{wait}.\mathrm{true}^*.\mathit{crit})^*.(\neg \mathit{wait})^\omega$$

which is a short form for the ω-regular expression over $AP = \{\, \mathit{wait}, \mathit{crit} \,\}$ that results by replacing $\neg \mathit{wait}$ with $\{\} + \{\mathit{crit}\}$, wait with $\{\mathit{wait}\} + \{\mathit{wait}, \mathit{crit}\}$, true with $\{\} + \{\mathit{crit}\} + \{\mathit{wait}\} + \{\mathit{wait}, \mathit{crit}\}$, and crit with $\{\mathit{crit}\} + \{\mathit{wait}, \mathit{crit}\}$. Intuitively, the first summand in the above expression stands for the case where $\mathcal{P}$ requests and enters its critical section infinitely often, while the second summand stands for the case where $\mathcal{P}$ is in its waiting phase only finitely many times. ■

4.3.2 Nondeterministic Büchi Automata

The issue now is to provide a kind of automaton that is suited for accepting ω-regular languages. Finite automata are not adequate for this purpose as they operate on finite

words, while we need an acceptor for infinite words. Automata models that recognize languages of infinite words are called ω-automata. The accepting runs of an ω-automaton have to "check" the entire input word (and not just a finite prefix thereof), and thus have to be infinite. This implies that acceptance criteria for infinite runs are needed.

In this monograph, the simplest variant of ω-automata, called *nondeterministic Büchi automata* (NBAs), suffices. The syntax of NBAs is exactly the same as for nondeterministic finite automata (NFAs). NBAs and NFAs differ, however, in their semantics: the accepted language of an NFA $\mathcal{A}$ is a language of finite words, i.e., $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$, whereas the accepted language of NBA $\mathcal{A}$ (denoted $\mathcal{L}_\omega(\mathcal{A})$ is an ω-language, i.e., $\mathcal{L}_\omega(\mathcal{A}) \subseteq \Sigma^\omega$. The intuitive meaning of the acceptance criterion named after Büchi is that the accept set of $\mathcal{A}$ (i.e., the set of accept states in $\mathcal{A}$) has to be visited infinitely often. Thus, the accepted language $\mathcal{L}_\omega(\mathcal{A})$ consists of all infinite words that have a run in which some accept state is visited infinitely often.

Definition 4.27. Nondeterministic Büchi Automaton (NBA)

A *nondeterministic Büchi automaton* (NBA) $\mathcal{A}$ is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ where

- Q is a finite set of states,
- Σ is an alphabet,
- $\delta : Q \times \Sigma \to 2^Q$ is a transition function,
- $Q_0 \subseteq Q$ is a set of initial states, and
- $F \subseteq Q$ is a set of *accept* (or: final) states, called the *acceptance set*.

A run for $\sigma = A_0 A_1 A_2 \ldots \in \Sigma^\omega$ denotes an infinite sequence $q_0\, q_1\, q_2 \ldots$ of states in $\mathcal{A}$ such that $q_0 \in Q_0$ and $q_i \xrightarrow{A_i} q_{i+1}$ for $i \geqslant 0$. Run $q_0\, q_1\, q_2 \ldots$ is *accepting* if $q_i \in F$ for infinitely many indices $i \in \mathbb{N}$. The *accepted language* of $\mathcal{A}$ is

$$\mathcal{L}_\omega(\mathcal{A}) \;=\; \{\, \sigma \in \Sigma^\omega \mid \text{there exists an accepting run for } \sigma \text{ in } \mathcal{A} \,\}.$$

The size of $\mathcal{A}$, denoted $|\mathcal{A}|$, is defined as the number of states and transitions in $\mathcal{A}$. ■

As for an NFA, we identify the transition function δ with the induced transition relation $\to \subseteq Q \times \Sigma \times Q$ which is given by

$$q \xrightarrow{A} p \;\text{ if and only if }\; p \in \delta(q, A).$$

Since the state space Q of an NBA $\mathcal{A}$ is finite, each run for an infinite word $\sigma \in \Sigma^\omega$ is infinite, and hence visits some state $q \in Q$ infinitely often. Acceptance of a run depends on whether or not the set of all states that appear infinitely often in the given run contains an accept state. The definition of an NBA allows for the special case where $F = \varnothing$, which means that there are no accept states. Clearly, in this case, no run is accepting. Thus $\mathcal{L}_\omega(\mathcal{A}) = \varnothing$ if $F = \varnothing$. There are also no accepting runs whenever, $Q_0 = \varnothing$ as in this case, no word has a run.

Example 4.28.

Consider the NBA of Figure 4.7 with the alphabet $\Sigma = \{A, B, C\}$. The word C^ω has only

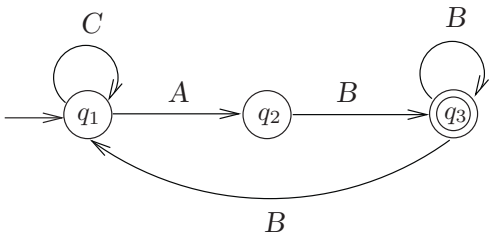


Figure 4.7: An example of an NBA.

one run in $\mathcal{A}$, namely $q_1\, q_1\, q_1\, q_1\, \ldots$, or in short, q_1^ω. Some other runs are $q_1\, q_2\, q_3^\omega$ for the word AB^ω, $(q_1\, q_1\, q_2\, q_3)^\omega$ for the word $(CABB)^\omega$, and $(q_1\, q_2\, q_3)^n\, q_1^\omega$ for the word $(ABB)^n C^\omega$ where $n \geqslant 0$.

The runs that go infinitely often through the accept state q_3 are accepting. For instance, $q_1\, q_2\, q_3^\omega$ and $(q_1\, q_1\, q_2\, q_3)^\omega$ are accepting runs. q_1^ω is not an accepting run as it never visits the accept state q_3, while runs of the form $(q_1\, q_2\, q_3)^n\, q_1^\omega$ are not accepting as they visit the accept state q_3 only finitely many times. The language accepted by this NBA is given by the ω-regular expression:

$$C^* AB \big(B^+ \;+\; BC^* AB\big)^\omega$$

■

Later in this chapter (page 198 ff.), NBAs are used for the verification of ω-regular properties – in the same vein as NFAs were exploited for the verification of regular safety properties. In that case, Σ is of the form $\Sigma = 2^{AP}$. As explained on page 159, propositional logic formulae are used as a shorthand notation for the transitions of such NBAs. For instance, if $AP = \{a, b\}$, then the label $a \vee b$ for an edge from q to p means that there are three transitions from q to p: one for the symbol $\{a\}$, one for the symbol $\{b\}$, and one for the symbol $\{a, b\}$.

Example 4.29. Infinitely Often Green

Let $AP = \{ green, red \}$ or any other set containing the proposition *green*. The language of words $\sigma = A_0 A_1 \ldots \in 2^{AP}$ satisfying the LT property "infinitely often *green*" is accepted by the NBA $\mathcal{A}$ depicted in Figure 4.8.

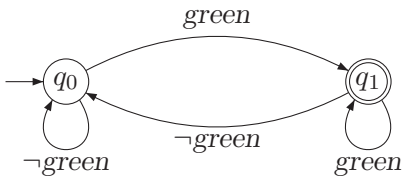


Figure 4.8: An NBA accepting "infinitely often *green*".

The automaton $\mathcal{A}$ is in the accept state q_1 if and only if the last input set of symbols (i.e., the last set A_i) contains the propositional symbol *green*. Therefore, $\mathcal{L}_\omega(\mathcal{A})$ is exactly the set of all infinite words $A_0 A_1 \ldots$ with infinitely many sets A_i with $green \in A_i$. For example, for the input word

$$\sigma \;=\; \{ green \}\,\{\,\}\,\{ green \}\,\{\,\}\,\{ green \}\,\{\,\}\ldots$$

we obtain the accepting run $q_0\, q_1\, q_0\, q_1 \ldots$. The same run $q_0\, q_1\, q_0\, q_1 \ldots$ is obtained for the word

$$\sigma' \;=\; (\{ green,\, red \}\,\{\,\}\,\{ green \}\,\{ red \})^\omega$$

or any other word $A_0\, A_1\, A_2 \ldots \in (2^{AP})^\omega$ with $green \in A_{2j}$ and $green \notin A_{2j+1}$ for all $j \geqslant 0$. ■

Example 4.30. Request Response

Many liveness properties are of the form

"Whenever some event a occurs,
some event b will eventually occur in the future"

For example, the property "once a request is provided, eventually a response occurs" is of this form. An associated NBA with propositions *req* and *resp* is indicated in Figure 4.9. It is assumed that $\{ req, resp \} \subseteq AP$, i.e., we assume the NBA to have alphabet 2^{AP} with AP containing at least *req* and *resp*. It is not difficult to see that this NBA accepts exactly those sequences in which each request is always eventually followed by a response. Note that an infinite trace in which only responses occur, but never a request (or finitely many requests) is also accepting. ■

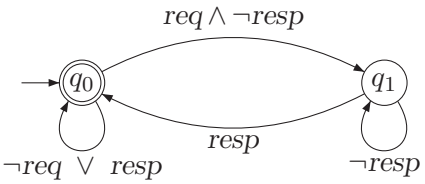


Figure 4.9: An NBA accepting "on each request, eventually a response is provided".

Remark 4.31. NBA and Regular Safety Properties

In Section 4.2, we have seen that there is a strong relationship between bad prefixes of regular safety properties and NFAs. In fact, there is also a strong relationship between NBAs and regular safety properties. This can be seen as follows. Let P_{safe} be a regular safety property over AP and $\mathcal{A} = (Q, 2^{AP}, \delta, Q_0, F)$ an NFA recognizing the language of all bad prefixes of P_{safe}. Each accept state $q_F \in F$ may be assumed to be a trapping state, i.e., $q_F \xrightarrow{A} q_F$ for all $A \subseteq AP$. This assumption is justified since each extension of a bad prefix is a bad prefix. (As a bad prefix contains a "bad" event that causes the violation of P_{safe}, each extension of this prefix contains this event.)

When interpreting $\mathcal{A}$ as an NBA, it accepts exactly the infinite words $\sigma \in (2^{AP})^\omega$ that violate P_{safe}, i.e.,

$$\mathcal{L}_\omega(\mathcal{A}) \;=\; (2^{AP})^\omega \setminus P_{safe}.$$

Here, it is important that $\mathcal{A}$ accepts all bad prefixes, and not just the minimal ones (see Exercise 4.18).

If $\mathcal{A}$ is a total deterministic automaton, i.e., in each state there is a single possible transition for each input symbol, then the NBA obtained by

$$\overline{\mathcal{A}} \;=\; \Big(Q, 2^{AP}, \delta, Q_0, Q \setminus F\Big)$$

accepts the language $\mathcal{L}_\omega\left(\overline{\mathcal{A}}\right) = P_{safe}$.

This is exemplified by means of a concrete case. Consider again the property "a red light phase should be immediately preceded by a yellow light phase" for a traffic light system. We have seen before (see Example 4.13 on page 161) that the bad prefixes of this safety property constitute a regular language and are accepted by the NFA shown in Figure 4.10. Note that this NFA is total. Applying the procedure described just above to this automaton yields the NBA depicted in Figure 4.11. It is easy to see that the infinite language accepted by this NBA consists exactly of all sequences of the form $\sigma = A_0\, A_1\, A_2 \ldots$ such that $\mathit{red} \in A_j$ implies $j > 0$ and $\mathit{yellow} \in A_{j-1}$. ■

The accepted languages of the NBA examples have so far been ω-regular. It is now shown

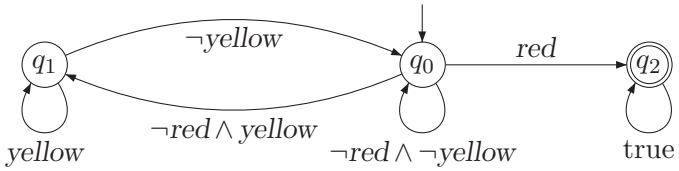


Figure 4.10: An NFA for the set of all bad prefixes of P_{safe}.

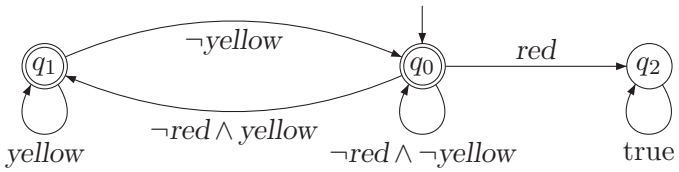


Figure 4.11: An NBA for the LT property "red should be preceded by yellow".

that this holds for any NBA. Moreover, it will be shown that any ω-regular language can be described by an NBA. Thus, *NBAs are as expressive as ω-regular languages.* This result is analogous to the fact that NFAs are as expressive as regular languages, and thus may act as an alternative formalism to describe regular languages. In the same spirit, NBAs are an alternative formalism for describing ω-regular languages. This is captured by the following theorem.

Theorem 4.32. NBAs and ω-Regular Languages

The class of languages accepted by NBAs agrees with the class of ω-regular languages.

The proof of Theorem 4.32 amounts to showing that (1) any ω-regular language is recognized by an NBA (see Corollary 4.38 on page 182) and (2) that the language $\mathcal{L}_\omega(\mathcal{A})$ accepted by the NBA $\mathcal{A}$ is ω-regular (see Lemma 4.39 on page 183).

We first consider the statement that ω-regular languages are contained in the class of languages recognized by an NBA. The proof of this fact is divided into the following three steps that rely on operations for NBAs to mimic the building blocks of ω-regular expressions:

(1) For any NBA $\mathcal{A}_1$ and $\mathcal{A}_2$ there exists an NBA accepting $\mathcal{L}_\omega(\mathcal{A}_1) \cup \mathcal{L}_\omega(\mathcal{A}_2)$.

(2) For any regular language $\mathcal{L}$ (of finite words) with $\varepsilon \notin \mathcal{L}$ there exists an NBA accepting $\mathcal{L}^\omega$.

(3) For regular language $\mathcal{L}$ and NBA $\mathcal{A}'$ there exists an NBA accepting $\mathcal{L}.\mathcal{L}_\omega(\mathcal{A}')$.

These three results that are proven below form the basic ingredients to construct an NBA for a given ω-regular expression $\mathsf{G} = \mathsf{E}_1.\mathsf{F}_1^\omega + \ldots + \mathsf{E}_n.\mathsf{F}_n^\omega$ with $\varepsilon \notin \mathsf{F}_i$. This works as follows. As an initial step, (2) is exploited to construct NBA $\mathcal{A}'_1, \ldots, \mathcal{A}'_n$ for the expressions $\mathsf{F}_1^\omega, \ldots, \mathsf{F}_n^\omega$. Then, (3) is used to construct an NBA for the expressions $\mathsf{E}_i.\mathsf{F}_i^\omega$, for $1 \leqslant i \leqslant n$. Finally, these NBA are combined using (1) to obtain an NBA for G.

Let us start with the union operator on two NBAs. Let $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, Q_{0,1}, F_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma, \delta_2, Q_{0,2}, F_2)$ be NBAs over the same alphabet Σ. Without loss of generality, it may be assumed that the state spaces Q_1 and Q_2 of $\mathcal{A}_1$ and $\mathcal{A}_2$ are disjoint, i.e., $Q_1 \cap Q_2 = \varnothing$. Let $\mathcal{A}_1 + \mathcal{A}_2$ be the NBA with the joint state spaces of $\mathcal{A}_1$ and $\mathcal{A}_2$, and with all transitions in $\mathcal{A}_1$ and $\mathcal{A}_2$. The initial states of $\mathcal{A}$ are the initial states of $\mathcal{A}_1$ and $\mathcal{A}_2$, and similarly, the accept states of $\mathcal{A}$ are the accept states of $\mathcal{A}_1$ and $\mathcal{A}_2$. That is,

$$\mathcal{A}_1 + \mathcal{A}_2 \;=\; (Q_1 \cup Q_2, \Sigma, \delta, Q_{0,1} \cup Q_{0,2}, F_1 \cup F_2)$$

where $\delta(q, A) = \delta_i(q, A)$ if $q \in Q_i$ for i=1, 2. Clearly, any accepting run in $\mathcal{A}_i$ is also an accepting run in $\mathcal{A}_1 + \mathcal{A}_2$, and vice versa, each accepting run in $\mathcal{A}_1 + \mathcal{A}_2$ is an accepting run in either $\mathcal{A}_1$ or $\mathcal{A}_2$. This yields $\mathcal{L}_\omega(\mathcal{A}_1 + \mathcal{A}_2) = \mathcal{L}_\omega(\mathcal{A}_1) \cup \mathcal{L}_\omega(\mathcal{A}_2)$. We thus obtain:

Lemma 4.33. Union Operator on NBA

For NBA $\mathcal{A}_1$ and $\mathcal{A}_2$ (both over the alphabet Σ) there exists an NBA $\mathcal{A}$ such that:

$$\mathcal{L}_\omega(\mathcal{A}) = \mathcal{L}_\omega(\mathcal{A}_1) \cup \mathcal{L}_\omega(\mathcal{A}_2) \quad \text{and} \quad |\mathcal{A}| = \mathcal{O}(|\mathcal{A}_1| + |\mathcal{A}_2|).$$

Now consider (2). We will show that for any regular language $\mathcal{L} \subseteq \Sigma^*$ there exists an NBA over the alphabet Σ that accepts the ω-regular language $\mathcal{L}^\omega$. To do so, we start with a representation of $\mathcal{L}$ by an NFA $\mathcal{A}$.

Lemma 4.34. ω-Operator for NFA

For each NFA $\mathcal{A}$ with $\varepsilon \notin \mathcal{L}(\mathcal{A})$ there exists an NBA $\mathcal{A}'$ such that

$$\mathcal{L}_\omega(\mathcal{A}') = \mathcal{L}(\mathcal{A})^\omega \quad \text{and} \quad |\mathcal{A}'| = \mathcal{O}(|\mathcal{A}|).$$

Proof: Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be an NFA with $\varepsilon \notin \mathcal{L}(\mathcal{A})$. Without loss of generality, we may assume that all initial states in $\mathcal{A}$ have no incoming transitions and are not accepting.

Any $\mathcal{A}$ that does not possess this property, can be modified into an equivalent NFA as follows. Add a new initial (nonaccept) state q_{new} to Q with the transitions $q_{new} \xrightarrow{A} q$ if and only if $q_0 \xrightarrow{A} q$ for some initial state $q_0 \in Q_0$. All other transitions, as well as the accept states, remain unchanged. The state q_{new} is the single initial state of the modified NFA, is not accept, and, clearly, has no incoming transitions. This modification neither affects the accepted language nor the asymptotic size of $\mathcal{A}$.

In the sequel, we assume that $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ is an NFA such that the states in Q_0 do not have any incoming transitions and $Q_0 \cap F = \varnothing$. We now construct an NBA $\mathcal{A}' = (Q, \Sigma, \delta', Q_0', F')$ with $\mathcal{L}_\omega(\mathcal{A}') = \mathcal{L}(\mathcal{A})^\omega$. The basic idea of the construction of $\mathcal{A}'$ is to add for any transition in $\mathcal{A}$ that leads to an accept state new transitions leading to the initial states of $\mathcal{A}$. Formally, the transition relation δ' in the NBA $\mathcal{A}'$ is given by

$$\delta'(q, A) \;=\; \begin{cases} \delta(q, A) & \text{if } \delta(q, A) \cap F = \varnothing \\ \delta(q, A) \cup Q_0 & \text{otherwise.} \end{cases}$$

The initial states in the NBA $\mathcal{A}'$ agree with the initial states in $\mathcal{A}$, i.e., $Q_0' = Q_0$. These are also the accept states in $\mathcal{A}'$, i.e., $F' = Q_0$.

Let us check that $\mathcal{L}_\omega(\mathcal{A}') = \mathcal{L}(\mathcal{A})^\omega$. This is proven as follows.

$\subseteq$: Assume that $\sigma \in \mathcal{L}_\omega(\mathcal{A}')$ and let $q_0 q_1 q_2 \ldots$ be an accepting run for σ in $\mathcal{A}'$. Hence, $q_i \in F' = Q_0$ for infinitely many indices i. Let $i_0 = 0 < i_1 < i_2 < \ldots$ be the strictly increasing sequence of natural numbers with $\{ q_{i_0}, q_{i_1}, q_{i_2}, \ldots \} \subseteq Q_0$ and $q_j \notin Q_0$ for all $j \in \mathbb{N} \setminus \{ i_0, i_1, i_2, \ldots \}$. The word σ can be divided into infinitely many nonempty finite subwords $w_i \in \Sigma^*$ yielding $\sigma = w_1 w_2 w_3 \ldots$ such that $q_{i_k} \in \delta'^*(q_{i_{k-1}}, w_k)$ for all $k \geqslant 1$. (The extension of δ' to a function $\delta'^* : Q \times \Sigma^* \to 2^Q$ is as for an NFA, see page 154.) By definition of $\mathcal{A}'$ and since the states $q_{i_k} \in Q_0$ do not have any predecessor in $\mathcal{A}$, we get $\delta^*(q_{i_{k-1}}, w_k) \cap F \neq \varnothing$. This yields $w_k \in \mathcal{L}(\mathcal{A})$ for all $k \geqslant 1$, which gives us $\sigma \in \mathcal{L}(\mathcal{A})^\omega$.

$\supseteq$: Let $\sigma = w_1 w_2 w_3 \ldots \in \Sigma^\omega$ such that $w_k \in \mathcal{L}(\mathcal{A})$ for all $k \geqslant 1$. For each k, we choose an accepting run $q_0^k q_1^k \ldots q_{n_k}^k$ for w_k in $\mathcal{A}$. Hence, $q_0^k \in Q_0$ and $q_{n_k}^k \in F$. By definition of $\mathcal{A}'$, we have $q_0^{k+1} \in \delta'^*(q_0^k, w_k)$ for all $k \geqslant 1$. Thus,

$$q_0^1 \ldots q_{n_1-1}^1 q_0^2 \ldots q_{n_2-1}^2 q_0^3 \ldots q_{n_3-1}^3 \cdots$$

is an accepting run for σ in $\mathcal{A}'$. Hence, $\sigma \in \mathcal{L}_\omega(\mathcal{A}')$. ∎

Example 4.35. ω-Operator for an NFA

Consider the NFA depicted in the left upper part of Figure 4.12. It accepts the language $A^* B$. In order to obtain an NBA recognizing $(A^* B)^\omega$, we first apply the transformation

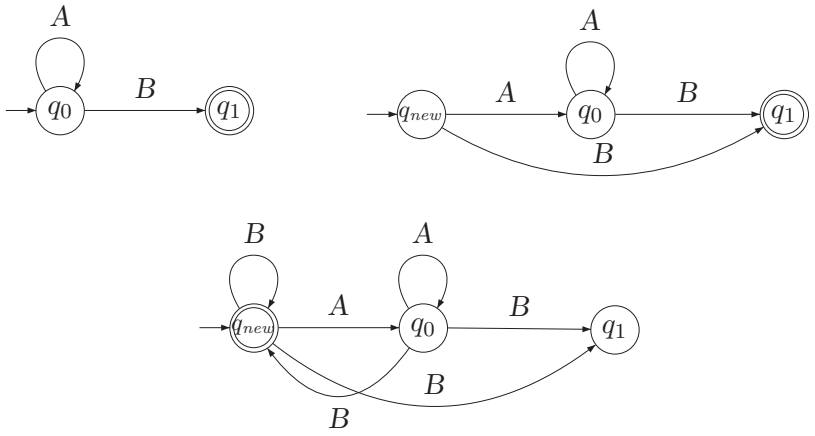


Figure 4.12: From an NFA accepting A^*B to an NBA accepting $(A^*B)^\omega$.

as described in the proof of Lemma 4.34 to remove initial states that have an incoming transition. This yields the NFA depicted in the right upper part of Figure 4.12. This automaton can now be used to apply the construction of the required NBA as detailed in the proof of Lemma 4.34. This yields the NBA depicted in the lower part of Figure 4.12. ■

It remains to provide a construction for task (3) above. Assume that we have NFA $\mathcal{A}$ for the regular language $\mathcal{L}(\mathcal{A})$ and a given NBA $\mathcal{A}'$ at our disposal. The proof of the following lemma will describe a procedure to obtain an NBA for the ω-language $\mathcal{L}(\mathcal{A}).\mathcal{L}_\omega(\mathcal{A}')$.

Lemma 4.36. Concatenation of an NFA and an NBA

For NFA $\mathcal{A}$ and NBA $\mathcal{A}'$ (both over the alphabet Σ), there exists an NBA $\mathcal{A}''$ with

$$\mathcal{L}_\omega(\mathcal{A}'') = \mathcal{L}(\mathcal{A}).\mathcal{L}_\omega(\mathcal{A}') \quad \textit{and} \quad |\mathcal{A}''| = \mathcal{O}(|\mathcal{A}| + |\mathcal{A}'|).$$

Proof: Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be an NFA and $\mathcal{A}' = (Q', \Sigma, \delta', Q_0', F')$ an NBA with $Q \cap Q' = \varnothing$. Let $\mathcal{A}'' = (Q'', \Sigma, \delta'', Q_0'', F'')$ be the following NBA. The state space is $Q'' = Q \cup Q'$. The set of initial and accept states are given by

$$Q_0'' = \begin{cases} Q_0 & \text{if } Q_0 \cap F = \varnothing \\ Q_0 \cup Q_0' & \text{otherwise,} \end{cases}$$

and $F'' = F'$ (set of accept state in the NBA $\mathcal{A}'$). The transition function δ'' is given by

$$\delta''(q,A) \;=\; \begin{cases} \delta(q,A) & \text{if } q \in Q \text{ and } \delta(q,A) \cap F = \varnothing \\ \delta(q,A) \cup Q'_0 & \text{if } q \in Q \text{ and } \delta(q,A) \cap F \neq \varnothing \\ \delta'(q,A) & \text{if } q \in Q' \end{cases}$$

It is now easy to check that $\mathcal{A}''$ fulfills the desired conditions. ■

Example 4.37. Concatenation of an NFA and an NBA

Consider the NFA $\mathcal{A}$ and the NBA $\mathcal{A}'$ depicted in the left and right upper part of Figure 4.13, respectively. We have $\mathcal{L}(\mathcal{A}) = (AB)^*$ and $\mathcal{L}(\mathcal{A}') = (A{+}B)^*BA^\omega$. Applying the transformation as described in Lemma 4.36 yields the NBA depicted in the lower part of Figure 4.13. It is not difficult to assess that this NBA accepts indeed the concatenated language $(AB)^*(A{+}B)^*BA^\omega$. ■

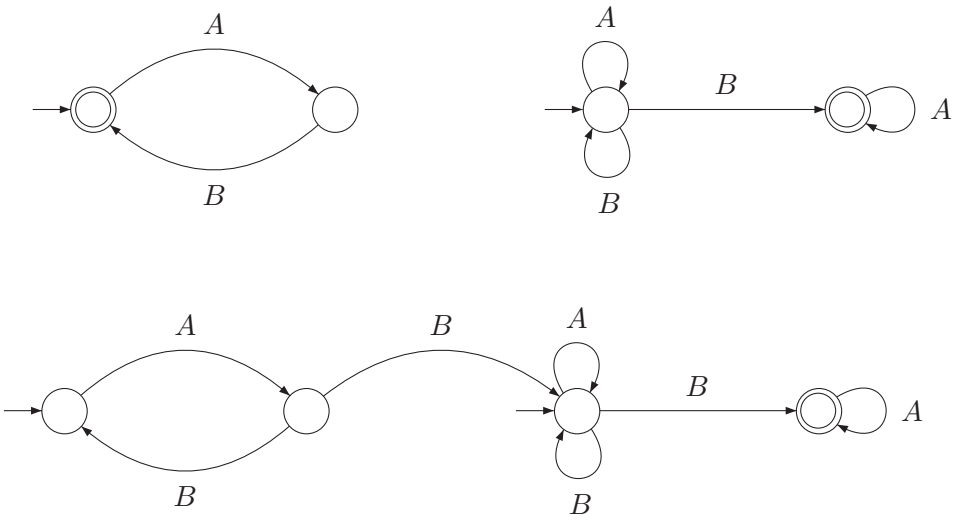


Figure 4.13: Concatenation of an NFA and an NBA.

By Lemmas 4.33, 4.34, and 4.36 we obtain the first part for the proof of Theorem 4.32:

Corollary 4.38. NBA for ω-Regular Languages

For any ω-regular language $\mathcal{L}$ there exists an NBA $\mathcal{A}$ with $\mathcal{L}_\omega(\mathcal{A}) = \mathcal{L}$.

The proof of the following lemma shows that the languages accepted by NBA can be described by an ω-regular expression.

Lemma 4.39. NBAs Accept ω-Regular Languages

For each NBA $\mathcal{A}$, the accepted language $\mathcal{L}_\omega(\mathcal{A})$ is ω-regular.

Proof: Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be an NBA. For states $q, p \in Q$, let $\mathcal{A}_{qp}$ be the NFA $(Q, \Sigma, \delta, \{q\}, \{p\})$. Then, $\mathcal{A}_{qp}$ recognizes the regular language consisting of all finite words $w \in \Sigma^*$ that have a run in $\mathcal{A}$ leading from q to p, that is,

$$\mathcal{L}_{qp} \stackrel{\text{def}}{=} \mathcal{L}(\mathcal{A}_{qp}) = \{\, w \in \Sigma^* \mid p \in \delta^*(q, w) \,\}.$$

Consider a word $\sigma \in \mathcal{L}_\omega(\mathcal{A})$ and an accepting run $q_0 q_1 \ldots$ for σ in $\mathcal{A}$. Some accept state $q \in F$ appears infinitely often in this run. Hence, we may split σ into nonempty finite subwords $w_0, w_1, w_2, w_3, \ldots \in \Sigma^*$ such that $w_0 \in \mathcal{L}_{q_0 q}$ and $w_k \in \mathcal{L}_{qq}$ for all $k \geqslant 1$ and

$$\sigma \;=\; \underbrace{w_0}_{\in \mathcal{L}_{q_0 q}} \underbrace{w_1}_{\in \mathcal{L}_{qq}} \underbrace{w_2}_{\in \mathcal{L}_{qq}} \underbrace{w_3}_{\in \mathcal{L}_{qq}} \cdots\cdots$$

On the other hand, any infinite word σ which has the form $\sigma = w_0 w_1 w_2 \ldots$ where the w_k's are nonempty finite words with $w_0 \in \mathcal{L}_{q_0 q}$ for some initial state $q_0 \in Q_0$ and $\{w_1, w_2, w_3, \ldots\} \subseteq \mathcal{L}_{qq}$ for some accept state $q \in F$ has an accepting run in $\mathcal{A}$. This yields

$$\sigma \in \mathcal{L}_\omega(\mathcal{A}) \quad \text{if and only if} \quad \exists q_0 \in Q_0 \; \exists q \in F.\; \sigma \in \mathcal{L}_{q_0 q} \, (\mathcal{L}_{qq} \setminus \{\varepsilon\})^\omega \,.$$

Hence, $\mathcal{L}_\omega(\mathcal{A})$ agrees with the language

$$\bigcup_{q_0 \in Q_0, q \in F} \mathcal{L}_{q_0 q} . \, (\mathcal{L}_{qq} \setminus \{\varepsilon\})^\omega$$

which is ω-regular. ∎

Example 4.40. From NBA to ω-Regular Expression

For the NBA $\mathcal{A}$ shown in Figure 4.7 on page 175, a corresponding ω-regular expression is obtained by

$$\mathcal{L}_{q_1 q_3} . (\mathcal{L}_{q_3 q_3} \setminus \{\varepsilon\})^\omega$$

since q_1 is the unique initial state and q_3 the unique accept state in $\mathcal{A}$. The regular language $\mathcal{L}_{q_3 q_3} \setminus \{\varepsilon\}$ can be described by the expression $(B^+ + BC^*AB)^+$, while $\mathcal{L}_{q_1 q_3}$ is given by $(C^*AB(B^* + BC^*AB)^*B)^* C^*AB$. Hence, $\mathcal{L}_\omega(\mathcal{A}) = \mathcal{L}_\omega(\mathsf{G})$ where G is the ω-regular expression:

$$\mathsf{G} \;=\; \underbrace{(C^*AB(B^* + BC^*AB)^*B)^* C^*AB}_{\mathcal{L}_{q_1 q_3}} \underbrace{((B^+ + BC^*AB)^+)^\omega}_{(\mathcal{L}_{q_3 q_3} \setminus \{\varepsilon\})^\omega}.$$

The thus obtained expression G can be simplified to the equivalent expression:

$$C^*AB(B^+ + BC^*AB)^\omega.$$

■

The above lemma, together with Corollary 4.38, completes the proof of Theorem 4.32 stating the equivalence of the class of languages accepted by NBAs and the class of all ω-regular languages. Thus, NBAs and ω-regular languages are equally expressive.

A fundamental question for any type of automata model is the question whether for a given automaton $\mathcal{A}$ the accepted language is empty. For nondeterministic Büchi automata, an analysis of the underlying directed graph by means of standard graph algorithms is sufficient, as we will show now.

Lemma 4.41. Criterion for the Nonemptiness of an NBA
Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be an NBA. Then, the following two statements are equivalent:

(a) $\mathcal{L}_\omega(\mathcal{A}) \neq \varnothing$,

(b) There exists a reachable accept state q that belongs to a cycle in $\mathcal{A}$. Formally,

$$\exists q_0 \in Q_0 \ \exists q \in F \ \exists w \in \Sigma^* \ \exists v \in \Sigma^+. \ \ q \in \delta^*(q_0, w) \cap \delta^*(q, v).$$

Proof: (a) $\Longrightarrow$ (b): Let $\sigma = A_0\, A_1\, A_2 \ldots \in \mathcal{L}_\omega(\mathcal{A})$ and let $q_0\, q_1\, q_2 \ldots$ be an accepting run for σ in $\mathcal{A}$. Let $q \in F$ be an accept state with $q = q_i$ for infinitely many indices i. Let i and j be two indices with $0 \leqslant i < j$ and $q_i = q_j = q$. We consider the finite words $w = A_0 A_1 \ldots A_{i-1}$ and $v = A_i A_{i+1} \ldots A_{j-1}$ and obtain $q = q_i \in \delta^*(q_0, w)$ and $q = q_j \in \delta^*(q_i, v) = \delta^*(q, v)$. Hence, (b) holds.

(b) $\Longrightarrow$ (a): Let q_0, q, w, v be as in statement (b). Then, the infinite word $\sigma = wv^\omega$ has a run of the form $q_0 \ldots q \ldots q \ldots q \ldots$ that infinitely often contains q. Since $q \in F$ this run is accepting which yields $\sigma \in \mathcal{L}_\omega(\mathcal{A})$, and thus, $\mathcal{L}_\omega(\mathcal{A}) \neq \varnothing$. ■

By the above lemma, the emptiness problem for NBAs can be solved by means of graph algorithms that explore all reachable states and check whether they belong to a cycle. One possibility to do so is to calculate the strongly connected components of the underlying directed graph of $\mathcal{A}$ and to check whether there is at least one nontrivial strongly con-

nected component[3] that is reachable (from at least one of the initial states) and contains an accept state. Since the strongly connected components of a (finite) directed graph can be computed in time linear in the number of states and edges, the time complexity of this algorithm for the emptiness check of NBA $\mathcal{A}$ is linear in the size of $\mathcal{A}$. An alternative algorithm that also runs in time linear in the size of $\mathcal{A}$, but avoids the explicit computation of the strongly connected components, can be derived from the results stated in Section 4.4.2.

Theorem 4.42. Checking Emptiness for NBA

The emptiness problem for NBA $\mathcal{A}$ can be solved in time $\mathcal{O}(|\mathcal{A}|)$.

Since NBAs serve as a formalism for ω-regular languages, we may identify two Büchi automata for the same language:

Definition 4.43. Equivalence of NBA

Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two NBAs with the same alphabet. $\mathcal{A}_1$ and $\mathcal{A}_2$ are called *equivalent*, denoted $\mathcal{A}_1 \equiv \mathcal{A}_2$, if $\mathcal{L}_\omega(\mathcal{A}_1) = \mathcal{L}_\omega(\mathcal{A}_2)$. ■

Example 4.44. Equivalent NBA

As for other finite automata, equivalent NBAs can have a totally different structure. For example, consider the NBA shown in Figure 4.14 over the alphabet 2^{AP} where $AP = \{a, b\}$. Both NBAs represent the liveness property "infinitely often a and infinitely often b", and thus, they are equivalent. ■

Remark 4.45. NFA vs. NBA Equivalence

It is interesting to consider more carefully the relationship between the notions of equivalence of NFAs and NBAs. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two automata that we can regard as either NFA or as NBA. To distinguish the equivalence symbol $\equiv$ for NFAs from that for NBAs we will write in this example $\equiv_{NFA}$ to denote the equivalence relation for NFA and the symbol $\equiv_{NBA}$ to denote the equivalence relation for NBA, i.e., $\mathcal{A}_1 \equiv_{NFA} \mathcal{A}_2$ iff $\mathcal{L}(\mathcal{A}_1) = \mathcal{L}(\mathcal{A}_2)$ and $\mathcal{A}_1 \equiv_{NBA} \mathcal{A}_2$ iff $\mathcal{L}_\omega(\mathcal{A}_1) = \mathcal{L}_\omega(\mathcal{A}_2)$.

1. If $\mathcal{A}_1$ and $\mathcal{A}_2$ accept the same finite words, i.e., $\mathcal{A}_1 \equiv_{NFA} \mathcal{A}_2$, then this does not mean that they also accept the same infinite words. The following two automata examples show this:

[3] A strongly connected component is nontrivial if it contains at least one edge. In fact, any cycle is contained in a nontrivial strongly connected component, and vice versa, any nontrivial strongly connected component contains a cycle that goes through all its states.

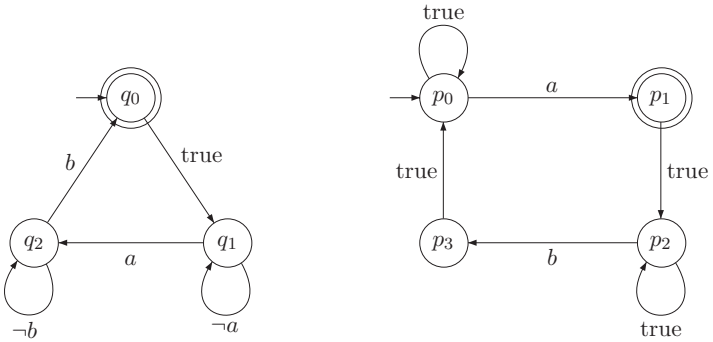
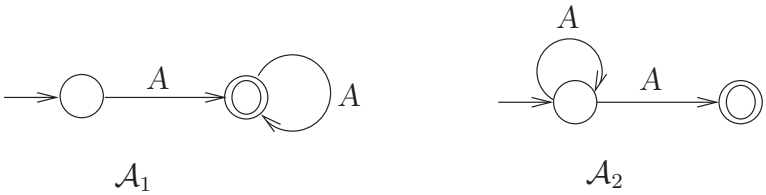
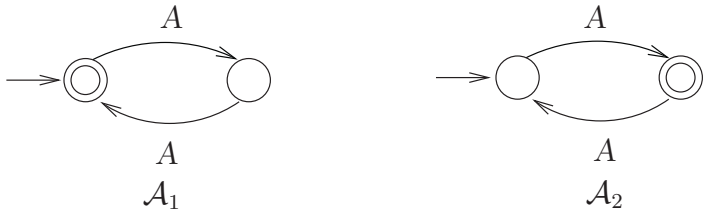


Figure 4.14: Two equivalent NBA.



We have $\mathcal{L}(\mathcal{A}_1) = \mathcal{L}(\mathcal{A}_2) = \{ A^n \mid n \geqslant 1 \}$, but $\mathcal{L}_\omega(\mathcal{A}_1) = \{ A^\omega \}$ and $\mathcal{L}_\omega(\mathcal{A}_2) = \varnothing$. Thus, $\mathcal{A}_1 \equiv_{NFA} \mathcal{A}_2$ but $\mathcal{A}_1 \not\equiv_{NBA} \mathcal{A}_2$.

2. If $\mathcal{A}_1$ and $\mathcal{A}_2$ accept the same infinite words, i.e., $\mathcal{A}_1 \equiv_{NBA} \mathcal{A}_2$, then one might expect that they would also accept the same finite words. This also turns out not to be true. The following example shows this:



We have $\mathcal{L}_\omega(\mathcal{A}_1) = \mathcal{L}_\omega(\mathcal{A}_2) = \{ A^\omega \}$, but $\mathcal{L}(\mathcal{A}_1) = \{ A^{2n} \mid n \geqslant 0 \}$ and $\mathcal{L}(\mathcal{A}_2) = \{ A^{2n+1} \mid n \geqslant 0 \}$.

3. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are both deterministic (see Definition 4.9 on page 156), then $\mathcal{A}_1 \equiv_{NFA} \mathcal{A}_2$ implies $\mathcal{A}_1 \equiv_{NBA} \mathcal{A}_2$. The reverse is, however, not true, as illustrated by the previous example.

■

For technical reasons, it is often comfortable to assume for an NBA that for each state q and for each input symbol A, there is a possible transition. Such an NBA can be seen to be nonblocking since no matter how the nondeterministic choices are resolved, the automaton cannot fail to consume the current input symbol.

Definition 4.46. Nonblocking NBA

Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be an NBA. $\mathcal{A}$ is called *nonblocking* if $\delta(q, A) \neq \varnothing$ for all states q and all symbols $A \in \Sigma$. ■

Note that for a given nonblocking NBA $\mathcal{A}$ and input word $\sigma \in \Sigma^\omega$, there is at least one (infinite) possibly nonaccepting run for σ in $\mathcal{A}$. The following remark demonstrates that it is not a restriction to assume a nonblocking NBA.

Remark 4.47. Nonblocking NBA

For each NBA $\mathcal{A}$ there exists a nonblocking NBA $\mathit{trap}(\mathcal{A})$ with $|\mathit{trap}(\mathcal{A})| = \mathcal{O}(|\mathcal{A}|)$ and $\mathcal{A} \equiv \mathit{trap}(\mathcal{A})$.

Let us see how such a nonblocking NBA can be derived from $\mathcal{A}$. NBA $\mathit{trap}(\mathcal{A})$ is obtained from $\mathcal{A}$ by inserting a nonaccept trapping state q_{trap} equipped with a self-loop for each symbol in the alphabet Σ. For every symbol $A \in \Sigma$ for which state q in $\mathcal{A}$ does not have an outgoing transition, a transition to q_{trap} is added. Formally, if $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$, then $\mathit{trap}(\mathcal{A}) = (Q', \Sigma, \delta', Q_0', F')$ as follows. Here, $Q' = Q \cup \{ q_{\mathit{trap}} \}$ where q_{trap} is a new state (not in Q) that will be reached in $\mathcal{A}'$ whenever $\mathcal{A}$ does not have a corresponding transition. Formally, the transition relation δ' of $\mathit{trap}(\mathcal{A})$ is defined by:

$$\delta'(q, A) = \begin{cases} \delta(q, A) & \text{if } q \in Q \text{ and } \delta(q, A) \neq \varnothing \\ \{ q_{\mathit{trap}} \} & \text{otherwise} \end{cases}$$

The initial and accept states are unchanged, i.e., $Q_0' = Q_0$ and $F' = F$. By definition, $\mathit{trap}(\mathcal{A})$ is nonblocking and – since the new trap state is nonaccepting – is equivalent to $\mathcal{A}$. ■

We conclude this subsection on automata over infinite words with a few more comments on Büchi automata and ω-regular languages. We first study the subclass of deterministic Büchi automata (Section 4.3.3 below) and then in Section 4.3.4 the class of NBA with a more general acceptance condition consisting of several acceptance sets that have to be visited infinitely often.

4.3.3 Deterministic Büchi Automata

An important difference between finite-state automata and Büchi automata is the expressive power of deterministic and nondeterministic automata. While for languages of finite words, DFAs and NFAs have the same expressiveness, this does not hold for Büchi automata.

The definition of a deterministic Büchi automaton is the same as for a DFA:

Definition 4.48. Deterministic Büchi Automaton (DBA)
Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be an NBA. $\mathcal{A}$ is called *deterministic*, if

$$|Q_0| \leqslant 1 \quad \text{and} \quad |\delta(q, A)| \leqslant 1$$

for all $q \in Q$ and $A \in \Sigma$. $\mathcal{A}$ is *total* if $|Q_0| = 1$ and $|\delta(q, A)| = 1$ for all $q \in Q$ and $A \in \Sigma$. ■

Obviously, the behavior of a DBA for a given input word is deterministic: either eventually the DBA will get stuck in some state as it fails to consume the current input symbol or there is a unique (infinite) run for the given input word. Total DBAs rule out the first alternative and ensure the existence of a unique run for every input word $\sigma \in \Sigma^\omega$.

Example 4.49. DBA for LT Properties
Figure 4.15 shows the DBA $\mathcal{A}'$ (on the left) and the NBA $\mathcal{A}$ (on the right) over the alphabet $\Sigma = 2^{AP}$ where $AP = \{ a, b \}$. These automata are equivalent since both represent the LT property "always b and infinitely often a". Let δ be the transition function of $\mathcal{A}$ and δ'

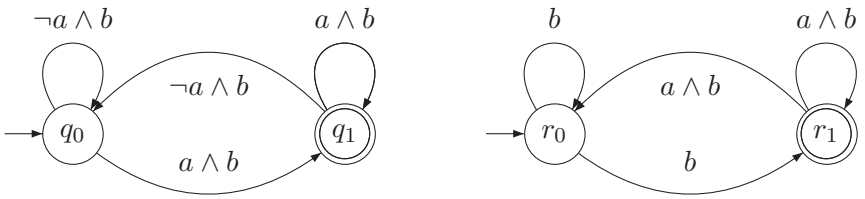


Figure 4.15: An equivalent DBA $\mathcal{A}'$ (left) and NBA $\mathcal{A}$ (right).

the transition function of $\mathcal{A}'$. The NBA $\mathcal{A}$ is not deterministic since for all input symbols containing a b, there is the possibility to move to either state r_0 or r_1. The DBA $\mathcal{A}'$ is

deterministic. Note that both $\mathcal{A}'$ and $\mathcal{A}$ are blocking, e.g., any state is blocking on an input symbol containing $\neg b$. ■

As for deterministic finite automata, the usual notation is $q' = \delta(q, A)$ (instead of $\{q'\} = \delta(q, A)$) and $\delta(q, A) = \perp$ (undefined), if $\delta(q, A) = \varnothing$. Thus, the transition relation of a DBA is understood as partial function $\delta : Q \times \Sigma \to Q$. Total DBAs are often written in the form $(Q, \Sigma, \delta, q_0, F)$ where q_0 is the unique initial state and δ is viewed as a total function $Q \times \Sigma \to Q$. Since DBA can always be extended by a nonaccept trapping state without changing the accepting language, it can be assumed without restriction that the transition relation is total. For instance, Figure 4.16 shows an equivalent total DBA for the DBA $\mathcal{A}'$ in Figure 4.15 that is obtained by adding such a trapping state.

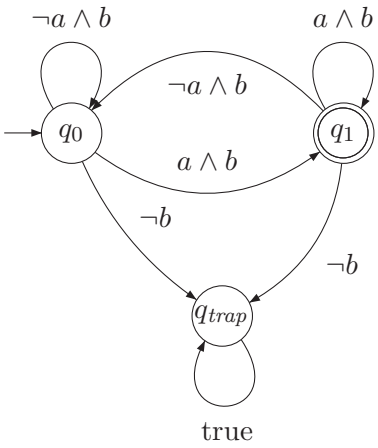


Figure 4.16: A total DBA for "always b and infinitely often a".

The transition function δ of a total DBA can be expanded to a total function $\delta^* : Q \times \Sigma^* \to Q$ in the obvious way; see also page 157 for the transition function of a total DFA. That is, let $\delta^*(q, \varepsilon) = q$, $\delta^*(q, A) = \delta(q, A)$ and

$$\delta^*(q, A_1 A_2 \ldots A_n) = \delta^*(\, \delta(q, A_1),\, A_2 \ldots A_n).$$

Then, for every infinite word $\sigma = A_0\, A_1\, A_2\, \ldots \in \Sigma^\omega$, the run $q_0\, q_1\, q_2 \ldots$ in $\mathcal{A}$ belonging to σ is given by $q_{i+1} = \delta^*(q_0, A_0 \ldots A_i)$ for all $i \geqslant 0$, where q_0 is the unique initial state of $\mathcal{A}$. In particular, for total DBA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ the accepted language is given by

$$\mathcal{L}_\omega(\mathcal{A}) \;=\; \{\; A_0\, A_1\, A_2 \ldots \in \Sigma^\omega \mid\; \delta^*(q_0, A_0 \ldots A_i) \in F \text{ for infinitely many } i \;\}$$

As we have seen before, NFAs are as expressive as deterministic ones. However, *NBAs are more expressive than deterministic ones.* That is, there do exist NBA for which there

does not exist an equivalent deterministic one. Stated differently, while any ω-language accepted by a DBA is ω-regular, there do exist ω-regular languages for which there does not exist a DBA accepting it. An example of such ω-regular language is the language given by the expression $(A+B)^*B^\omega$.

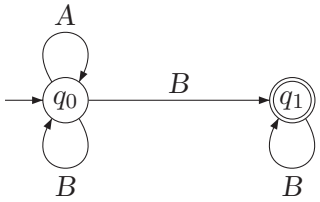


Figure 4.17: NBA for the ω-regular expression $(A + B)^*B^\omega$.

In fact, the language $\mathcal{L}_\omega(\,(A+B)^*B^\omega\,)$ is accepted by a rather simple NBA, shown in Figure 4.17. The idea of this NBA is that given an input word $\sigma = wB^\omega$ where $w \in \{A, B\}^*$ the automaton may stay in q_0 and guess nondeterministically when the suffix consisting of B's starts and then moves to the accept state q_1. This behavior, however, cannot be simulated by a DBA as formally shown in the following theorem.

Theorem 4.50. NBAs are More Powerful than DBAs

There does not exist a DBA $\mathcal{A}$ such that $\mathcal{L}_\omega(\mathcal{A}) = \mathcal{L}_\omega((A + B)^*B^\omega)$.

Proof: By contradiction. Assume that $\mathcal{L}_\omega((A + B)^*B^\omega) = \mathcal{L}_\omega(\mathcal{A})$ for some DBA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ with $\Sigma = \{\, A, B \,\}$. Note that since $\mathcal{A}$ is deterministic, δ^* can be considered as a function of type $Q \times \Sigma^* \to Q$.

Since the word $\sigma_1 = B^\omega$ belongs to $\mathcal{L}_\omega((A + B)^*B^\omega) = \mathcal{L}_\omega(\mathcal{A})$, there exists an accepting state $q_1 \in F$ and a $n_1 \in \mathbb{N}_{\geqslant 1}$ such that

$$(1) \qquad \delta^*\,(q_0, B^{n_1}) \;=\; q_1 \;\in F \quad .$$

(Since $\mathcal{A}$ is deterministic, q_1 is uniquely determined.) Now consider the word $\sigma_2 = B^{n_1}AB^\omega \;\in \mathcal{L}_\omega((A + B)^*B^\omega) = \mathcal{L}_\omega(\mathcal{A})$. Since σ_2 is accepted by $\mathcal{A}$, there exists an accepting state $q_2 \in F$ and $n_2 \in \mathbb{N}_{\geqslant 1}$, such that

$$(2) \qquad \delta^*\,(q_0, B^{n_1}AB^{n_2}) \;=\; q_2 \in F \quad .$$

The word $B^{n_1}AB^{n_2}AB^\omega$ is in $\mathcal{L}_\omega((A + B)^*B^\omega)$, and, thus, is accepted by $\mathcal{A}$. So, there is an accepting state $q_3 \in F$ and $n_3 \in \mathbb{N}_{\geqslant 1}$ with

$$(3) \qquad \delta^*\left(q_0, B^{n_1}AB^{n_2}AB^{n_3}\right) \;=\; q_3 \in F.$$

Continuing this process, we obtain a sequence $n_1, n_2, n_3, \ldots$ of natural numbers $\geqslant 1$ and a sequence $q_1, q_2, q_3, \ldots$ of accepting states such that

$$\delta^*\left(q_0, B^{n_1}AB^{n_2}A \ldots B^{n_{i-1}}AB^{n_i}\right) \;=\; q_i \in F, \quad i \geqslant 1 \ldots$$

Since there are only finitely many states, there exist $i < j$ such that

$$\delta^*\left(q_0, B^{n_1}A \ldots AB^{n_i}\right) \;=\; \delta^*\left(q_0, B^{n_1}A \ldots AB^{n_i} \ldots AB^{n_j}\right)$$

Thus $\mathcal{A}$ has an accepting run on

$$B^{n_1}A \ldots AB^{n_i}\left(AB^{n_{i+1}}A \ldots AB^{n_j}\right)^{\omega}.$$

But this word has infinitely many occurrences of A, and thus does not belong to $\mathcal{L}_\omega((A+B)^*B^\omega)$. Contradiction. ∎

Example 4.51. The Need for Nondeterminism

In Examples 4.29 and 4.30, we provided DBAs for LT properties. To represent liveness properties of the form "eventually forever", the concept of nondeterminism is, however, necessary. Consider the property "eventually forever a", where a is some atomic proposition. Let $\{\,a\,\} = AP$, i.e., $2^{AP} = \{A, B\}$ where $A = \{\}$ and $B = \{a\}$. Then, the linear-time property "eventually forever a" is given by the ω-regular expression

$$(A + B)^*B^\omega \;=\; (\{\} + \{a\})^*\{a\}^\omega.$$

By Theorem 4.50, there is no DBA for "eventually forever a". On the other hand, this property can be described by the NBA $\mathcal{A}$ depicted in Figure 4.18. (Note that state q_2 could be omitted, as there is no accepting run that starts in q_2.) Intuitively, $\mathcal{A}$ nondeterministically decides (by means of an omniscient oracle) from which instant the proposition a is continuously true. This behavior cannot be mimicked by a DBA. ∎

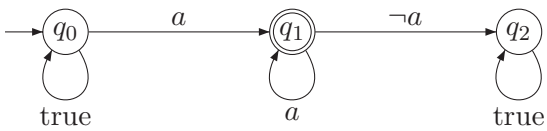


Figure 4.18: An NBA accepting "eventually forever a".

The reader might wonder why the powerset construction known for finite automata (see page 157) fails for Büchi automata. The deterministic automaton $\mathcal{A}_{det}$ obtained through

the powerset construction allows simulating the given nondeterministic automaton $\mathcal{A}$ by keeping track of the set Q' of states that are reachable in $\mathcal{A}$ for any finite prefix of the given input word. (This set Q' is a state in $\mathcal{A}_{det}$.) What is problematic here is the acceptance condition: while for NFAs the information whether an accept state is reachable is sufficient, for infinite words we need one single run that passes an accept state infinitely often. The latter is not equivalent to the requirement that a state Q' with $Q' \cap F \neq \varnothing$ is visited infinitely often, since there might be infinitely many possibilities (runs) to enter F at different time points, i.e., for different prefixes of the input word. This, in fact, is the case for the NBA in Figure 4.17. For the input word $\sigma = ABABA\ldots = (AB)^\omega$, the automaton in Figure 4.17 can enter the accept state q_1 after the second, fourth, sixth, etc., symbol by staying in q_0 for the first $2n-1$ symbols and moving with the nth B to state q_1 (for $n = 1, 2, \ldots$). Thus, at infinitely many positions there is the possibility to enter F, although there is no run that visits q_1 infinitely often, since whenever q_1 has been entered the automaton $\mathcal{A}$ rejects when reading the next A. In fact, the powerset construction applied to the NBA $\mathcal{A}$ in Figure 4.17 yields a DBA $\mathcal{A}_{det}$ with two reachable states (namely $\{q_0\}$ and $\{q_0, q_1\}$) for the language consisting of all infinite words with infinitely many B's, but not for the language given by $(A + B)^*B^\omega$.

Another example that illustrates why the powerset construction fails for Büchi automata is provided in Exercise 4.16 (page 225).

4.3.4 Generalized Büchi Automata

In several applications, other ω-automata types are useful as automata models for ω-regular languages. In fact, there are several variants of ω-automata that are equally expressive as nondeterministic Büchi automata, although they use more general acceptance conditions than the Büchi acceptance condition "visit infinitely often the acceptance set F". For some of these ω-automata types, the deterministic version has the full power of ω-regular languages. These automata types are not relevant for the remaining chapters of this monograph and will not be treated here. [4]

For the purposes of this monograph, it suffices to consider a slight variant of nondeterministic Büchi automata, called *generalized* nondeterministic Büchi automata, or GNBA for short. The difference between an NBA and a GNBA is that the acceptance condition for a GNBA requires to visit several sets $F_1, \ldots, F_k$ infinitely often. Formally, the syntax of a GNBA is as for an NBA, except that the acceptance condition is a set $\mathcal{F}$ consisting of *finitely many acceptance sets* $F_1, \ldots, F_k$ with $F_i \subseteq Q$. That is, if Q is the state space of the automaton then the acceptance condition of a GNBA is an element $\mathcal{F}$ of 2^{2^Q}. Recall

[4] In Chapter 10, deterministic Rabin automata will be used for representing ω-regular properties.

that for an NBA, it is an element $F \in 2^Q$. The accepted language of a GNBA $\mathcal{G}$ consists of all infinite words which have an infinite run in $\mathcal{G}$ that visits *all* sets $F_i \in \mathcal{F}$ infinitely often. Thus, the acceptance criterion in a generalized Büchi automaton can be understood as the conjunction of a number of Büchi acceptance conditions.

Definition 4.52. Generalized NBA (GNBA)

A *generalized NBA* is a tuple $\mathcal{G} = (Q, \Sigma, \delta, Q_0, \mathcal{F})$ where Q, Σ, δ, Q_0 are defined as for an NBA (see Definition 4.27 on page 174) and $\mathcal{F}$ is a (possibly empty) subset of 2^Q.

The elements $F \in \mathcal{F}$ are called *acceptance sets*. Runs in a GNBA are defined as for an NBA. That is, a run in $\mathcal{G}$ for the infinite word $A_0\, A_1 \ldots \in \Sigma^\omega$ is an infinite state sequence $q_0\, q_1\, q_2 \ldots \in Q^\omega$ such that $q_0 \in Q_0$ and $q_{i+1} \in \delta(q_i, A_i)$ for all $i \geqslant 0$.

The infinite run $q_0\, q_1\, q_2 \ldots$ is called *accepting* if

$$\forall\, F \in \mathcal{F}.\ \left(\overset{\infty}{\exists}\ j \in \mathbb{N}.\ q_j \in F \right).$$

The accepted language of $\mathcal{G}$ is:

$$\mathcal{L}_\omega(\mathcal{G}) \ = \ \{\, \sigma \in \Sigma^\omega \mid \text{there exists an accepting run for } \sigma \text{ in } \mathcal{G} \,\}.$$

■

Equivalence of GNBAs and the size of a GNBA are defined as for NBAs. Thus, GNBA $\mathcal{G}$ and $\mathcal{G}'$ are equivalent if $\mathcal{L}_\omega(\mathcal{G}) = \mathcal{L}_\omega(\mathcal{G}')$. The size of GNBA $\mathcal{G}$, denoted $|\mathcal{G}|$, equals the number of states and transitions in $\mathcal{G}$.

Example 4.53. GNBA

Figure 4.19 shows a GNBA $\mathcal{G}$ over the alphabet 2^{AP} where $AP = \{\, \mathit{crit}_1, \mathit{crit}_2 \,\}$ with the acceptance sets $F_1 = \{\, q_1 \,\}$ and $F_2 = \{\, q_2 \,\}$. That is, $\mathcal{F} = \{\{\, q_1 \,\}, \{\, q_2 \,\}\}$. The accepted language is the LT property P_{live} consisting of all infinite words $A_0\, A_1\, A_2 \ldots \in (2^{AP})^\omega$ such that the atomic propositions crit_1 and crit_2 hold infinitely often (possibly at different positions), i.e.,

$$\overset{\infty}{\exists}\ j \geqslant 0.\, \mathit{crit}_1 \in A_j \quad \text{and} \quad \overset{\infty}{\exists}\ j \geqslant 0.\, \mathit{crit}_2 \in A_j.$$

Thus, P_{live} formalizes the property "both processes are infinitely often in their critical section". Let us justify that indeed $\mathcal{L}_\omega(\mathcal{G}) = P_{\mathit{live}}$. This goes as follows.

"$\subseteq$": Each accepting run has to pass infinitely often through the edges (labeled with crit_1 or crit_2) leading to the states q_1 and q_2. Thus, in every accepted word $\sigma = A_0 A_1 A_2 \ldots \in$

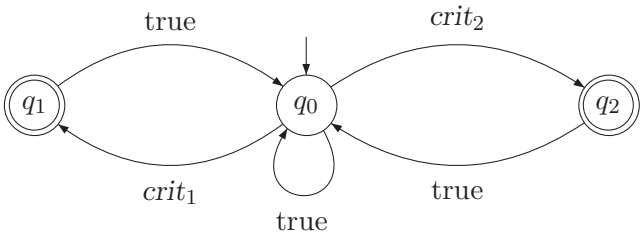


Figure 4.19: GNBA for "infinitely often processes 1 and 2 are in their critical section".

$\mathcal{L}_\omega(\mathcal{G})$ the atomic propositions $crit_1$ and $crit_2$ occur infinitely often as elements of the sets $A_i \in 2^{AP}$. Thus, $\sigma \in P_{live}$.

"$\supseteq$": Let $\sigma = A_0\, A_1\, A_2 \ldots \in P_{live}$. Since both propositions $crit_1$ and $crit_2$ occur infinitely often in the symbols A_i, the GNBA $\mathcal{G}$ can behave for the input word σ as follows. $\mathcal{G}$ remains in state q_0 until the first input symbol A_i with $crit_1 \in A_i$ appears. The automaton then moves to state q_1. From there, $\mathcal{G}$ consumes the next input symbol A_{i+1} and returns to q_0. It then waits in q_0 until a symbol A_j with $crit_2 \in A_j$ occurs, in which case the automaton moves to state q_2 for the symbol A_j and returns to q_0 on the next symbol A_{j+1}. Now the whole procedure restarts, i.e., $\mathcal{G}$ stays in q_0 while reading the symbols $A_{j+1}, \ldots, A_{\ell-1}$ and moves to q_1 as soon as the current input symbol A_ℓ contains $crit_1$. And so on. In this way, $\mathcal{G}$ generates an accepting run of the form

$$q_0^{k_1}\, q_1\, q_0^{k_2}\, q_2\, q_0^{k_3}\, q_1\, q_0^{k_4} q_2\, q_0^{k_5} \cdots$$

for the input word σ. These considerations show that $P_{live} \subseteq \mathcal{L}_\omega(\mathcal{G})$. ■

Remark 4.54. No Acceptance Set

The set $\mathcal{F}$ of acceptance sets of a GNBA may be empty. If $\mathcal{F} = \varnothing$ then $\sigma \in \mathcal{L}_\omega(\mathcal{G})$ if and only if there exists an infinite run for σ in $\mathcal{G}$. We like to stress the difference with NBA with an empty set of accepting states. For an NBA $\mathcal{A} = (Q, \Sigma, \delta, Q_0, \varnothing)$ there are *no* accepting runs. Therefore, the language $\mathcal{L}_\omega(\mathcal{A})$ is empty. Contrary to that, *every* infinite run of a GNBA $\mathcal{G} = (Q, \Sigma, \delta, Q_0, \varnothing)$ is accepting.

In fact, every GNBA $\mathcal{G}$ is equivalent to a GNBA $\mathcal{G}'$ having *at least one* acceptance set. This is due to the fact that the state space Q can always be added to the set $\mathcal{F}$ of the acceptance sets without affecting the accepted language of the GNBA. Formally, for GNBA $\mathcal{G} = (Q, \Sigma, \delta, Q_0, \mathcal{F})$ let GNBA $\mathcal{G}' = (Q, \Sigma, \delta, Q_0, \mathcal{F} \cup \{Q\})$. Then it easily follows that: $\mathcal{L}_\omega(\mathcal{G}) = \mathcal{L}_\omega(\mathcal{G}')$. ■

Remark 4.55. Nonblocking GNBA

As for NBAs, each GNBA $\mathcal{G}$ can be replaced with an equivalent GNBA $\mathcal{G}'$, in which all possible behaviors for a given infinite input word yield an infinite run. Such a GNBA $\mathcal{G}'$ can be constructed by inserting a nonaccept trapping state, as we did for NBA in the remark on page 187. ■

Obviously, every NBA can be understood as a GNBA with exactly one acceptance set. Conversely, every GNBA can be transformed into an equivalent NBA:

Theorem 4.56. From GNBA to NBA

For each GNBA $\mathcal{G}$ there exists an NBA $\mathcal{A}$ with $\mathcal{L}_\omega(\mathcal{G}) = \mathcal{L}_\omega(\mathcal{A})$ and $|\mathcal{A}| = \mathcal{O}(|\mathcal{G}| \cdot |\mathcal{F}|)$ where $\mathcal{F}$ denotes the set of acceptance sets in $\mathcal{G}$.

Proof: Let $\mathcal{G} = (Q, \Sigma, \delta, Q_0, \mathcal{F})$ be a GNBA. According to the remark on page 194, we may assume without loss of generality that $\mathcal{F} \neq \varnothing$. Let $\mathcal{F} = \{F_1, \ldots, F_k\}$ where $k \geqslant 1$. The basic idea of the construction of $\mathcal{A}$ is to create k copies of $\mathcal{G}$ such that the acceptance set F_i of the ith copy is connected to the corresponding states of the $(i{+}1)$th copy. The accepting

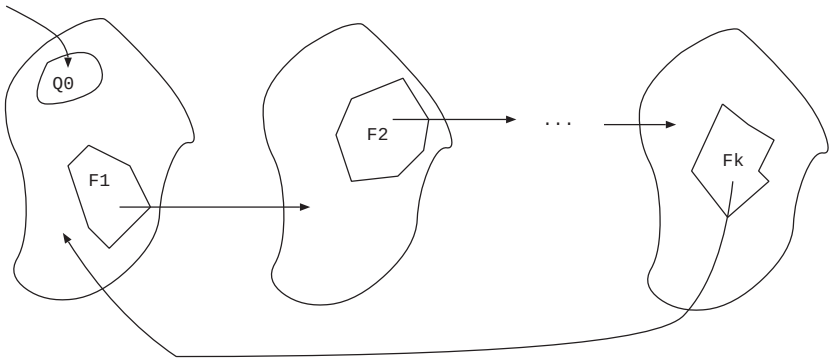


Figure 4.20: Idea for transforming a GNBA into an NBA.

condition for $\mathcal{A}$ consists of the requirement that an accepting state of the first copy is visited infinitely often. This ensures that all other accepting sets F_i of the k copies are visited infinitely often too, see Figure 4.20 on page 195. Formally, let $\mathcal{A} = (Q', \Sigma, \delta', Q'_0, F')$ where:

$$Q' = Q \times \{1, \ldots, k\},$$

$$Q'_0 = Q_0 \times \{1\} = \{\langle q_0, 1\rangle \mid q_0 \in Q_0\}, \text{ and}$$

$$F' = F_1 \times \{1\} = \{\langle q_F, 1\rangle \mid q_F \in F_1\}.$$

The transition function δ' is given by

$$\delta'(\langle q, i\rangle, A) = \begin{cases} \{\, \langle q', i\rangle \mid q' \in \delta(q, A) \,\} & \text{if } q \notin F_i \\ \{\, \langle q', i{+}1\rangle \mid q' \in \delta(q, A) \,\} & \text{otherwise.} \end{cases}$$

We thereby identify $\langle q, k{+}1\rangle$ and $\langle q, 1\rangle$. It is not difficult to check that $\mathcal{A}$ can be constructed in time and space $\mathcal{O}(|\mathcal{G}| \cdot |\mathcal{F}|)$ where $|\mathcal{F}| = k$ is the number of acceptance sets in $\mathcal{G}$. The fact $\mathcal{L}_\omega(\mathcal{G}) = \mathcal{L}_\omega(\mathcal{A})$ can be seen as follows.

$\supseteq$: For a run of $\mathcal{A}$ to be accepting it has to visit some state $\langle q, 1\rangle$ infinitely often, where $q \in F_1$. As soon as a run reaches $\langle q, 1\rangle$, the NBA $\mathcal{A}$ moves to the second copy. From the second copy the next copy can be reached by visiting $\langle q', 2\rangle$ with $q' \in F_2$. NBA $\mathcal{A}$ can only return to $\langle q, 1\rangle$ if it goes through all k copies. This is only possible if it reaches an accept state in each copy since that is the only opportunity to move to the next copy. So, for a run to visit $\langle q, 1\rangle$ infinitely often it has to visit some accept state in each copy infinitely often.

$\subseteq$: By a similar reasoning it can be deduced that every word in $\mathcal{L}_\omega(\mathcal{G})$ is also accepting in $\mathcal{A}$. ■

Example 4.57. Transformation of a GNBA into an NBA

Consider the GNBA $\mathcal{G}$ described in Example 4.53 on page 193. The construction indicated in the proof of Theorem 4.56 provides an NBA consisting of two copies (as there are two accept sets) of $\mathcal{G}$, see Figure 4.21. For example,

$$\delta'(\langle q_0, 1\rangle, \{\, \mathit{crit}_1 \,\}) = \{\, \langle q_0, 1\rangle, \langle q_1, 1\rangle \,\},$$

since $q_0 \notin F_1 = \{\, q_1 \,\}$ and $\delta'(\langle q_2, 2\rangle, A) = \{\, \langle q_0, 1\rangle \,\}$, since $F_2 = \{\, q_2 \,\}$. Thereby, $A \subseteq \{\, \mathit{crit}_1, \mathit{crit}_2 \,\}$ is arbitrary. ■

Any NBA can be considered as a GNBA by simply replacing the acceptance set F of the NBA with the singleton set $\mathcal{F} = \{\, F \,\}$ for the corresponding GNBA. Using this fact, together with the result that NBAs are equally expressive as ω-regular languages (see Theorem 4.32 on page 178), we obtain by Theorem 4.56:

Corollary 4.58. GNBA and ω-Regular Languages

The class of languages accepted by GNBAs agrees with the class of ω-regular languages.

As we have seen before, ω-regular languages are closed under union. This is immediate from the definition of ω-regular expressions and can also simply be proven by means of

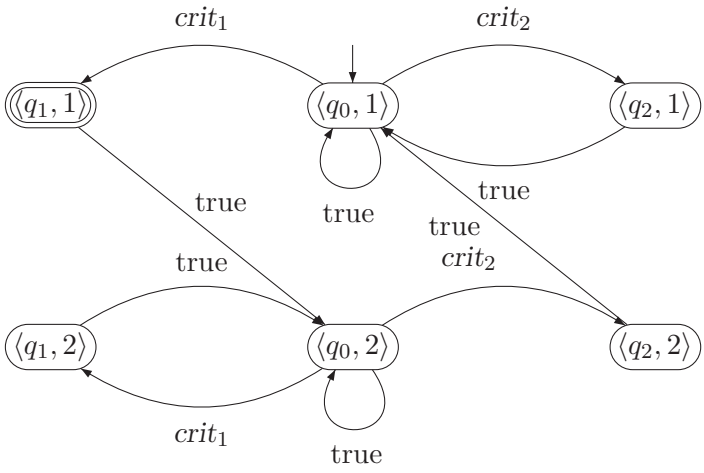


Figure 4.21: Example for the transformation of a GNBA into an equivalent NBA.

NBA representations (see Lemma 4.33 on page 179). We now use GNBAs to show that ω-regular languages are closed under intersection too.

Lemma 4.59. Intersection of GNBA

For GNBA $\mathcal{G}_1$ and $\mathcal{G}_2$ (both over the alphabet Σ), there exists a GNBA $\mathcal{G}$ with

$$\mathcal{L}_\omega(\mathcal{G}) = \mathcal{L}_\omega(\mathcal{G}_1) \cap \mathcal{L}_\omega(\mathcal{G}_2) \quad \textit{and} \quad |\mathcal{G}| = \mathcal{O}(|\mathcal{G}_1| \cdot |\mathcal{G}_2|).$$

Proof: Let $\mathcal{G}_1 = (Q_1, \Sigma, \delta_1, Q_{0,1}, \mathcal{F}_1)$ and $\mathcal{G}_2 = (Q_2, \Sigma, \delta_2, Q_{0,2}, \mathcal{F}_2)$ where without loss of generality $Q_1 \cap Q_2 = \varnothing$. Let $\mathcal{G}$ be the GNBA that results from $\mathcal{G}_1$ and $\mathcal{G}_2$ by a synchronous product construction (as for NFA) and "lifts" the acceptance sets $F \in \mathcal{F}_1 \cup \mathcal{F}_2$ to acceptance sets in $\mathcal{G}$. Formally,

$$\mathcal{G} = \mathcal{G}_1 \otimes \mathcal{G}_2 = (Q_1 \times Q_2, \Sigma, \delta, Q_{0,1} \times Q_{0,2}, \mathcal{F})$$

where the transition relation δ is defined by the rule

$$\frac{q_1 \xrightarrow{A}_1 q_1' \ \wedge \ q_2 \xrightarrow{A}_2 q_2'}{(q_1, q_2) \xrightarrow{A} (q_1', q_2')}.$$

The acceptance condition in $\mathcal{G}$ is given by

$$\mathcal{F} = \{F_1 \times Q_2 \mid F_1 \in \mathcal{F}_1\} \cup \{Q_1 \times F_2 \mid F_2 \in \mathcal{F}_2\}.$$

It is now easy to verify that $\mathcal{G}$ has the desired properties. ■

The same result also holds for union, that is, given two GNBAs $\mathcal{G}_1$ and $\mathcal{G}_2$ with the same alphabet Σ there is a GNBA $\mathcal{G}$ with $\mathcal{L}_\omega(\mathcal{G}) = \mathcal{L}_\omega(\mathcal{G}_1) \cup \mathcal{L}_\omega(\mathcal{G}_2)$ and $|\mathcal{G}| = \mathcal{O}(|\mathcal{G}_1| + |\mathcal{G}_2|)$. The argument is the same as for NBA (Lemma 4.33): we simply may take the disjoint union of the two GNBAs and decide nondeterministically which of them is chosen to "scan" the given input word.

Since GNBA yield an alternative characterization of ω-regular languages, we obtain by Lemma 4.59:

Corollary 4.60. Intersection of ω-Regular Languages

If $\mathcal{L}_1$ and $\mathcal{L}_2$ are ω-regular languages over the alphabet Σ, then so is $\mathcal{L}_1 \cap \mathcal{L}_2$.

4.4 Model-Checking ω-Regular Properties

The examples provided in Section 4.3.2 (see page 176 ff.) illustrated that NBAs yield a simple formalism for ω-regular properties. We now address the question how the automata-based approach for verifying regular safety properties can be generalized for the verification of ω-regular properties.

The starting point is a finite transition system $TS = (S, Act, \rightarrow, I, AP, L)$ without terminal states and an ω-regular property P. The aim is to check algorithmically whether $TS \models P$. As we did for regular safety properties, the verification algorithm we present now attempts to show that $TS \not\models P$ by providing a counterexample, i.e., a path π in TS with $trace(\pi) \notin P$. (If no such path exists, then P holds for TS.) For this, we assume that we are given an automata-representation of the "bad traces" by means of an NBA $\mathcal{A}$ for the complement property $\overline{P} = (2^{AP})^\omega \setminus P$. The goal is then to check whether $Traces(TS) \cap \mathcal{L}_\omega(\mathcal{A}) \neq \varnothing$. Note that:

$$
\begin{aligned}
& Traces(TS) \cap \mathcal{L}_\omega(\mathcal{A}) \neq \varnothing \\
\text{if and only if} \quad & Traces(TS) \cap \overline{P} \neq \varnothing \\
\text{if and only if} \quad & Traces(TS) \cap (2^{AP})^\omega \setminus P \neq \varnothing \\
\text{if and only if} \quad & Traces(TS) \not\subseteq P \\
\text{if and only if} \quad & TS \not\models P.
\end{aligned}
$$

The reader should notice the similarities with regular safety property checking. In that case, we started with an NFA for the bad prefixes for the given safety property P_{safe} (the bad behaviors). This can be seen as an automata-representation of the complement property $\overline{P_{safe}}$, see Remark 4.31 on page 177. The goal was then to find a finite, initial path fragment in TS that yields a trace accepted by the NFA, i.e., a bad prefix for P_{safe}.

Let us now address the problem to check whether $\mathit{Traces}(TS) \cap \mathcal{L}_\omega(\mathcal{A}) \neq \varnothing$. For this, we can follow the same pattern as for regular safety properties and construct the product $TS \otimes \mathcal{A}$ which combines paths in TS with the runs in $\mathcal{A}$. We then perform a graph analysis in $TS \otimes \mathcal{A}$ to check whether there is a path that visits an accept state of $\mathcal{A}$ infinitely often, which then yields a counterexample and proves $TS \not\models P$. If no such path in the product exists, i.e., if accept states can be visited at most finitely many times on all paths in the product, then all runs for the traces in TS are nonaccepting, and hence $\mathit{Traces}(TS) \cap \mathcal{L}_\omega(\mathcal{A}) = \varnothing$ and thus $TS \models P$

In the sequel, we will explain these ideas in more detail. For doing so, we first introduce the notion of a persistence property. This is a simple type of LT property that will serve to formalize the condition stating that *accept states are only visited finitely many times*. The problem of verifying ω-regular properties is then shown to be reducible to the persistence checking problem. Recall that the problem of verifying regular safety properties is reducible to the invariant checking problem, see Section 4.2.

4.4.1 Persistence Properties and Product

Persistence properties are special types of liveness properties that assert that from some moment on a certain state condition Φ holds continuously. Stated in other words, $\neg\Phi$ is required to hold at most finitely many times. As for invariants, we assume a representation of Φ by a propositional logic formula over AP.

Definition 4.61. Persistence Property

A *persistence property* over AP is an LT property $P_{pers} \subseteq (2^{AP})^\omega$ "eventually forever Φ" for some propositional logic formula Φ over AP. Formally,

$$P_{pers} \;=\; \Big\{ A_0 A_1 A_2 \ldots \in (2^{AP})^\omega \mid \overset{\infty}{\forall}\, j.\; A_j \models \Phi \Big\}$$

where $\overset{\infty}{\forall}\, j$ is short for $\exists i \geqslant 0.\ \forall j \geqslant i$. Formula Φ is called a persistence (or state) condition of P_{pers}. ■

Intuitively, a persistence property "eventually forever Φ" ensures the tenacity of the state

property given by the persistence condition Φ. One may say that Φ is an invariant after a while; i.e., from a certain point on all states satisfy Φ. The formula "eventually forever Φ" is true for a path if and only if almost all, i.e., all except for finitely many, states satisfy the proposition Φ.

Our goal is now to show that the question whether $\mathit{Traces}(TS) \cap \mathcal{L}_\omega(\mathcal{A}) = \varnothing$ holds can be reduced to the question whether a certain persistence property holds in the product of TS and $\mathcal{A}$. The formal definition of the product $TS \otimes \mathcal{A}$ is exactly the same as for an NFA. For completeness, we recall the definition here:

Definition 4.62. Product of Transition System and NBA

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system without terminal states and $\mathcal{A} = (Q, 2^{AP}, \delta, Q_0, F)$ a nonblocking NBA. Then, $TS \otimes \mathcal{A}$ is the following transition system:

$$TS \otimes \mathcal{A} \;=\; (S \times Q, Act, \rightarrow', I', AP', L')$$

where $\rightarrow'$ is the smallest relation defined by the rule

$$\frac{s \xrightarrow{\alpha} t \;\wedge\; q \xrightarrow{L(t)} p}{\langle s, q\rangle \xrightarrow{\alpha}' \langle t, p\rangle}$$

and where

- $I' = \{\, \langle s_0, q\rangle \mid s_0 \in I \;\wedge\; \exists q_0 \in Q_0.\; q_0 \xrightarrow{L(s_0)} q \,\}$,
- $AP' = Q$ and $L' : S \times Q \to 2^Q$ is given by $L'(\langle s, q\rangle) \;=\; \{\, q \,\}$.

Furthermore, let $P_{pers(\mathcal{A})}$ be the persistence property over $AP' = Q$ given by

"eventually forever $\neg F$"

where $\neg F$ denotes the propositional formula $\bigwedge\limits_{q \in Q} \neg q$ over $AP' = Q$. ■

We now turn to the formal proof that the automata-based approach for checking ω-regular properties relies on checking a persistence property for the product transition system:

Theorem 4.63. Verification of ω-Regular Properties

Let TS be a finite transition system without terminal states over AP and let P be an ω-regular property over AP. Furthermore, let $\mathcal{A}$ be a nonblocking NBA with the alphabet 2^{AP} and $\mathcal{L}_\omega(\mathcal{A}) = (2^{AP})^\omega \setminus P$. Then, the following statements are equivalent:

(a) $TS \models P$

(b) $\mathit{Traces}(TS) \cap \mathcal{L}_\omega(\mathcal{A}) = \varnothing$

(c) $TS \otimes \mathcal{A} \models P_{\mathit{pers}(\mathcal{A})}$

Proof: Let $TS = (S, Act, \to, I, AP, L)$ and $\mathcal{A} = (Q, 2^{AP}, \delta, Q_0, F)$. The equivalence of (a) and (b) was shown on page 198. Let us now check the equivalence of (b) and (c). For this we show

$$\mathit{Traces}(TS) \cap \mathcal{L}_\omega(\mathcal{A}) \neq \varnothing \text{ if and only if } TS \otimes \mathcal{A} \not\models P_{\mathit{pers}(\mathcal{A})}.$$

"$\Leftarrow$": Assume $TS \otimes \mathcal{A} \not\models P_{\mathit{pers}(\mathcal{A})}$. Let $\pi' = \langle s_0, q_1\rangle \langle s_1, q_2\rangle \ldots$ be a path in $TS \otimes \mathcal{A}$ such that

$$\pi' \not\models P_{\mathit{pers}(\mathcal{A})}.$$

Then there are infinitely many indices i with $q_i \in F$. The projection of π' to the states in TS yields a path $\pi = s_0\, s_1\, s_2 \ldots$ in TS.

Let $q_0 \in Q_0$ be an initial state of $\mathcal{A}$ such that $q_0 \xrightarrow{L(s_0)} q_1$. Such a state q_0 exists, since $\langle s_0, q_1\rangle$ is an initial state of $TS \otimes \mathcal{A}$. The state sequence $q_0\, q_1\, q_2 \ldots$ is a run in $\mathcal{A}$ for the word

$$\mathit{trace}(\pi) = L(s_0)\, L(s_1)\, L(s_2) \ldots \in \mathit{Traces}(TS).$$

Since there are infinitely many i with $q_i \in F$, the run $q_0\, q_1\, q_2 \ldots$ is accepting. Hence,

$$\mathit{trace}(\pi) \in \mathcal{L}_\omega(\mathcal{A}).$$

This yields $\mathit{trace}(\pi) \in \mathit{Traces}(TS) \cap \mathcal{L}_\omega(\mathcal{A})$, and thus $\mathit{Traces}(TS) \cap \mathcal{L}_\omega(\mathcal{A}) \neq \varnothing$.

"$\Rightarrow$": Assume that $\mathit{Traces}(TS) \cap \mathcal{L}_\omega(\mathcal{A}) \neq \varnothing$. Then there exists a path in TS, say $\pi = s_0\, s_1\, s_2 \ldots$ with

$$\mathit{trace}(\pi) = L(s_0)\, L(s_1)\, L(s_2) \ldots \in \mathcal{L}_\omega(\mathcal{A}).$$

Let $q_0\, q_1\, q_2 \ldots$ be an accepting run in $\mathcal{A}$ for $\mathit{trace}(\pi)$. Then

$$q_0 \in Q_0 \quad \text{and} \quad q_i \xrightarrow{L(s_i)} q_{i+1} \quad \text{for all } i \geqslant 0.$$

Furthermore, $q_i \in F$ for infinitely many indices i. Thus, we can combine π and the run $q_0\, q_1 \ldots$ to obtain a path in the product

$$\pi' = \langle s_0, q_1\rangle \langle s_1, q_2\rangle \ldots \in \mathit{Paths}(TS \otimes \mathcal{A}).$$

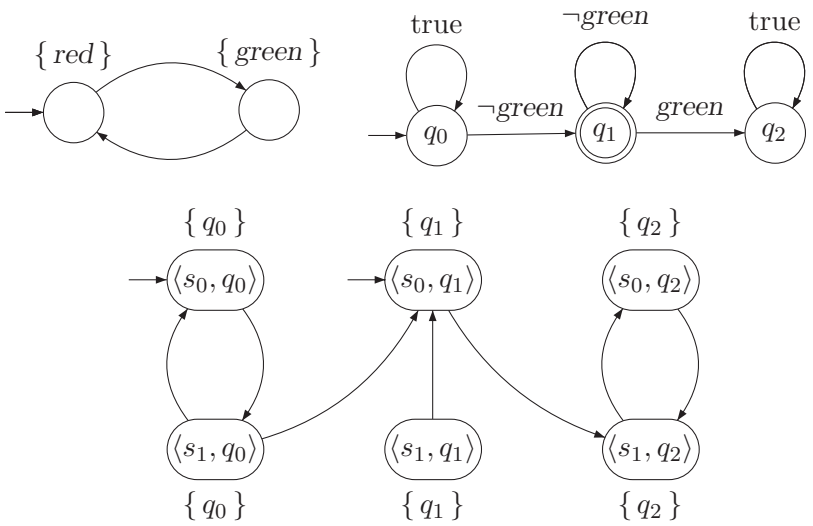


Figure 4.22: A simple traffic light (upper left), an NBA corresponding to P_{pers} (upper right), and their product (below).

Since $q_i \in F$ for infinitely many i, we have $\pi' \not\models P_{pers(\mathcal{A})}$. This yields $TS \otimes \mathcal{A} \not\models P_{pers(\mathcal{A})}$. ■

Example 4.64. Checking a Persistence Property

Consider a simple traffic light as one typically encounters at pedestrian crossings. It only has two possible modes: red or green. Assume that the traffic light is initially red, and alternates between red and green, see the transition system *PedTrLight* depicted in the upper left part of Figure 4.22. The ω-regular property P to be checked is "infinitely often green". The complement property $\overline{P}$ thus is "eventually always not green". The NBA depicted in the upper right part of Figure 4.22 accepts $\overline{P}$.

To check the validity of P, we first construct the product automaton $PedTrLight \otimes \mathcal{A}$, see the lower part of Figure 4.22. Note that the state $\langle s_1, q_1 \rangle$ is unreachable and could be omitted. Let $P_{pers(\mathcal{A})}$ = "eventually forever $\neg q_1$". From the lower part of Figure 4.22 one can immediately infer that there is no run of the product automaton that goes infinitely often through a state of the form $\langle \cdot, q_1 \rangle$. That is, transition system *PedTrLight* and NBA $\mathcal{A}$ do not have any trace in common. Thus we conclude:

$$PedTrLight \otimes \mathcal{A} \models \text{"eventually forever"}\ \neg q_1$$

and consequently (as expected):

$$PedTrLight \models \text{"infinitely often green"}.$$

As a slight alternative, we now consider a pedestrian traffic light that may automatically switch off to save energy. Assume, for simplicity, that the light may only switch off when it is red for some undefined amount of time. Clearly, this traffic light cannot guarantee the validity of $\overline{P}$ = "eventually forever $\neg$*green*" as it exhibits a run that (possibly after a while) alternates between red and off infinitely often. This can be formally shown as follows. First, we construct the product automaton, see Figure 4.23 (lower part). For instance, the path $\langle s_0, q_0\rangle\,(\langle s_2, q_1\rangle\,\langle s_0, q_1\rangle)^\omega$ goes infinitely often through the accept state q_1 of $\mathcal{A}$ and generates the trace

$$\{\,red\,\}\,\varnothing\,\{\,red\,\}\,\varnothing\,\{\,red\,\}\,\varnothing\,\ldots\ldots$$

That is, *Traces*(*PedTrLight'*) $\cap\ \mathcal{L}_\omega(\mathcal{A}) \neq \varnothing$, and thus

$$\mathit{PedTrLight'} \otimes \mathcal{A}\ \not\models\ \text{"eventually forever"}\ \neg q_1$$

and thus

$$\mathit{PedTrLight'} \not\models \text{"infinitely often green"}.$$

More concretely, the path $\pi = s_0\,s_1\,s_0\,s_1\ldots$ generating the trace

$$\mathit{trace}(\pi) = \{\,red\,\}\,\varnothing\,\{\,red\,\}\,\varnothing\,\{\,red\,\}\,\varnothing\ldots$$

has an accepting run $q_0\,(q_1)^\omega$ in $\mathcal{A}$. ■

According to Theorem 4.63, the problem of checking an arbitrary ω-regular property can be solved with algorithms that check a simple type of ω-regular liveness property, namely persistence properties. An algorithm for the latter will be provided in the following section where the transition system under consideration results from the product of the original transition system and an NBA for the undesired behaviors.

4.4.2 Nested Depth-First Search

The next problem that we need to tackle is how to establish whether for a given finite transition system *TS*:

$$TS \not\models P_{pers}$$

where P_{pers} is a persistence property. Let Φ be the underlying propositional formula that specifies the state condition which has to hold "eventually forever".

The following result shows that answering the question "does $TS \not\models P_{pers}$ hold?" amounts to checking whether *TS* contains a reachable state violating Φ that is on a cycle in *TS*.

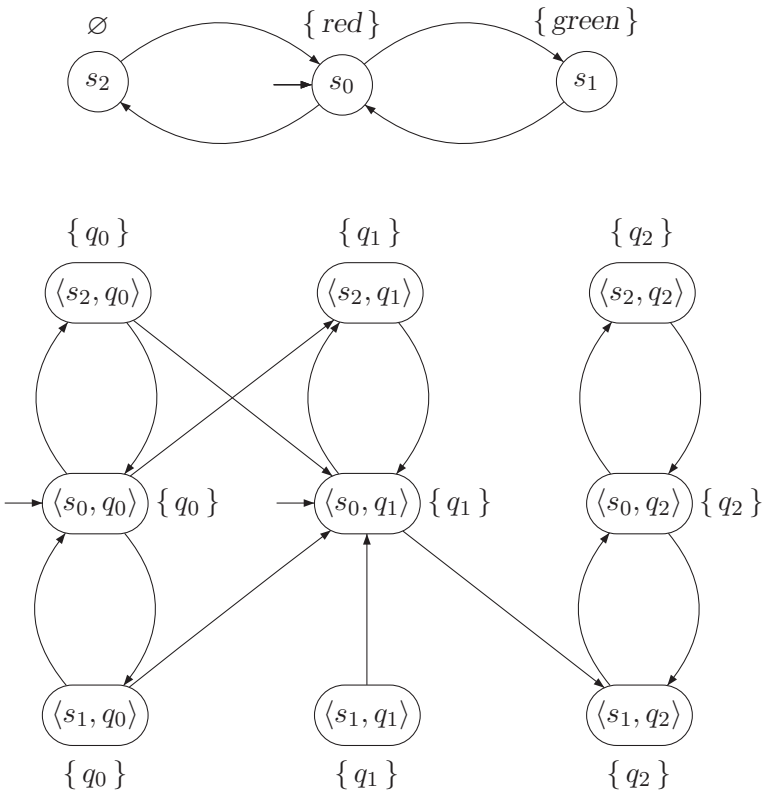


Figure 4.23: A simple traffic light that can switch off (upper part) and its product (lower part).

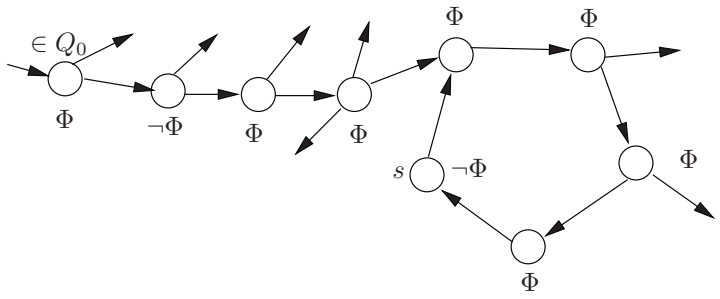


Figure 4.24: An example of a run violating "eventually always" Φ.

This can be justified intuitively as follows. Suppose s is a state that is reachable from an initial state in TS and $s \not\models \Phi$. As s is reachable, TS has an initial path fragment that ends in s. If s is on a cycle, then this path fragment can be continued by an infinite path that is obtained by traversing the cycle containing s infinitely often. In this way, we obtain a path in TS that visits the $\neg\Phi$-state s infinitely often. But then, $TS \not\models P_{pers}$. This is exemplified in Figure 4.24 where a fragment of a transition system is shown; for simplicity the action labels have been omitted. (Note that – in contrast to invariants – a state violating Φ which is not on a cycle does not cause the violation of P_{pers}.)

The reduction of checking whether $TS \models P_{pers}$ to a cycle detection problem is formalized by the following theorem.

Theorem 4.65. Persistence Checking and Cycle Detection

Let TS be a finite transition system without terminal states over AP, Φ a propositional formula over AP, and P_{pers} the persistence property "eventually forever Φ". Then, the following statements are equivalent:

(a) $TS \not\models P_{pers}$,

(b) There exists a reachable ¬Φ-state s which belongs to a cycle. Formally:

$$\exists s \in \mathit{Reach}(TS).\ s \not\models \Phi \wedge s \text{ is on a cycle in } G(TS)\ .^5$$

Before providing the proof, let us first explain how to obtain an error indication whenever $TS \not\models P_{pers}$. Let $\widehat{\pi} = u_0\, u_1\, u_2\, \ldots\, u_k$ be a path in the graph induced by TS, i.e., $G(TS)$, such that $k > 0$ and $s = u_0 = u_k$. Assume $s \not\models \Phi$. That is, $\widehat{\pi}$ is a cycle in $G(TS)$ containing

[5] Recall that $G(TS)$ denotes the underlying directed graph of TS.

a state violating Φ. Let $s_0\, s_1\, s_2 \ldots s_n$ be an initial path fragment of TS such that $s_n = s$. Then the concatenation of this initial path fragment and the unfolding of the cycle

$$\pi = s_0\, s_1\, s_2 \ldots \underbrace{s_n}_{=s}\, u_1\, u_2\, \ldots\, \underbrace{u_k}_{=s}\, u_1\, u_2\, \ldots\, \underbrace{u_k}_{=s} \ldots$$

is a path in TS. As state $s \not\models \Phi$ is visited by π infinitely often, it follows that π does not satisfy "eventually always Φ". The prefix

$$s_0\, s_1\, s_2 \ldots \underbrace{s_n}_{=s}\, u_1\, u_2\, \ldots\, \underbrace{u_k}_{=s}$$

can be used as diagnostic feedback as it shows that s may be visited infinitely often.

Proof: Let $TS = (S, Act, \rightarrow, I, AP)$.

(a) $\Longrightarrow$ (b): Assume $TS \not\models P_{pers}$, i.e., there exists a path $\pi = s_0\, s_1\, s_2 \ldots$ in TS such that $trace(\pi) \notin P_{pers}$. Thus, there are infinitely many indices i such that $s_i \not\models \Phi$. Since TS is finite, there is a state s with $s = s_i \not\models \Phi$ for infinitely many i. As s appears on a path starting in an initial state we have $s \in Reach(TS)$. A cycle $\widehat{\pi}$ is obtained by any fragment $s_i\, s_{i+1}\, s_{i+2} \ldots s_{i+k}$ of π where $s_i = s_{i+k} = s$ and $k > 0$.

(b) $\Longrightarrow$ (a): Let s and $\widehat{\pi} = u_0 u_1 \ldots u_k$ be as indicated above, i.e., $s \in Reach(TS)$ and $\widehat{\pi}$ is a cycle in TS with $s = u_0 = u_k$. Since $s \in Reach(TS)$, there in an initial state $s_0 \in I$ and a path fragment $s_0\, s_1 \ldots s_n$ with $s_n = s$. Then:

$$\pi = s_0\, s_1\, s_2 \ldots \underbrace{s_n}_{=s}\, u_1\, u_2\, \ldots\, \underbrace{u_k}_{=s}\, u_1\, u_2\, \underbrace{u_k}_{=s} \ldots$$

is a path in TS. Since $s \not\models \Phi$, it follows that π does not satisfy "eventually forever Φ", and thus $TS \not\models P_{pers}$. ■

Example 4.66. Pedestrian Traffic Lights Revisited

Consider the transition system model of the simple traffic light that one typically encounters at pedestrian crossings (see Figure 4.22) and the persistence property "eventually forever" $\neg q_1$ where q_1 is the accept state of the NBA $\mathcal{A}$. As there is no reachable cycle in the product transition system that contains a state violating $\neg q_1$, i.e., a cycle that contains a state labeled with q_1, it follows that

$$PedTrLight \otimes \mathcal{A} \models \text{"eventually forever"}\ \neg q_1.$$

For the traffic light that has the possibility to automatically switch off, it can be inferred directly from the product transition system (see the lower part of Figure 4.23) that there

is a reachable state, e.g., $\langle s_2, q_1 \rangle \not\models \neg q_1$, that lies on a cycle. Thus:

$$\mathit{PedTrLight'} \otimes \mathcal{A} \not\models \text{“eventually forever” } \neg q_1.$$

■

Thus, persistence checking for a finite transition system requires the same techniques as checking emptiness of an NBA, see page 184. In fact, the algorithm we suggest below can also be used for checking emptiness in an NBA.

A Naive Depth-First Search Theorem 4.65 entails that in order to check the validity of a persistence property, it suffices to check whether there exists a reachable cycle containing a $\neg\Phi$-state. How to check for such reachable cycles? A possibility is to compute the strongly connected components (SCCs, for short) in $G(TS)$ – this can be done in a worst-case time complexity that is linear in the number of states and transitions – and to check whether one such SCC is reachable from an initial state, contains at least one edge, and, moreover, contains a $\neg\Phi$-state. If indeed such a SCC does exist, P_{pers} is refuted; otherwise the answer is affirmative.

Although this SCC-based technique is optimal with respect to the asymptotic worst-case time complexity, it is more complex and less adequate for an on-the-fly implementation. In that respect, pure cycle-check algorithms are more appropriate. Therefore, in the sequel we will detail the standard DFS-based cycle detection algorithm.

Let us first recall how for a finite directed graph G and node v, it can be checked with a DFS-based approach whether v belongs to a cycle. For this, one may simply start a depth-first search in node v and check for any visited node w whether there is an edge from w to v. If so, a cycle has been found: it starts in v and follows the path to node w given by the current stack content and then takes the edge from w to v. Vice versa, if no such edge is found, then v does not belong to a cycle. To determine whether G has a cycle, a similar technique can be exploited: we perform a depth-first search to visit all nodes in G. Moreover, on investigating the edge from w to v it is checked whether v has already been visited and whether v is still on the DFS stack. If so, then a so-called *backward edge* has been found which closes a cycle. Otherwise, if no backward edge has been found during the DFS in G then G is acyclic. (A detailed description of this DFS-based cycle detection technique can be found in textbooks on algorithms and data structures, e.g., [100].)

We now DFS-based cycle checks (by searching for backward edges) for persistence checking. The naive approach works in two phases, as illustrated in Algorithm 6:

1. In the first step, all states satisfying $\neg\Phi$ that are reachable from some initial state are determined. This is performed by a standard depth-first search.

2. In the second step, for each reachable $\neg\Phi$-state s, it is checked whether it belongs to a cycle. This algorithm (called cycle_check, see Algorithm 7) relies on the technique sketched above: we start a depth-first search in s (with initially empty DFS stack V and initially empty set T of visited states) and check for all states reachable from s whether there is an outgoing backward edge leading to s.

This yields an algorithm for checking the validity of the persistence property with quadratic worst-case running time. More precisely, its time complexity is in $\mathcal{O}(N{\cdot}(|\Phi| + N{+}M))$ where N is the number of *reachable* states in TS and M the number of transitions between these reachable states. This can be seen as follows. Visiting all states that are reachable from some initial state takes $\mathcal{O}(N{+}M{+}N{\cdot}|\Phi|)$ as a depth-first search over all states suffices and in each reachable state the validity of Φ is checked (which is assumed to be linear in the size of Φ). In the worst case, all states refute Φ, and a depth-first search takes place (procedure cycle_check) for all these states. This takes $\mathcal{O}(N{\cdot}(N{+}M))$. Together this yields $\mathcal{O}(N{\cdot}(|\Phi| + N{+}M))$.

Several simple modifications of the suggested technique are possible to increase efficiency. For instance, cycle_check(s') can be invoked inside visit(s) immediately before or after s' is inserted into $R_{\neg\Phi}$, in which case the whole persistence checking algorithm can abort with the answer "no" if cycle_check(s') returns true. However, the quadratic worst-case running time cannot be avoided if the cycle check algorithm for the $\neg\Phi$-states relies on separate depth-first searches. The problem is that certain fragments of TS might be reachable from different $\neg\Phi$-states. These fragments are (re-)explored in the depth-first searches (cycle_check) invoked by several $\neg\Phi$-states. To obtain linear running time, we aim at a cycle detection algorithm that searches for backward edges leading to one of the $\neg\Phi$-states and ensures that any state is visited *at most once* in the depth-first searches for the cycle detection. This will be explained in the following subsection.

A Nested Depth-First Search Algorithm The rough idea of the linear-time cycle detection-based persistence checking algorithm is to perform two depth-first searches (DFSs) in TS in an interleaved way. The first (outer) depth-first search serves to encounter all reachable $\neg\Phi$-states. The second (inner) depth-first search seeks backward edges leading to a $\neg\Phi$-state. The inner depth-first search is *nested* in the outer one in the following sense: whenever a $\neg\Phi$-state s has been fully expanded by the outer depth-first search, then the inner depth-first search continues with state s and visits all states s' that are reachable from s and that have not yet been visited in the inner depth-first search before. If no backward edge has been found when treating s in the inner DFS, then the outer DFS continues until the next $\neg\Phi$-state t has been fully expanded, in which case the inner DFS proceeds with t.

Algorithm 6 Naive persistence checking

Input: finite transition system TS without terminal states, and proposition Φ
Output: "yes" if $TS \models$ "eventually forever Φ", otherwise "no".

```
set of  states R := ∅; R_¬Φ := ∅;          (* set of reachable states resp. ¬Φ-states *)
stack of  states U := ε;                   (* DFS stack for first DFS, initial empty *)
set of  states T := ∅;                     (* set of visited states for the cycle check *)
stack of  states V := ε;                   (* DFS stack for the cycle check *)

for all s ∈ I \ R do visit(s); od          (* a DFS for each unvisited initial state *)
for all s ∈ R_¬Φ do
  T := ∅; V := ε;                          (* initialize set T and stack V *)
  if  cycle_check(s) then return "no"      (* s belongs to a cycle *)
od
return "yes"                               (* none of the ¬Φ-states belongs to a cycle *)
```

```
procedure visit (state s)
  push(s, U);                              (* push s on the stack *)
  R := R ∪ { s };                          (* mark s as reachable *)
  repeat
    s' := top(U);
    if Post(s') ⊆ R then
      pop(U);
      if s' ⊭ Φ then R_¬Φ := R_¬Φ ∪ { s' }; fi
    else
      let s'' ∈ Post(s') \ R
      push(s'', U);
      R := R ∪ { s'' };                    (* state s'' is a new reachable state *)
    fi
  until (U = ε)
endproc
```

Algorithm 7 Cycle detection

Input: finite transition system TS and state s in TS with $s \not\models \Phi$
Output: true if s lies on a cycle in TS, otherwise false

```
(* T organizes the set of states that have been visited, V serves as DFS stack.   *)
(* In the standard approach to check whether there is a backward edge to s,      *)
(* T and V are initially empty.                                                  *)

procedure boolean cycle_check(state s)
  boolean cycle_found := false;                          (* no cycle found yet *)
  push(s, V);                                            (* push s on the stack *)
  T := T ∪ { s };
  repeat
    s' := top(V);                                        (* take top element of V *)
    if s ∈ Post(s') then
      cycle_found := true;                               (* if s ∈ Post(s'), a cycle is found *)
      push(s, V);                                        (* push s on the stack *)
    else
      if Post(s') \ T ≠ ∅ then
        let s'' ∈ Post(s') \ T;
        push(s'', V);                                    (* push an unvisited successor of s' *)
        T := T ∪ { s'' };                                (* and mark it as reachable *)
      else
        pop(V);                                          (* unsuccessful cycle search for s' *)
      fi
    fi
  until ((V = ε) ∨ cycle_found)
  return cycle_found
endproc
```

Algorithm 8 Persistence checking by nested depth-first search

Input: transition system TS without terminal states, and proposition Φ
Output: "yes" if $TS \models$ "eventually forever Φ", otherwise "no" plus counterexample

```
set of states R := ∅;                          (* set of visited states in the outer DFS *)
stack of states U := ε;                        (* stack for the outer DFS *)
set of states T := ∅;                          (* set of visited states in the inner DFS *)
stack of states V := ε;                        (* stack for the inner DFS *)
boolean cycle_found := false;

while (I \ R ≠ ∅ ∧ ¬cycle_found) do
  let s ∈ I \ R;                               (* explore the reachable *)
  reachable_cycle(s);                          (* fragment with outer DFS *)
od
if ¬cycle_found then
  return ("yes")                               (* TS ⊨ "eventually forever Φ" *)
else
  return ("no", reverse(V.U))                  (* stack contents yield a counterexample *)
fi
```

```
procedure reachable_cycle (state s)
  push(s, U);                                  (* push s on the stack *)
  R := R ∪ { s };
  repeat
    s' := top(U);
    if Post(s') \ R ≠ ∅ then
      let s'' ∈ Post(s') \ R;
      push(s'', U);                            (* push the unvisited successor of s' *)
      R := R ∪ { s'' };                        (* and mark it reachable *)
    else
      pop(U);                                  (* outer DFS finished for s' *)
      if s' ⊭ Φ then
        cycle_found := cycle_check(s');        (* proceed with the inner *)
                                               (* DFS in state s' *)
      fi
    fi
  until ((U = ε) ∨ cycle_found)                (* stop when stack for the outer *)
                                               (* DFS is empty or cycle found *)
endproc
```

This algorithm is called *nested depth-first search*. Algorithm 8 shows the pseudocode for the outer depth-first search (called reachable_cycle) which invokes the inner depth-first search (cycle_check, cf. Algorithm 7 on page 210). Later we will explain that it is important that cycle_check(s) is called immediately after the outer depth-first search has visited all successors of s. The major difference from the naive approach is that cycle_check(s) reuses the information of previous calls of cycle_check($\cdot$) and ignores all states in T. When cycle_check(s) is invoked then T consists of all states that have been visited in the inner DFS before, i.e., during the execution of cycle_check(u) called prior to cycle_check(s).

An interesting aspect of this nested depth-first strategy is that once a cycle is determined containing a $\neg\Phi$-state s, then a path to s can be computed easily: stack U for the outer DFS contains a path fragment from the initial state $s_0 \in I$ to s (in reversed order), while stack V for the inner DFS — as maintained by the procedure cycle_check($\cdot$) — contains a cycle from state s to s (in reversed order). Concatenating these path fragments thus provides an error indication in the same vein as described before in the proof of Theorem 4.65 on page 205.

Example 4.67. Running the Nested DFS Algorithm

Consider the transition system depicted in Figure 4.25 and assume that $s_0 \models \Phi$, $s_3 \models \Phi$, whereas $s_1 \not\models \Phi$ and $s_2 \not\models \Phi$. Consider the scenario in which s_2 is considered as a first successor of s_0, i.e., s_2 is considered in the outer depth-first search prior to considering state s_1. This means that the order in which the states are put by Algorithm 8 on the stack U equals $\langle s_1, s_3, s_2, s_0\rangle$ where we write the stack content from the top to the bottom, i.e., s_1 is the top element (the last element that has been pushed on the stack). Besides, $R = S$. Thus, the outer DFS takes s_1 as the top element of U and verifies $Post(s_1) \subseteq R$. As $s_1 \not\models \Phi$, the invocation cycle_check(s_1) takes place and s_1 is deleted from stack U for the outer DFS. This yields that s_1 is put on the stack V for the inner DFS, followed by its only successor s_3. Then the cycle $s_1 \to s_3 \to s_1$ is detected, cycle_check(s_1) yields true, and as a result, reachable_cycle(s_0) terminates with the conclusion that P_{pers} on Φ is refuted. An error indication is obtained by first concatenating the content of $V = \langle s_1, s_3, s_1\rangle$ and $U = \langle s_3, s_2, s_0\rangle$ and then reversing the order. This yields the path $s_0\, s_2\, s_3\, s_1\, s_3\, s_1$, which is indeed a prefix of a run refuting "eventually always Φ". ■

The soundness of the nested depth-first search is no longer trivial because – by the treatment of T as a global variable – not all states reachable from s are explored in cycle_check(s), but only those states that have not been visited before in the inner depth-first search. Thus, we could imagine that there is a cycle with $\neg\Phi$-states which will not be found with the nested depth-first search: this cycle might have been explored during cycle_check(u) where u does not belong to this or any other cycle. Then, none of the following procedure calls of cycle_check($\cdot$) will find this cycle. The goal is now to show

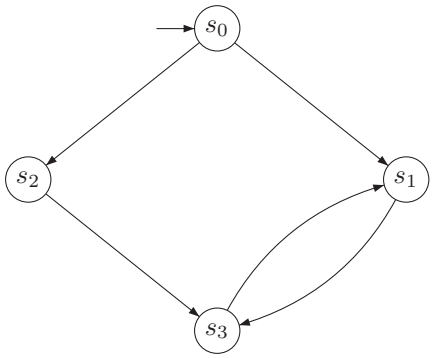


Figure 4.25: An example of a transition system for nested DFS.

that this cannot happen, i.e., if there is a reachable cycle with $\neg\Phi$-states, then the nested depth-first search will find such a cycle.

In fact, things are more tricky than it seems. In Algorithm 8 it is important that the invocations to the procedure cycle_check(s) for $s \in \mathit{Reach}(TS)$ and $s \not\models \Phi$ occur in appropriate order. Let $s \not\models \Phi$. Then cycle_check(s) is only invoked if $\mathit{Post}(s) \subseteq R$, i.e., when all states reachable from s have been encountered and visited.[6] So, the invocation cycle_check(s) is made immediately once all states that are reachable from s have been visited and expanded in the outer depth-first search. As a result, Algorithm 8 satisfies the following property: if s' is a successor of state s in the visit-order of the outer depth-first search and $s \not\models \Phi$ and $s' \not\models \Phi$, then the invocation cycle_check(s') occurs prior to the invocation cycle_check(s).

Example 4.68. Modifying the Nested DFS

Let us illustrate by an example that the nested depth-first search algorithm would be wrong if the outer and inner DFS are interleaved in an arbitrary way. We consider again the transition system in Figure 4.25 on page 213. We start the outer DFS with the initial state s_0 and assume that state s_2 is visited prior to s_1. Let us see what happens if we do not wait until s_2 has been fully expanded in the outer DFS and start cycle_check(s_2) immediately. Then, cycle_check(s_2) visits s_3 and s_1 and returns false, since there is no backward edge leading to state s_2. Thus, cycle_check(s_2) yields $T = \{s_2, s_3, s_1\}$ (and $V = \varepsilon$). Now the outer DFS proceeds and visits s_3 and s_1. It calls cycle_check(s_1) which fails to find the cycle $s_1 s_3 s_1$. Note that cycle_check(s_1) immediately halts and returns false since $s_1 \in T = \{s_2, s_3, s_1\}$. Thus, the nested depth-first search would return the wrong answer "yes". ■

[6] For a recursive formulation of the outer depth-first search, this is the moment where the depth-first search call for s terminates.

Theorem 4.69. Correctness of the Nested DFS

Let TS be a finite transition system over AP without terminal states, Φ a propositional formula over AP, and P_{pers} the persistence property "eventually forever Φ". Then:

$$\textit{Algorithm 8 returns the answer "no" if and only if } TS \not\models P_{pers}.$$

Proof:

$\Rightarrow$: This follows directly from the fact that the answer "false" is only obtained when an initial path fragment of the form $s_0 \ldots s \ldots s$ has been encountered for some $s_0 \in I$ and $s \not\models \Phi$. But then, $TS \not\models P_{pers}$.

$\Leftarrow$: To establish this direction, we first prove that the following condition holds:

(*) On invoking cycle_check(s), there is no cycle $s'_0\, s'_1 \ldots s'_k$ in TS such that:

$$\{ s'_0, s'_1, \ldots, s'_k \} \cap T \neq \varnothing \quad \text{and} \quad s \in \{ s'_0, \ldots, s'_k \}.$$

So, on invoking cycle_check(s), there is no cycle containing s and a state in T. Statement (*) ensures that in the inner DFS, all already visited states in T—the states that were visited during an earlier cycle detection—can be safely ignored during the next cycle search. In other words, if s belongs to a cycle then cycle_check(s) has to find one such cycle. We prove the statement (*) by contradiction. Consider the invocation cycle_check(s). Assume there exists a cycle $s'_0\, s'_1 \ldots s'_k$ such that

$$s'_0 = s'_k = s \quad \text{and} \quad s \not\models \Phi \quad \text{and} \quad \{ s'_0, s'_1, \ldots, s'_k \} \cap T \neq \varnothing.$$

Without loss of generality we assume that s is the *first* state for which this condition holds on invoking cycle_check(s). More precisely, we assume:

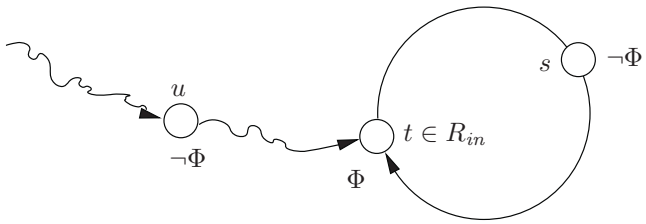
(+) For all states $\widehat{s}$ with $\widehat{s} \not\models \Phi$ where cycle_check($\widehat{s}$) has been invoked prior to cycle_check(s), there is no cycle containing $\widehat{s}$ and a state in T, on invoking cycle_check(s).

Let $t \in \{ s'_0, \ldots, s'_k \} \cap T$, i.e., t is a state on a cycle with state s. Since $t \in T$ on the invocation cycle_check(s), there must be a state, u say, for which cycle_check($\cdot$) has been invoked earlier and during this search t has been encountered (and added to T). Thus, we have $u \not\models \Phi$ and the following conditions (4.1) and (4.2):

$$\text{cycle\_check}(u) \text{ was invoked prior to cycle\_check}(s) \tag{4.1}$$

as a result of cycle_check(u), t was visited and added to T (4.2)

Obviously, from (4.2), it follows that state t is reachable from u. As the states s and t are on a cycle, and t is reachable from u, we have that s is reachable from u. This situation is sketched in the following figure:



Now consider the outer depth-first search, i.e., reachable_cycle(·), and discuss the following cases 1 and 2:

1. u has been visited prior to s in the outer DFS, i.e., s has been pushed on stack U after u.

   Since s is reachable from u, state s is visited during the expansion of u in the outer DFS and taken from stack U before u. Hence, cycle_check(s) is invoked prior to cycle_check(u) which contradicts (4.1).

2. u has been visited after s in the outer DFS, i.e., s has been pushed on stack U before u.

   By (4.1), s is still in stack U when cycle_check(u) is invoked. This yields that u is reachable from s. But as s is reachable from u, this means that s and u are on a cycle. This cycle or another cycle with state u would have been encountered during cycle_check(u) because of (+) and Algorithm 8 would have terminated without invoking cycle_check(s).

■

It remains to discuss the complexity of the nested depth-first search algorithm. Since T is increasing during the execution of the nested depth-first search (i.e., we insert states in T, but never take them out) any state in TS is visited at most once in the inner depth-first search, when ranging over all procedure calls of cycle_check(·). The same holds for the outer depth-first search since it is roughly a standard depth-first search. Thus, each reachable state s' in TS is taken

- at most once[7] as the top element of stack U in the outer depth-first search;
- at most once as the top element of stack V in the inner depth-first search (when ranging over all calls of cycle_check).

In particular, this observation yields the termination of Algorithm 8. The cost caused by the top-element s' for the body of the repeat loop in cycle_check (inner depth-first search) and in reachable_cycle (outer depth-first search) is $\mathcal{O}(|Post(s')|)$. Thus, the worst-case time complexity of Algorithm 8 is linear in the size of the reachable fragment of TS, but has the chance to terminate earlier when a cycle has been found. In the following theorem we also take into account that Φ might be a complex formula and evaluating the truth value of Φ for a given state (by means of its label) requires $\mathcal{O}(|\Phi|)$ steps.

Theorem 4.70. Time Complexity of Persistence Checking

The worst-case time complexity of Algorithm 8 is in $\mathcal{O}((N+M) + N \cdot |\,\Phi\,|)$ *where* N *is the number of reachable states, and* M *the number of transitions between the reachable states.*

The space complexity is bounded above by $\mathcal{O}(|S| + |\rightarrow|)$ where S is the state space of TS and $|\rightarrow|$ the number of transitions in TS. This is an adequate bound if we assume a representation of TS by adjacency lists. However, in the context of model checking the starting point is typically not an explicit representation of the composite transition system, but a syntactic description of the concurrent processes, e.g., by high-level modeling languages with a program graph or channel system semantics. Such syntactic descriptions are typically much smaller than the resulting transition system. (Recall the state explosion problem which appears through parallel composition and the unfolding of program graphs into transition systems, see page 77 ff.) In the persistence checking algorithm, we may assume that the elements in $Post(s')$ are generated on the fly by means of the semantic rules for the transition relation. Ignoring the space required for the syntactic descriptions of the processes, the additional space requirements for the presented persistence checking algorithm is $\mathcal{O}(N)$ for the sets T and R and the stacks U and V, where N is the number of reachable states in TS. In fact, the representation of T and R is the most (space-)critical aspect when implementing the nested depth-first search and running it on large examples. Typically, T and R are organized using appropriate hash techniques. In fact, T and R can even be represented by a single hash table where the entries are pairs $\langle s, b\rangle$ with $b \in \{0, 1\}$. The meaning of $\langle s, 0\rangle$ is that s is in R, but not in T (i.e., s has been visited in the outer, but not yet in the inner DFS). The pair $\langle s, 1\rangle$ means that s has been visited in both the outer and the inner DFS. The single bit b is sufficient to cover all possible cases, since T is always a subset of R.

[7] exactly once, if Algorithm 8 does not abort with the answer "no".

Another simple observation can speed up the nested depth-first search in case the persistence property is violated: whenever the inner DFS cycle_check(s) reaches a state t which is on the stack U (the DFS stack for the outer DFS), then the upper part of U yields a path fragment from t to s, while the content of the DFS stack V for the inner DFS describes a path fragment from s to t. Thus, a cycle has been found which visits s infinitely often and the nested DFS may abort with a counterexample. To support checking whether state t is contained in stack U, a further bit can be added to the entries in the hash table for representing T and R. I.e., we then have to deal with a hash table for triples $\langle s, b, c\rangle$ with $s \in R$ and $b, c \in \{0, 1\}$ depending on whether s is in T (in which case $b = 1$) and whether s is in U (in which case $c = 1$). This leads to a slight variant of Algorithm 8 which is often faster and generates smaller counterexamples than the original version.

4.5 Summary

- NFAs and DFAs are equivalent automata models for regular languages and can serve to represent the bad prefixes of regular safety properties.
- Checking a regular safety property on a finite transition system is solvable by checking an invariant on a product automaton, and thus amounts to solving a reachability problem.
- ω-Regular languages are languages of infinite words that can be described by ω-regular expressions.
- NBAs are acceptors for infinite words. The syntax is as for NFAs. The accepted language of an NBA is the set of all infinite words that have a run where an accept state is visited infinitely often.
- NBAs can serve to represent ω-regular properties.
- The class of languages that are recognized by NBAs agrees with the class of ω-regular languages.
- DBAs are less powerful than NBAs and fail, for instance, to represent the persistence property "eventually forever a".
- Generalized NBAs are defined as NBAs, except that they require repeated visits for several acceptance sets. Their expressiveness is the same as for NBAs.
- Checking an ω-regular property P on a finite transition system TS can be reduced to checking the persistence property "eventually forever no accept state" in the product of TS and an NBA for the undesired behaviors (i.e., the complement property $\overline{P}$).

- Persistence checking requires checking the existence of a reachable cycle containing a state violating the persistence condition. This is solvable in linear time by a nested depth-first search (or by analyzing the strongly connected components). The same holds for the nonemptiness problem for NBAs which boils down to checking the existence of a reachable cycle containing an accept state.
- The nested depth-first search approach consists of the interleaving of two depth-first searches: one for encountering the reachable states, and one for cycle detection.

4.6 Bibliographic Notes

Finite automata and regular languages. The first papers on finite automata were published in the nineteen fifties by Huffman [217], Mealy [291], and Moore [303] who used deterministic finite automata for representing sequential circuits. Regula r expressions and their equivalence to finite automata goes back to Kleene [240]. Rabin and Scott [350] presented various algorithms on finite automata, including the powerset construction. The existence of minimal DFAs relies on results stated by Myhill [309] and Nerode [313]. The $\mathcal{O}(N \log N)$ minimization algorithm we mentioned at the end of Section 4.1 has been suggested by Hopcroft [213]. For other algorithms on finite automata, a detailed description of the techniques sketched here and other aspects of regular languages, we refer to the text books [272, 363, 214, 383] and the literature mentioned therein.

Automata over infinite words. Research on automata over infinite words (and trees) started in the nineteen sixties with the work of Büchi [73], Trakhtenbrot [392], and Rabin [351] on decision problems for mathematical logics. At the same time, Muller [307] studied a special type of deterministic ω-automata (today called Muller automata) in the context of asynchronous circuits. The equivalence of nondeterministic Büchi automata and ω-regular expressions has been shown by McNaughton [290]. He also established a link between NBAs and deterministic Muller automata [307] by introducing another acceptance condition that has been later formalized by Rabin [351]. (The resulting ω-automata type is today called Rabin automata.) An alternative transformation from NBA to deterministic Rabin automata has been presented by Safra [361]. Unlike Büchi automata, the nondeterministic and deterministic versions of Muller and Rabin automata are equally expressive and yield automata-characterizations of ω-regular languages. The same holds for several other types of ω-automata that have been introduced later, e.g., Streett automata [382] or automata with the parity condition [305].

The fact that ω-regular languages are closed under complementation (we stated this result without proof) can be derived easily from deterministic automata representations. The proof of Theorem 4.50 follows the presentation given in the book by Peled [327]. Various

decision problems for ω-automata have been addressed by Landweber [261] and later by Emerson and Lei [143] and Sistla, Vardi, and Wolper [373]. For a survey of automata on infinite words, transformations between the several classes of ω-automata, complementation operators and other algorithms on ω-automata, we refer to the articles by Choueka [81], Kaminsky [229], Staiger [376], and Thomas [390, 391]. An excellent overview of the main concepts of and recent results on ω-automata is provided by the tutorial proceedings [174].

Automata and linear-time properties. The use of Büchi automata for the representation and verification of linear-time properties goes back to Vardi and Wolper [411, 412] who studied the connection of Büchi automata with linear temporal logic. Approaches with similar automata models have been developed independently by Lichtenstein, Pnueli, and Zuck [274] and Kurshan [250]. The verification of (regular) safety properties has been described by Kupferman and Vardi [249]. The notion of persistence property has been introduced by Manna and Pnueli [282] who provided a hierarchy of temporal properties. The nested depth-first algorithm (see Algorithm 8) originates from Courcoubetis et al. [102] and its implementation in the model checker SPIN has been reported by Holzmann, Peled, and Yannakakis [212]. The Murφ verifier developed by Dill [132] focuses on verifying safety properties. Variants of the nested depth-first search have been proposed by several authors, see, e.g., [106, 368, 161, 163]. Approaches that treat generalized Büchi conditions (i.e., conjunctions of Büchi conditions) are discussed in [102, 388, 184, 107]. Further implementation details of the nested depth-first search approach can be found in the book by Holzman [209].

4.7 Exercises

Exercise 4.1. Let $AP = \{ a, b, c \}$. Consider the following LT properties:

(a) If a becomes valid, afterward b stays valid ad infinitum or until c holds.

(b) Between two neighboring occurrences of a, b always holds.

(c) Between two neighboring occurrences of a, b occurs more often than c.

(d) $a \wedge \neg b$ and $b \wedge \neg a$ are valid in alternation or until c becomes valid.

For each property P_i ($1 \leqslant i \leqslant 4$), decide if it is a regular safety property (justify your answers) and if so, define the NFA $\mathcal{A}_i$ with $\mathcal{L}(\mathcal{A}_i) = \mathit{BadPref}(P_i)$. (*Hint: You may use propositional formulae over the set AP as transition labels.*)

EXERCISE 4.2. Let $n \geqslant 1$. Consider the language $\mathcal{L}_n \subseteq \Sigma^*$ over the alphabet $\Sigma = \{ A, B \}$ that consists of all finite words where the symbol B is on position n from the right, i.e., $\mathcal{L}$ contains exactly the words $A_1 A_2 \ldots A_k \in \{A, B\}^*$ where $k \geqslant n$ and $A_{k-n+1} = B$. For instance, the word $ABBAABAB$ is in $\mathcal{L}_3$.

(a) Construct an NFA $\mathcal{A}_n$ with at most $n{+}1$ states such that $\mathcal{L}(\mathcal{A}_n) = \mathcal{L}_n$.

(b) Determinize this NFA $\mathcal{A}_n$ using the powerset construction algorithm.

EXERCISE 4.3. Consider the transition system TS_{Sem} for the two-process mutual exclusion with a semaphore (see Example 2.24 on page 43) and TS_{Pet} for Peterson's algorithm (see Example 2.25 on page 45).

(a) Let P_{safe} be the regular safety property "process 1 never enters its critical section from its noncritical section (i.e., process 1 must be in its waiting location before entering the critical section)" and $AP = \{ \mathit{wait}_1, \mathit{crit}_1 \}$.

  (i) Depict an NFA for the minimal bad prefixes for P_{safe}.

  (ii) Apply the algorithm in Section 4.2 to verify $TS_{Sem} \models P_{safe}$.

(b) Let P_{safe} be the safety property "process 1 never enters its critical section from a state where $x = 2$" and $AP = \{ \mathit{crit}_1, x = 2 \}$.

  (i) Depict an NFA for the minimal bad prefixes for P_{safe}.

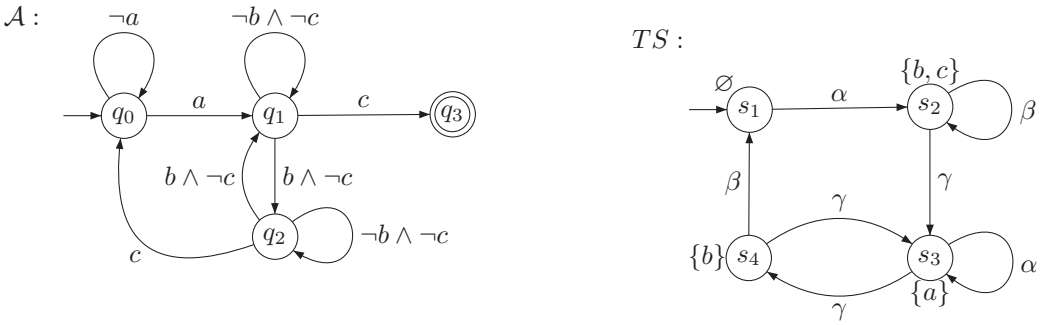
  (ii) Apply the algorithm in Section 4.2 to verify $TS_{Pet} \not\models P_{safe}$. Which counterexample is returned by the algorithm?

EXERCISE 4.4. Let P_{safe} be a safety property. Prove or disprove the following statements:

(a) If $\mathcal{L}$ is a regular language with $\mathit{MinBadPref}(P_{safe}) \subseteq \mathcal{L} \subseteq \mathit{BadPref}(P_{safe})$, then P_{safe} is regular.

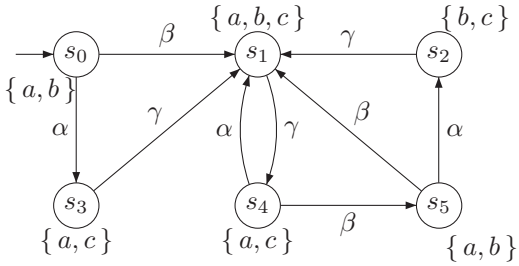
(b) If P_{safe} is regular, then any $\mathcal{L}$ for which $\mathit{MinBadPref}(P_{safe}) \subseteq \mathcal{L} \subseteq \mathit{BadPref}(P_{safe})$ is regular.

EXERCISE 4.5. Let $AP = \{ a, b, c \}$. Consider the following NFA $\mathcal{A}$ (over the alphabet 2^{AP}) and the following transition system TS:



Construct the product $TS \otimes \mathcal{A}$ of the transition system and the NFA.

Exercise 4.6. Consider the following transition system TS



and the regular safety property

$$P_{safe} = \begin{array}{l}\text{“always if } a \text{ is valid and } b \wedge \neg c \text{ was valid somewhere before,}\\ \text{then } a \text{ and } b \text{ do not hold thereafter at least until } c \text{ holds”}\end{array}$$

As an example, it holds:

$$\begin{array}{ll}\{b\}\varnothing\{a, b\}\{a, b, c\} & \in \mathit{pref}(P_{safe})\\ \{a, b\}\{a, b\}\varnothing\{b, c\} & \in \mathit{pref}(P_{safe})\\ \{b\}\{a, c\}\{a\}\{a, b, c\} & \in \mathit{BadPref}(P_{safe})\\ \{b\}\{a, c\}\{a, c\}\{a\} & \in \mathit{BadPref}(P_{safe})\end{array}$$

Questions:

(a) Define an NFA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathit{MinBadPref}(P_{safe})$.

(b) Decide whether $TS \models P_{safe}$ using the $TS \otimes \mathcal{A}$ construction. Provide a counterexample if $TS \not\models P_{safe}$.

Exercise 4.7. Prove or disprove the following equivalences for ω-regular expressions:

(a) $(\mathsf{E}_1 + \mathsf{E}_2).\mathsf{F}^\omega \equiv \mathsf{E}_1.\mathsf{F}^\omega + \mathsf{E}_2.\mathsf{F}^\omega$

(b) $\mathsf{E}.(\mathsf{F}_1 + \mathsf{F}_2)^\omega \equiv \mathsf{E}.\mathsf{F}_1^\omega + \mathsf{E}.\mathsf{F}_2^\omega$

(c) $\mathsf{E}.(\mathsf{F}.\mathsf{F}^*)^\omega \equiv \mathsf{E}.\mathsf{F}^\omega$

(d) $(\mathsf{E}^*.\mathsf{F})^\omega \equiv \mathsf{E}^*.\mathsf{F}^\omega$

where $\mathsf{E}, \mathsf{E}_1, \mathsf{E}_2, \mathsf{F}, \mathsf{F}_1, \mathsf{F}_2$ are arbitrary regular expressions with $\varepsilon \notin \mathcal{L}(\mathsf{F}) \cup \mathcal{L}(\mathsf{F}_1) \cup \mathcal{L}(\mathsf{F}_2)$.

Exercise 4.8. Generalized ω-regular expressions are built from the symbols $\underline{\varnothing}$ (to denote the empty language), $\underline{\varepsilon}$ (to denote the language $\{\varepsilon\}$ consisting of the empty word), the symbols $\underline{A}$ for $A \in \Sigma$ (for the singleton sets $\{A\}$) and the language operators "+" (union), "." (concatenation), "$*$" (Kleene star, finite repetition), and "ω"(infinite repetition). The semantics of a generalized ω-regular expression G is a language $\mathcal{L}_g(\mathsf{G}) \subseteq \Sigma^* \cup \Sigma^\omega$, which is defined by

- $\mathcal{L}_g(\underline{\varnothing}) = \varnothing, \quad \mathcal{L}_g(\underline{\varepsilon}) = \{\varepsilon\}, \quad \mathcal{L}_g(\underline{A}) = \{A\},$
- $\mathcal{L}_g(\mathsf{G}_1 + \mathsf{G}_2) = \mathcal{L}_g(\mathsf{G}_1) \cup \mathcal{L}_g(\mathsf{G}_2)$ and $\mathcal{L}_g(\mathsf{G}_1.\mathsf{G}_2) = \mathcal{L}_g(\mathsf{G}_1).\mathcal{L}_g(\mathsf{G}_2),$
- $\mathcal{L}_g(\mathsf{G}^*) = \mathcal{L}_g(\mathsf{G})^*$, and $\mathcal{L}_g(\mathsf{G}^\omega) = \mathcal{L}_g(\mathsf{G})^\omega$.

Two generalized ω-regular expressions G and G' are called equivalent iff $\mathcal{L}_g(\mathsf{G}) = \mathcal{L}_g(\mathsf{G}')$.

Show that for each generalized ω-regular expression G there exists an equivalent generalized ω-regular expression G' of the form

$$\mathsf{G}' = \mathsf{E} + \mathsf{E}_1.\mathsf{F}_1^\omega + \ldots \mathsf{E}_n.\mathsf{F}_n^\omega$$

where $\mathsf{E}, \mathsf{E}_1, \ldots, \mathsf{E}_n, \mathsf{F}_1, \ldots, \mathsf{F}_n$ are regular expressions and $\varepsilon \notin \mathcal{L}(\mathsf{F}_i)$, $i = 1, \ldots, n$.

Exercise 4.9. Let $\Sigma = \{A, B\}$. Construct an NBA $\mathcal{A}$ that accepts the set of infinite words σ over Σ such that A occurs infinitely many times in σ and between any two successive A's an odd number of B's occur.

Exercise 4.10. Let $\Sigma = \{A, B, C\}$ be an alphabet.

(a) Construct an NBA $\mathcal{A}$ that accepts exactly the infinite words σ over Σ such that A occurs infinitely many times in σ and between any two successive A's an odd number of B's or an odd number of C's occur. Moreover, between any two successive A's either only B's or only C's are allowed. That is, the accepted words should have the form

$$wAv_1Av_2Av_3\ldots$$

where $w \in \{B, C\}^*$, $v_i \in \{B^{2k+1} \mid k \geqslant 0\} \cup \{C^{2k+1} \mid k \geqslant 0\}$ for all $i > 0$. Give also an ω-regular expression for this language.

(b) Repeat the previous exercise such that any accepting word contains only finitely many C's.

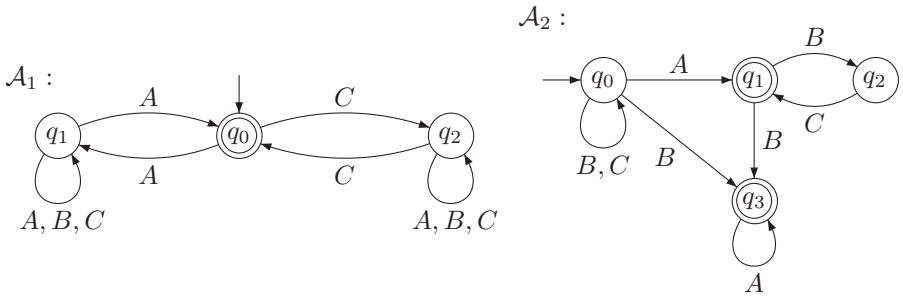
(c) Change your automaton from part (a) such that between any two successive A's an odd number of symbols from the set $\{B, C\}$ may occur.

(d) Same exercise as in (c), except that now an odd number of B's *and* an odd number of C's must occur between any two successive A symbols.

Exercise 4.11. Depict an NBA for the language described by the ω-regular expression

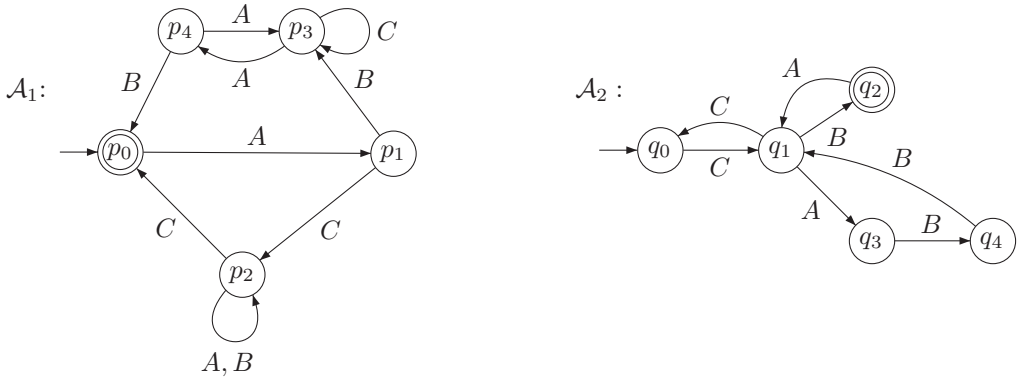
$$(AB + C)^*((AA + B)C)^\omega + (A^*C)^\omega.$$

Exercise 4.12. Consider the following NBA $\mathcal{A}_1$ and $\mathcal{A}_2$ over the alphabet $\{ A, B, C \}$:



Find ω-regular expressions for the languages accepted by $\mathcal{A}_1$ and $\mathcal{A}_2$.

Exercise 4.13. Consider the NFA $\mathcal{A}_1$ and $\mathcal{A}_2$:



Construct an NBA for the language $\mathcal{L}(\mathcal{A}_1).\mathcal{L}\,(\mathcal{A}_2)^{\omega}$.

Exercise 4.14. Let $AP = \{ a, b \}$. Give an NBA for the LT property consisting of the infinite words $A_0 A_1 A_2 \ldots \left(2^{AP}\right)^{\omega}$ such that

$$\overset{\infty}{\exists}\, j \geqslant 0.\, (a \in A_j \wedge b \in A_j) \quad \text{and} \quad \exists j \geqslant 0.\, (a \in A_j \wedge b \notin A_j).$$

Provide an ω-regular expression for $\mathcal{L}_{\omega}(\mathcal{A})$.

Exercise 4.15. Let $AP = \{a, b, c\}$. Depict an NBA for the LT property consisting of the infinite words $A_0 A_1 A_2 \ldots \left(2^{AP}\right)^{\omega}$ such that

$$\forall j \geqslant 0.\, A_{2j} \models (a \ \vee \ (b \wedge c))$$

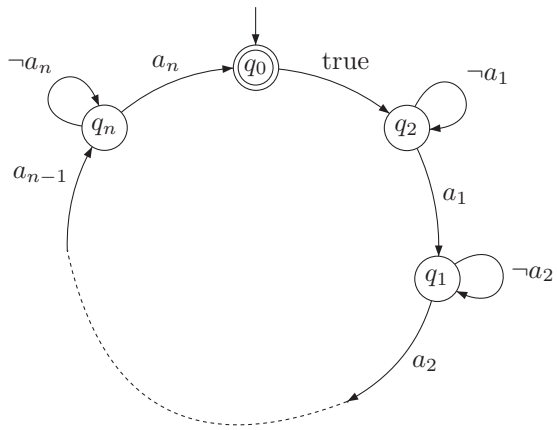
Recall that $A \models (a \ \vee \ (b \wedge c))$ means $a \in A$ or $\{b, c\} \subseteq A$, i.e., $A \in \{\{a\}, \{b, c\}, \{a, b, c\}\}$.

EXERCISE 4.16. Consider NBA $\mathcal{A}_1$ and $\mathcal{A}_2$ depicted in Figure 4.26. Show that the powerset construction applied to $\mathcal{A}_1$ and $\mathcal{A}_2$ (viewed as NFA) yields the same deterministic automaton, while $\mathcal{L}_\omega(\mathcal{A}_1) \neq \mathcal{L}_\omega(\mathcal{A}_2)$. (This exercise is taken from [408].)

(a) (b)

Figure 4.26: NBA $\mathcal{A}_1$ (a) and $\mathcal{A}_2$ (b).

EXERCISE 4.17. Consider the following NBA $\mathcal{A}$ with the alphabet $\Sigma = 2^{AP}$ where $AP = \{a_1, \ldots, a_n\}$ for $n > 0$.



(a) Determine the accepted language $\mathcal{L}_\omega(\mathcal{A})$.

(b) Show that there is no NBA $\mathcal{A}'$ with $\mathcal{L}_\omega(\mathcal{A}) = \mathcal{L}_\omega(\mathcal{A}')$ and less than n states.

(This exercise is inspired by [149].)

Exercise 4.18. Provide an example for a regular safety property P_{safe} over AP and an NFA $\mathcal{A}$ for its minimal bad prefixes such that

$$\mathcal{L}_\omega(\mathcal{A}) \;\neq\; \big(2^{AP}\big)^\omega \setminus P_{safe}$$

when $\mathcal{A}$ is viewed as an NBA.

Exercise 4.19. Provide an example for a liveness property that is not ω-regular. Justify your answer.

Exercise 4.20. Is there a DBA that accepts the language described by the ω-regular expression $(A+B)^*(AB+BA)^\omega$? Justify your answer.

Exercise 4.21. Provide an example for an ω-regular language $\mathcal{L} = \mathcal{L}_k$ that is recognizable for a DBA such that the following two conditions are satisfied:

(a) There exists an NBA $\mathcal{A}$ with $|\mathcal{A}| = \mathcal{O}(k)$ and $\mathcal{L}_\omega(\mathcal{A}) = \mathcal{L}$.

(b) Each DBA $\mathcal{A}'$ for $\mathcal{L}$ is of the size $|\mathcal{A}'| = \Omega(2^k)$.

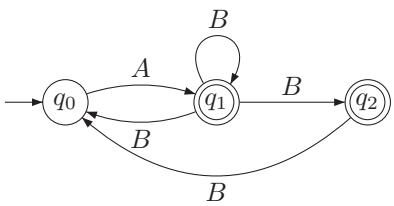
Hint: There is a simple answer to this question that uses the result that the regular language for the expression $(A+B)^*B(A+B)^k$ is recognizable by an NFA of size $\mathcal{O}(k)$, while any DFA has $\Omega(2^k)$ states.

Exercise 4.22. Show that the class of languages that are accepted by DBAs is not closed under complementation.

Exercise 4.23. Show that the class of languages that are accepted by DBAs is closed under union. To do so, prove the following stronger statement:

Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two DBAs both over the alphabet Σ. Show that there exists a DBA $\mathcal{A}$ with $|\mathcal{A}| = \mathcal{O}(|\mathcal{A}_1| \cdot |\mathcal{A}_2|)$ and $\mathcal{L}_\omega(\mathcal{A}) = \mathcal{L}_\omega(\mathcal{A}_1) \cup \mathcal{L}_\omega(\mathcal{A}_2)$.

Exercise 4.24.
Consider the GNBA outlined on the right with acceptance sets $F_1 = \{ q_1 \}$ and $F_2 = \{ q_2 \}$. Construct an equivalent NBA.

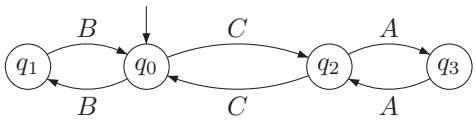


EXERCISE 4.25. Provide NBA $\mathcal{A}_1$ and $\mathcal{A}_2$ for the languages given by the expressions $(AC+B)^*B^\omega$ and $(B^*AC)^\omega$ and apply the product construction to obtain a GNBA $\mathcal{G}$ with $\mathcal{L}_\omega(\mathcal{G}) = \mathcal{L}_\omega(\mathcal{A}_1) \cap \mathcal{L}_\omega(\mathcal{A}_2)$. Justify that $\mathcal{L}_\omega(\mathcal{G}) = \varnothing$.

EXERCISE 4.26. A nondeterministic Muller automaton is a quintuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, \mathcal{F})$ where Q, Σ, δ, Q_0 are as for NBA and $\mathcal{F} \subseteq 2^Q$. For an infinite run ρ of $\mathcal{A}$, let $\lim(\rho) := \Big\{ q \in Q \mid \overset{\infty}{\exists}\, i \geq 0.\ \rho[i] = q \Big\}$. Let $\alpha \in \Sigma^\omega$.

$\mathcal{A}$ accepts $\alpha \iff$ ex. infinite run ρ of $\mathcal{A}$ on α s.t. $\lim(\rho) \in \mathcal{F}$

(a) Consider the following Muller automaton $\mathcal{A}$ with $\mathcal{F} = \{\{q_2, q_3\}, \{q_1, q_3\}, \{q_0, q_2\}\}$:



Define the language accepted by $\mathcal{A}$ by means of an ω-regular expression.

(b) Show that every GNBA $\mathcal{G}$ can be transformed into a nondeterministic Muller automaton $\mathcal{A}$ such that $\mathcal{L}_\omega(\mathcal{A}) = \mathcal{L}_\omega(\mathcal{G})$ by defining the corresponding transformation.

EXERCISE 4.27. Consider the transition systems TS_{Sem} and TS_{Pet} for mutual exclusion with a semaphore and the Peterson algorithm, respectively. Let P_{live} be the following ω-regular property over $AP = \{ wait_1, crit_1 \}$:

"whenever process 1 is in its waiting location then it will eventually enter its critical section"

(a) Depict an NBA for P_{live} and an NBA $\bar{\mathcal{A}}$ for the complement property $\bar{P}_{live} = \big(2^{AP}\big)^\omega \setminus P_{live}$.

(b) Show that $TS_{Sem} \not\models P_{live}$ by applying the techniques explained in Section 4.4:

(i) Depict the reachable fragment of the product $TS_{Sem} \otimes \bar{\mathcal{A}}$

(ii) Sketch the main steps of the nested depth-first search applied to $TS_{Sem} \otimes \bar{\mathcal{A}}$ for the persistence property "eventually forever $\neg F$" where F is the acceptance set of $\bar{\mathcal{A}}$. Which counterexample is returned by Algorithm ?8

(c) Apply now the same techniques (product construction, nested DFS) to show that $TS_{Pet} \models P_{live}$.

EXERCISE 4.28. The nested depth-first search approach can also be reformulated for an emptiness check for NBA. The path fragment returned by Algorithm 8 in case of a negative answer then yields a prefix of an accepting run.

Consider the automaton shown in Exercise 4.24 as an NBA, i.e., the acceptance set is $F = \{ q_1, q_2 \}$. Apply the nested depth-first search approach to verify that $\mathcal{L}_\omega(\mathcal{A}) \neq \varnothing$.

Chapter 5

Linear Temporal Logic

This chapter introduces (propositional) linear temporal logic (LTL), a logical formalism that is suited for specifying LT properties. The syntax and semantics of linear temporal logic are defined. Various examples are provided that show how linear temporal logic can be used to specify important system properties. The second part of the chapter is concerned with a model-checking algorithm—based on Büchi automata—for LTL. This algorithm can be used to answer the question: given a transition system TS and LTL-formula φ, how to check whether φ holds in TS?

5.1 Linear Temporal Logic

For reactive systems, correctness depends on the executions of the system—not only on the input and output of a computation—and on fairness issues. Temporal logic is a formalism par excellence for treating these aspects. Temporal logic extends propositional or predicate logic by modalities that permit to referral to the infinite behavior of a reactive system. They provide a very intuitive but mathematically precise notation for expressing properties about the relation between the state labels in executions, i.e., LT properties. Temporal logics and related modal logics have been studied in ancient times in different areas such as philosophy. Their application to verifying complex computer systems was proposed by Pnueli in the late seventies.

In this monograph, we will focus our attention on *propositional temporal logics*, i.e., extensions of propositional logic by temporal modalities. These logics should be distinguished from first- (or higher-) order temporal logics that impose temporal modalities on top of

predicate logic. Throughout this monograph we assume some familiarity with the basic principles of propositional logic. A brief introduction and summary of our notations can be found in Appendix A.3. The elementary temporal modalities that are present in most temporal logics include the operators:

$\Diamond$ "eventually" (eventually in the future)
$\Box$ "always" (now and forever in the future)

The underlying nature of time in temporal logics can be either *linear* or *branching*. In the linear view, at each moment in time there is a single successor moment, whereas in the branching view it has a branching, tree-like structure, where time may split into alternative courses. This chapter considers LTL (Linear Temporal Logic), a temporal logic that is based on a linear-time perspective. Chapter 6 introduces CTL (Computation Tree Logic), a logic that that is based on a branching-time view. Several model-checking tools use LTL (or a slight variant thereof) as a property specification language. The model checker SPIN is a prominent example of such an automated verification tool. One of the main advantages of LTL is that imposing fairness assumptions (such as strong and weak fairness) does not require the use of any new machinery: the typical fairness assumptions can all be specified in LTL. Verifying LTL-formulae under fairness constraints can be done using the algorithm for LTL. This does not apply to CTL.

Before introducing LTL in more detail, a short comment on the adjective "temporal" is in order to avoid any possible confusion. Although the term *temporal* suggests a relationship with the real-time behavior of a reactive system, this is only true in an abstract sense. A temporal logic allows for the specification of the relative *order* of events. Some examples are "the car stops once the driver pushes the brake", or "the message is received after it has been sent". It does however *not* support any means to refer to the precise timing of events. The fact that there is a minimal delay of at least 3 μs between braking and the actual halting of the car cannot be specified. In terms of transition systems, neither the duration of taking a transition nor state residence times can be specified using the elementary modalities of temporal logics. Instead, these modalities do allow for specifying the order in which state labels occur during an execution, or to assess that certain state labels occur infinitely often in a (or all) system execution. One might thus say that the modalities in temporal logic are *time-abstract*.

As will be discussed in this chapter, LTL may be used to express the timing for the class of synchronous systems in which all components proceed in a lock-step fashion. In this setting, a transition corresponds to the advance of a single time-unit. The underlying time domain is thus *discrete*, i.e., the present moment refers to the current state and the next moment corresponds to the immediate successor state. Stated differently, the system behavior is assumed to be observable at the time points $0, 1, 2, \ldots$. The treatment

of real-time constraints in asynchronous systems by means of a *continuous-time* domain will be discussed in Chapter 9 where a timed version of CTL, called Timed CTL, will be introduced. Table 5.1 summarizes the distinguishing features of the main temporal logics considered in this monograph.

logic	linear-time (path-based)	branching-time (state-based)	real-time requirements (continuous-time domain)
LTL	$\surd$		
CTL		$\surd$	
Timed CTL		$\surd$	$\surd$

Table 5.1: Classification of the temporal logics in this monograph.

5.1.1 Syntax

This subsection describes the syntactic rules according to which formulae in LTL can be constructed. The basic ingredients of LTL-formulae are atomic propositions (state labels $a \in AP$), the Boolean connectors like conjunction $\wedge$, and negation $\neg$, and two basic temporal modalities $\bigcirc$ (pronounced "next") and U (pronounced "until"). The atomic proposition $a \in AP$ stands for the state label a in a transition system. Typically, the atoms are assertions about the values of control variables (e.g., locations in program graphs) or the values of program variables such as "$x > 5$" or "$x \leqslant y$". The $\bigcirc$-modality is a unary prefix operator and requires a single LTL formula as argument. Formula $\bigcirc\,\varphi$ holds at the current moment, if φ holds in the next "step". The U-modality is a binary infix operator and requires two LTL formulae as argument. Formula $\varphi_1\,\mathsf{U}\,\varphi_2$ holds at the current moment, if there is some future moment for which φ_2 holds and φ_1 holds at all moments until that future moment.

Definition 5.1. Syntax of LTL

LTL formulae over the set AP of atomic proposition are formed according to the following grammar:[1]

$$\varphi ::= \text{true} \;\Big|\; a \;\Big|\; \varphi_1 \wedge \varphi_2 \;\Big|\; \neg\varphi \;\Big|\; \bigcirc\,\varphi \;\Big|\; \varphi_1\,\mathsf{U}\,\varphi_2$$

where $a \in AP$. ■

[1] The Backus Naur form (BNF) is used in a somewhat liberal way. More concretely, nonterminals are identified with derived words (formulae) and indices in the rules. Moreover, brackets will be used, e.g. in $a \wedge (b\,\mathsf{U}\,c)$, which are not shown in the grammar. Such simplified notations for grammars to determine the syntax of formulae of some logic (or terms of other calculi) are often called *abstract* syntax.

We mostly abstain from explicitly indicating the set AP of propositions as this follows either from the context or can be defined as the set of atomic propositions occurring in the LTL formula at hand.

The precedence order on the operators is as follows. The unary operators bind stronger than the binary ones. $\neg$ and $\bigcirc$ bind equally strong. The temporal operator U takes precedence over $\wedge$, $\vee$, and $\rightarrow$. Parentheses are omitted whenever appropriate, e.g., we write $\neg\varphi_1 \,\mathsf{U}\, \bigcirc\, \varphi_2$ instead of $(\neg\varphi_1) \,\mathsf{U}\, (\bigcirc\, \varphi_2)$. Operator U is right-associative, e.g., $\varphi_1 \,\mathsf{U}\, \varphi_2 \,\mathsf{U}\, \varphi_3$ stands for $\varphi_1 \,\mathsf{U}\, (\varphi_2 \,\mathsf{U}\, \varphi_3)$.

Using the Boolean connectives $\wedge$ and $\neg$, the full power of propositional logic is obtained. Other Boolean connectives such as disjunction $\vee$, implication $\rightarrow$, equivalence $\leftrightarrow$, and the parity (or: exclusive or) operator $\oplus$ can be derived as follows:

$$\begin{array}{lcl}
\varphi_1 \vee \varphi_2 & \stackrel{\text{def}}{=} & \neg(\neg\varphi_1 \wedge \neg\varphi_2) \\
\varphi_1 \rightarrow \varphi_2 & \stackrel{\text{def}}{=} & \neg\varphi_1 \;\vee\; \varphi_2 \\
\varphi_1 \leftrightarrow \varphi_2 & \stackrel{\text{def}}{=} & (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1) \\
\varphi_1 \oplus \varphi_2 & \stackrel{\text{def}}{=} & (\varphi_1 \wedge \neg\varphi_2) \;\vee\; (\varphi_2 \wedge \neg\varphi_1) \\
 & \vdots &
\end{array}$$

The until operator allows to derive the temporal modalities $\Diamond$ ("eventually", sometimes in the future) and $\Box$ ("always", from now on forever) as follows:

$$\Diamond\varphi \;\stackrel{\text{def}}{=}\; \text{true}\,\mathsf{U}\,\varphi \qquad\qquad \Box\varphi \;\stackrel{\text{def}}{=}\; \neg\Diamond\neg\varphi$$

As a result, the following intuitive meaning of $\Diamond$ and $\Box$ is obtained. $\Diamond\varphi$ ensures that φ will be true eventually in the future. $\Box\varphi$ is satisfied if and only if it is not the case that eventually $\neg\varphi$ holds. This is equivalent to the fact that φ holds from now on forever.

Figure 5.1 sketches the intuitive meaning of temporal modalities for the simple case in which the arguments of the modalities are just atomic propositions from $\{a, b\}$. On the left-hand side, some LTL formulae are indicated, whereas on the right hand side sequences of states (i.e., paths) are depicted.

By combining the temporal modalities $\Diamond$ and $\Box$, new temporal modalities are obtained. For instance, $\Box\Diamond a$ ("always eventually a") describes the (path) property stating that at any moment j there is a moment $i \geqslant j$ at which an a-state is visited. This thus amounts to assert that an a-state is visited infinitely often. The dual modality $\Diamond\Box a$ expresses that from some moment j on, only a-states are visited. So:

$$\begin{array}{ll}
\Box\Diamond\varphi & \text{"infinitely often } \varphi\text{"} \\
\Diamond\Box\varphi & \text{"eventually forever } \varphi\text{"}
\end{array}$$

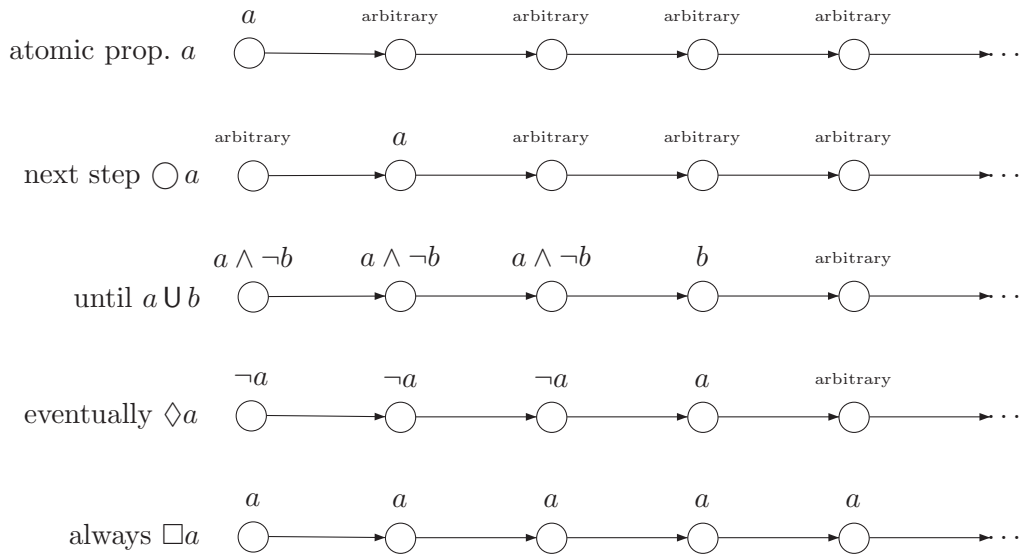


Figure 5.1: Intuitive semantics of temporal modalities.

Before proceeding with the formal semantics of LTL, we present some examples.

Example 5.2. Properties for the Mutual Exclusion Problem

Consider the mutual exclusion problem for two concurrent processes P_1 and P_2, say. Process P_i is modeled by three locations: (1) the noncritical section, (2) the waiting phase which is entered when the process intends to enter the critical section, and (3) the critical section. Let the propositions $wait_i$ and $crit_i$ denote that process P_i is in its waiting phase and critical section, respectively.

The safety property stating that P_1 and P_2 never simultaneously have access to their critical sections can be described by the LTL-formula:

$$\Box(\neg\, crit_1 \ \vee \ \neg\, crit_2).$$

This formula expresses that always ($\Box$) at least one of the two processes is not in its critical section ($\neg crit_i$).

The liveness requirement stating that each process P_i is infinitely often in its critical

section is described by the LTL formula:

$$(\Box\Diamond \mathit{crit}_1) \ \wedge \ (\Box\Diamond \mathit{crit}_2).$$

The weakened form that every waiting process will eventually enter its critical section (i.e., starvation freedom) can—by using the additional proposition wait_i—be formulated as follows:

$$(\Box\Diamond \mathit{wait}_1 \rightarrow \Box\Diamond \mathit{crit}_1) \ \wedge \ (\Box\Diamond \mathit{wait}_2 \rightarrow \Box\Diamond \mathit{crit}_2).$$

These formulae only refer to the locations (i.e., values of the program counters) by the atomic propositions wait_i and crit_i. Propositions can however also refer to program variables. For instance, for the solution of the mutual exclusion problem using a binary semaphore y, the formula:

$$\Box((y = 0) \rightarrow \mathit{crit}_1 \ \vee \ \mathit{crit}_2)$$

states that whenever the semaphore y has the value 0, one of the processes is in its critical section. ■

Example 5.3. Properties for the dining philosophers

For the dining philosophers (see Example 3.2 on page 90) deadlock freedom can be described by the LTL formula

$$\Box\neg\Big(\bigwedge_{0\leqslant i<n} \mathit{wait}_i \ \wedge \ \bigwedge_{0\leqslant i<n} \mathit{occupied}_i\Big).$$

We assume here that there are n philosophers and chop sticks, indexed from 0 to $n-1$. The atom wait_i means that philosopher i waits for one of the sticks on his left or right, but keeps the other one in his hand. Similarly, $\mathit{occupied}_i$ indicates that stick i is in use. ■

Example 5.4. Properties for a Traffic Light

For a traffic light with the phases "green", "red" and "yellow", the liveness property $\Box\Diamond \mathit{green}$ expresses that the traffic light is infinitely often green. A specification of the traffic light cycles and their chronological order can be provided by means of a conjunction of LTL-formulae stating the predecessor phase of any phase. For instance, the requirement "once red, the light cannot become green immediately" can be expressed by the LTL formula

$$\Box(\mathit{red} \rightarrow \neg \bigcirc \mathit{green}).$$

The requirement "once red, the light always becomes green eventually after being yellow for some time" is expressed by

$$\Box(\mathit{red} \rightarrow \bigcirc (\mathit{red} \,\mathsf{U}\, (\mathit{yellow} \ \wedge \ \bigcirc \ (\mathit{yellow} \,\mathsf{U}\, \mathit{green})))).$$

A progress property like "every request will eventually lead to a response" can be described by the following formula of the type

$$\Box(\mathit{request} \rightarrow \Diamond \mathit{response}).$$

■

Remark 5.5. Length of a Formula

Let $|\varphi|$ denote the length of LTL formula φ in terms of the number of operators in φ. This can easily be defined by induction on the structure of φ. For instance, the length of the formula true and $a \in AP$ is 0. Formulae $\bigcirc a \vee b$ and $a \vee \neg b$ have length 2, and $(\bigcirc a) \,\mathsf{U}\, (a \wedge \neg b)$ has length 4. Throughout this monograph, mostly the asymptotic size $\Theta(|\varphi|)$ is needed. For this purpose, it is irrelevant whether or not the derived Boolean operators $\vee$, $\rightarrow$, and so on, and the derived temporal modalities $\Diamond$ and $\Box$ are taken into account in determining the length. ■

5.1.2 Semantics

LTL formulae stand for properties of paths (or in fact their trace). This means that a path can either fulfill an LTL-formula or not. To precisely formulate when a path satisfies an LTL formula, we proceed as follows. First, the semantics of LTL formula φ is defined as a language $\mathit{Words}(\varphi)$ that contains all infinite words over the alphabet 2^{AP} that satisfy φ. That is, to every LTL formula a single LT property is associated. Then, the semantics is extended to an interpretation over paths and states of a transition system.

Definition 5.6. Semantics of LTL (Interpretation over Words)

Let φ be an LTL formula over AP. The LT property induced by φ is

$$\mathit{Words}(\varphi) \;=\; \Big\{\sigma \in (2^{AP})^{\omega} \mid \sigma \models \varphi\Big\}$$

where the satisfaction relation $\models \subseteq \; (2^{AP})^{\omega} \times \mathrm{LTL}$ is the smallest relation with the properties in Figure 5.2. ■

Here, for $\sigma = A_0\, A_1\, A_2 \ldots \in (2^{AP})^{\omega}$, $\sigma[j\ldots] = A_j\, A_{j+1}\, A_{j+2} \ldots$ is the suffix of σ starting in the $(j{+}1)$st symbol A_j.

Note that in the definition of the semantics of LTL-formulae the word fragment $\sigma[j\ldots]$ cannot be replaced with A_j. For the formula $\bigcirc\,(a \,\mathsf{U}\, b)$, e.g., the suffix $A_1\, A_2\, A_3 \ldots$ has to

$$
\begin{array}{lcll}
\sigma & \models & \text{true} & \\
\sigma & \models & a & \text{iff} \quad a \in A_0 \quad \text{(i.e., } A_0 \models a\text{)} \\
\sigma & \models & \varphi_1 \wedge \varphi_2 & \text{iff} \quad \sigma \models \varphi_1 \text{ and } \sigma \models \varphi_2 \\
\sigma & \models & \neg\, \varphi & \text{iff} \quad \sigma \not\models \varphi \\
\sigma & \models & \bigcirc \varphi & \text{iff} \quad \sigma[1\ldots] = A_1A_2A_3\ldots \models \varphi \\
\sigma & \models & \varphi_1 \,\mathsf{U}\, \varphi_2 & \text{iff} \quad \exists j \geqslant 0.\ \sigma[j\ldots] \models \varphi_2 \ \text{ and } \ \sigma[i\ldots] \models \varphi_1, \text{ for all } 0 \leqslant i < j
\end{array}
$$

Figure 5.2: LTL semantics (satisfaction relation $\models$) for infinite words over 2^{AP}.

be regarded in order to be able to refer to the truth-value of the subformula $a \,\mathsf{U}\, b$ in the "next step".

For the derived operators $\Diamond$ and $\Box$ the expected result is:

$$
\begin{array}{lcll}
\sigma & \models & \Diamond\varphi & \text{iff} \quad \exists j \geqslant 0.\ \sigma[j\ldots] \models \varphi \\
\sigma & \models & \Box\varphi & \text{iff} \quad \forall j \geqslant 0.\ \sigma[j\ldots] \models \varphi.
\end{array}
$$

The statement for $\Diamond$ is immediate from the definition of $\Diamond$ and the semantics of U. The statement for $\Box$ follows from:

$$
\begin{array}{lll}
\sigma \models \Box\varphi = \neg\Diamond\neg\varphi & \text{iff} & \neg\exists j \geqslant 0.\ \sigma[j\ldots] \models \neg\varphi \\
& \text{iff} & \neg\exists j \geqslant 0.\ \sigma[j\ldots] \not\models \varphi \\
& \text{iff} & \forall j \geqslant 0.\ \sigma[j\ldots] \models \varphi.
\end{array}
$$

The semantics of the combinations of $\Box$ and $\Diamond$ can now be derived:

$$
\begin{array}{lcll}
\sigma & \models & \Box\Diamond\varphi & \text{iff} \quad \overset{\infty}{\exists}\, j.\ \sigma[j\ldots] \models \varphi \\
\sigma & \models & \Diamond\Box\varphi & \text{iff} \quad \overset{\infty}{\forall}\, j.\ \sigma[j\ldots] \models \varphi.
\end{array}
$$

Here, $\overset{\infty}{\exists}\, j$ means $\forall i \geqslant 0.\ \exists j \geqslant i$, "for infinitely many $j \in \mathbb{N}$", while $\overset{\infty}{\forall}\, j$ stands for $\exists i \geqslant 0.\ \forall j \geqslant i$, "for almost all $j \in \mathbb{N}$". Let us verify the first statement. The argument for the second statement is similar.

$$
\begin{array}{lll}
\sigma \models \Box\Diamond\varphi & \text{iff} & \forall i \geqslant 0.\ \sigma[i\ldots] \models \Diamond\varphi \\
& \text{iff} & \forall i \geqslant 0.\ \exists j \geqslant i.\ \sigma[j\ldots] \models \varphi \\
& \text{iff} & \overset{\infty}{\exists}\, j.\ \sigma[j\ldots] \models \varphi.
\end{array}
$$

As a subsequent step, we determine the semantics of LTL-formulae with respect to a

transition system. According to the satisfaction relation for LT properties (see Definition 3.11 on page 100), the LTL formula φ holds in state s if all paths starting in s satisfy φ. The transition system TS satisfies φ if TS satisfies the LT property $Words(\varphi)$, i.e., if *all* initial paths of TS—paths starting in an initial state $s_0 \in I$—satisfy φ.

Recall that we may assume without loss of generality that transition system TS has no terminal states (if it has such states, a trap state can be introduced. Thus, we may assume that all paths and traces are infinite. This assumption is made for the sake of simplicity; it is also possible to define the semantics of LTL for finite paths. Note that for the semantics it is irrelevant whether or not TS is finite. Only for the model-checking algorithm later on in this chapter, is the finiteness of TS required.

As for the LT properties, when defining $TS \models \varphi$ for transition system TS over AP', it is assumed that φ is an LTL-formula with atomic propositions in $AP = AP'$. (Here, one could be more liberal and allow for $AP \subseteq AP'$.)

Definition 5.7. Semantics of LTL over Paths and States

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system without terminal states, and let φ be an LTL-formula over AP.

- For infinite path fragment π of TS, the satisfaction relation is defined by
$$\pi \models \varphi \quad \text{iff} \quad trace(\pi) \models \varphi.$$
- For state $s \in S$, the satisfaction relation $\models$ is defined by
$$s \models \varphi \quad \text{iff} \quad (\forall \pi \in Paths(s).\ \pi \models \varphi).$$
- TS satisfies φ, denoted $TS \models \varphi$, if $Traces(TS) \subseteq Words(\varphi)$.

■

From this definition, it immediately follows that

$TS \models \varphi$
iff (* Definition 5.7 *)
$Traces(TS) \subseteq Words(\varphi)$
iff (* Definition of $\models$ for LT properties *)
$TS \models Words(\varphi)$
iff (* Definition of $Words(\varphi)$ *)
$\pi \models \varphi$ for all $\pi \in Paths(TS)$
iff (* Definition 5.7 of $\models$ for states *)
$s_0 \models \varphi$ for all $s_0 \in I$.

Thus, $TS \models \varphi$ if and only if $s_0 \models \varphi$ for all initial states s_0 of TS.

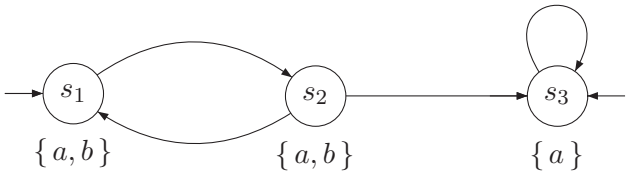


Figure 5.3: Example for semantics of LTL.

Example 5.8. Semantics of LTL

Consider the transition system TS depicted in Figure 5.3 with the set of propositions $AP = \{a, b\}$. For example, we have that $TS \models \Box a$, since all states are labeled with a, and hence, all traces of TS are words of the form $A_0 A_1 A_2 \ldots$ with $a \in A_i$ for all $i \geqslant 0$. Thus, $s_i \models \Box a$ for $i = 1, 2, 3$. Moreover:

$$s_1 \models \bigcirc (a \wedge b) \text{ since } s_2 \models a \wedge b \text{ and } s_2 \text{ is the only successor of } s_1$$

$$s_2 \not\models \bigcirc (a \wedge b) \text{ and } s_3 \not\models \bigcirc (a \wedge b) \text{ as } s_3 \in \mathit{Post}(s_2), s_3 \in \mathit{Post}(s_3) \text{ and } s_3 \not\models a \wedge b.$$

This yields $TS \not\models \bigcirc (a \wedge b)$ as s_3 is an initial state for which $s_3 \not\models \bigcirc (a \wedge b)$. As another example:

$$TS \models \Box(\neg b \rightarrow \Box(a \wedge \neg b)),$$

since s_3 is the only $\neg b$ state, s_3 cannot be left anymore, and $a \wedge \neg b$ in s_3 is true. However,

$$TS \not\models b \,\mathsf{U}\, (a \wedge \neg b),$$

since the initial path $(s_1 s_2)^\omega$ does not visit a state for which $a \wedge \neg b$ holds. Note that the initial path $(s_1 s_2)^* s_3^\omega$ satisfies $b \,\mathsf{U}\, (a \wedge \neg b)$. ■

Remark 5.9. Semantics of Negation

For paths, it holds $\pi \models \varphi$ if and only if $\pi \not\models \neg\varphi$. This is due to the fact that

$$\mathit{Words}(\neg\varphi) = (2^{AP})^\omega \setminus \mathit{Words}(\varphi).$$

However, the statements $TS \not\models \varphi$ and $TS \models \neg\varphi$ are *not* equivalent in general. Instead, we have $TS \models \neg\varphi$ implies $TS \not\models \varphi$. Note that

$$\begin{aligned} TS \not\models \varphi \quad & \text{iff } \mathit{Traces}(TS) \not\subseteq \mathit{Words}(\varphi) \\ & \text{iff } \mathit{Traces}(TS) \setminus \mathit{Words}(\varphi) \neq \varnothing \\ & \text{iff } \mathit{Traces}(TS) \cap \mathit{Words}(\neg\varphi) \neq \varnothing. \end{aligned}$$

Thus, it is possible that a transition system (or a state) satisfies neither φ nor $\neg\varphi$. This is caused by the fact that there might be paths π_1 and π_2 in TS such that $\pi_1 \models \varphi$ and $\pi_2 \models \neg\varphi$ (and therefore $\pi_2 \not\models \varphi$). In this case, $TS \not\models \varphi$ and $TS \not\models \neg\varphi$ holds.

To illustrate this effect, consider the transition system depicted in Figure 5.4. Let $AP = \{ a \}$. It follows that $TS \not\models \Diamond a$, since the initial path $s_0(s_2)^\omega \not\models \Diamond a$. On the other hand, $TS \not\models \neg\Diamond a$ also holds, since the initial path $s_0(s_1)^\omega \models \Diamond a$, and thus, $s_0(s_1)^\omega \not\models \neg\Diamond a$. ■

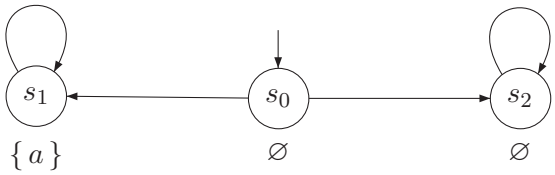


Figure 5.4: A transition system for which $TS \not\models \Diamond a$ and $TS \not\models \neg\Diamond a$.

5.1.3 Specifying Properties

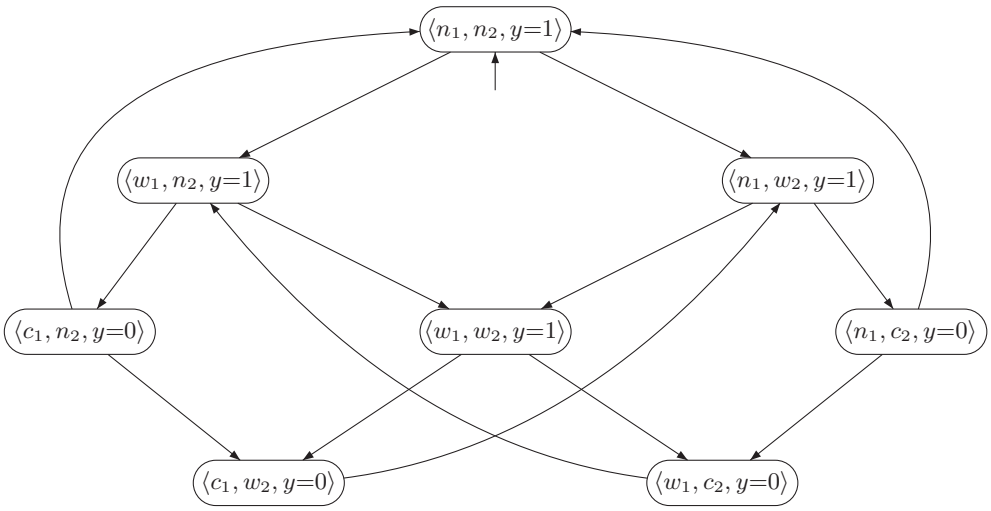


Figure 5.5: Transition system of semaphore-based mutual exclusion algorithm.

Example 5.10. Semaphore-Based Mutual Exclusion Revisited

Consider the transition system TS_{Sem} depicted in Figure 5.5 which represents a semaphore-based solution to the mutual exclusion problem; see also Example 3.9 on page 98. Each

state of the form $\langle c_1, \cdot, \cdot \rangle$ is labeled with proposition $crit_1$ and each state of the form $\langle \cdot, c_2, \cdot \rangle$ is labeled with $crit_2$. It follows that

$$TS_{Sem} \models \Box(\neg\, crit_1 \vee \neg\, crit_2) \quad \text{and} \quad TS_{Sem} \models \Box\Diamond crit_1 \vee \Box\Diamond crit_2,$$

where the first LTL-formula stands for the mutual exclusion property and the second LTL-formula for the fact that at least one of the two processes enters its critical section infinitely often. However,

$$TS_{Sem} \not\models \Box\Diamond crit_1 \wedge \Box\Diamond crit_2,$$

since—in the absence of any fairness assumption—it is not ensured that process P_1 is enabled infinitely often. It may not be able to acquire access to its critical section once. (A similar argument applies to process P_2.) The same argument applies to show that

$$TS_{Sem} \not\models \Box\Diamond wait_1 \rightarrow \Box\Diamond crit_1$$

as in principle process P_1 may not get its turn once it starts to wait. ■

Example 5.11. Modulo 4 Counter

A modulo 4 counter can be represented by a sequential circuit C, which outputs 1 in every fourth cycle, otherwise 0. C has no input bits, one output bit y, and two registers r_1 and r_2. The register evaluation $[r_1 = c_1, r_2 = c_2]$ can be identified with the number $i = 2 \cdot r_1 + r_2$. In every cycle, the value of i is increased by 1 (modulo 4). We construct C in a way such that the output bit y is set exactly for $i = 0$ (hence, $r_1 = r_2 = 0$). The transition relation and output function are given by

$$\delta_{r_1} = r_1 \oplus r_2, \quad \delta_{r_2} = \neg r_1, \quad \lambda_y = \neg r_1 \wedge \neg r_2.$$

Figure 5.6 illustrates the diagram (on the left) and the transition system TS_C (on the right). Let $AP = \{ r_1, r_2, y \}$. The following statement can be directly inferred from TS_C:

$$\begin{aligned} TS_C &\models \Box\, (\, y \leftrightarrow \neg r_1 \wedge \neg r_2\,) \\ TS_C &\models \Box\, (\, r_1 \rightarrow (\bigcirc y \vee \bigcirc \bigcirc y)\,) \\ TS_C &\models \Box\, (\, y \rightarrow (\bigcirc \neg y \wedge \bigcirc \bigcirc \neg y)\,) \end{aligned}$$

If it is assumed that only the output variable y (and not the register evaluations) can be perceived by an observer, then an appropriate choice for AP is $AP = \{ y \}$. The property that at least during every four cycles the output 1 is obtained holds for TS_C, i.e., we have

$$TS_C \models \Box\, (\, y \vee \bigcirc y \vee \bigcirc \bigcirc y \vee \bigcirc \bigcirc \bigcirc y\,).$$

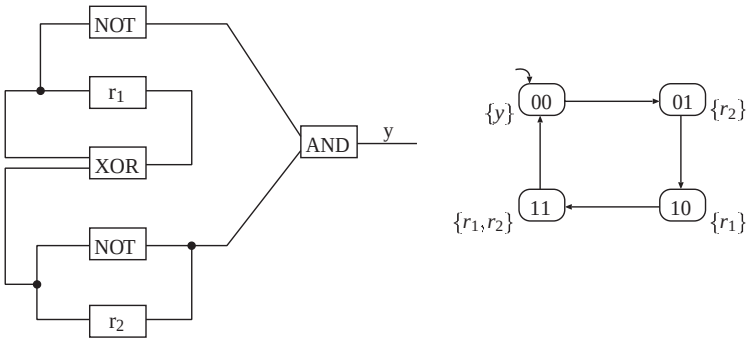


Figure 5.6: A modulo 4 counter.

The fact that these outputs are produced in a periodic manner where every fourth cycle yields the output 1 is expressed as

$$TS_C \models \Box\,(\, y \longrightarrow (\bigcirc\neg y \wedge \bigcirc\bigcirc\neg y \wedge \bigcirc\bigcirc\bigcirc\neg y)\,).$$

■

Example 5.12. A Communication Channel

Consider an unidirectional channel between two communicating processes, a sender S and a receiver R. Sender S is equipped with an output buffer $S.out$ and recipient R with an input buffer $R.in$. If sender S sends a message m to R it inserts the message into its output buffer $S.out$. The output buffer $S.out$ and the input buffer $R.in$ are connected via an unidirectional channel. The receiver R receives messages by deleting messages from its input buffer $R.in$. The capacity of the buffers is not of importance here.

A schematic view of the system under consideration is:

In the following LTL-specifications, we use the atoms "$m \in S.out$" and "$m \in R.in$" where m is an arbitrary message. We formalize the following informal requirements by LTL formulae:

- "Whenever message m is in the out-buffer of S, then m will eventually be consumed by the receiver."

$$\Box(m \in S.out \longrightarrow \Diamond(m \in R.in))$$

The above property is still satisfied for paths $s_1 s_2 s_3 \ldots$ where $s_1 \models m \in S.out$, $s_2 \models m \notin S.out$, $s_2 \models m \notin R.in$, and $s_3 \models m \in R.in$. However, such paths stand for a mysterious behavior where message m in the output buffer for S (state s_1) gets lost (state s_2), but still arrives in the input buffer of R (state s_3). In fact, such a behavior is impossible for a reliable FIFO channel which satisfies the following stronger condition

$$\Box(m \in S.out \longrightarrow (m \in S.out \,\mathsf{U}\, m \in R.in))$$

stating that message m stays in $S.out$ until the receiver R consumes m. Since writing and reading in a FIFO channel cannot happen at the same moment, we can even use the formula

$$\Box(m \in S.out \longrightarrow \bigcirc\,(m \in S.out \,\mathsf{U}\, m \in R.in)).$$

- If we assume that no message occurs twice in $S.out$ then the asynchronous behavior of a FIFO channel ensures that the property "message m cannot be in both buffers at the same time". This is formalized by the LTL formula:

$$\Box\,\neg\,(m \in S.out \;\wedge\; m \in R.in).$$

- The characteristic of FIFO-channels is that they are "order-preserving" according to the "first in, first out principle" stating that if message m is offered first by S to its output buffer $S.out$ and subsequently m', then m will be received by R before m':

$$\begin{aligned}\Box\Big(&m \in S.out \;\wedge\; \neg\, m' \in S.out \;\wedge\; \Diamond(m' \in S.out)\\ &\longrightarrow \Diamond\,(m \in R.in \;\wedge\; \neg\, m' \in R.in \;\wedge\; \Diamond(m' \in R.in))\Big).\end{aligned}$$

Note that in the premise the conjunct $\neg\, m' \in S.out$ is needed in order to specify that m' is put in $S.out$ after m. $\Diamond\,(m' \in S.out)$ on its own does not exclude that m' is already in the sender's buffer when message m is in $S.out$.

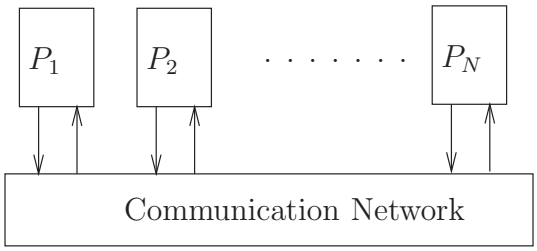
The above formulae refer to fixed messages m and m'. In order to state the above properties for all messages, we have to take the conjunction over all messages m, m'. As long as the message alphabet is finite we still obtain an LTL formula. ■

Example 5.13. Dynamic Leader Election

(This example has been taken from [69].) In current distributed systems several services are offered by some dedicated process(es) in the system. Consider, for example, address assignment and registration, query coordination in a distributed database system,

clock distribution, token regeneration after token loss in a token ring network, initiation of topology updates in a mobile network, load balancing, and so forth. Usually many processes in the system are potentially capable of providing these services. However, for consistency reasons it is usually the case that at any time only one process is allowed to actually provide a given service. This process – called the "leader" – is in fact elected. Sometimes it suffices to elect an arbitrary process, but for other services it is important to elect the process with the best capabilities for performing that service. Here we abstract from specific capabilities and use ranking on the basis of process identities. The idea is therefore that the higher the process' identity, the better its capabilities.

Assume we have a finite number $N > 0$ of processes connected via some communication means. The communication between processes is asynchronous, as in the previous example. Pictorially,



Each process has a unique identity, and it is assumed that a total ordering exists on these identities. Processes behave dynamically in the sense that they are initially inactive, i.e., not participating in the election, and may become active, i.e., participating in the election, at arbitrary moments. In order to have some progress we assume that a process cannot be inactive indefinitely; that is, each process becomes active at some time. (This corresponds to a fairness condition.) Once a process participates it continues to do so, i.e., it does not become inactive anymore. For a given set of active processes a leader will be elected; if an inactive process becomes active, a new election takes place if this process has a higher identity than the current leader.

To give an idea of using LTL as specification formalism we formulate several properties by LTL formulae. We will use i, j as process identities. Let the set of atomic propositions be $\{\, \mathit{leader}_i, \mathit{active}_i \mid 1 \leqslant i, j \leqslant N \,\}$, where leader_i means that process i is a leader, active_i means that process i is active. An inactive process cannot be a leader.

- The property "There is always one leader" can be formalized by

$$\Box\Big(\bigvee_{1 \leqslant i \leqslant N} \mathit{leader}_i \wedge \bigwedge_{\substack{1 \leqslant j \leqslant N \\ j \neq i}} \neg\, \mathit{leader}_j \Big).$$

Although this formula expresses the informally stated property, it will not be satisfied by any realistic protocol. One reason is that processes may be initially inactive, and thus no leader is guaranteed to exist initially. Besides, in a distributed system with asynchronous communication, switching from one leader to another can hardly be made atomic. So, it is more realistic to allow the temporary absence of a leader. As a first attempt to do so, one could modify the above formula into

$$\varphi \;=\; \Box\Diamond\Big(\bigvee_{1\leqslant i\leqslant N} \mathit{leader}_i \wedge \bigwedge_{\substack{1\leqslant j\leqslant N\\ j\neq i}} \neg\, \mathit{leader}_j\Big).$$

Problematic, though, is that this allows there to be more than one leader at a time temporarily – it is only stated that infinitely often there should be exactly one leader, but no statement is made about the moments at which this is not the case. For consistency reasons this is not desired. We therefore replace the above formula φ with $\varphi_1 \wedge \varphi_2$ where φ_1 and φ_2 correspond to the following two properties.

- "There must always be at most one leader":

$$\varphi_1 \;=\; \Box \bigwedge_{1\leqslant i\leqslant N} \Big(\mathit{leader}_i \rightarrow \bigwedge_{\substack{1\leqslant j\leqslant N\\ j\neq i}} \neg\, \mathit{leader}_j\Big)$$

- "There will be enough leaders in due time":

$$\varphi_2 \;=\; \Box\Diamond \bigvee_{1\leqslant i\leqslant N} \mathit{leader}_i$$

  φ_2 does not imply that there will be infinitely many leaders. It only states that there are infinitely many states at which a leader exists. This requirement classifies a leader election protocol that never elects a leader to be wrong. In fact, such a protocol would fulfill the previous requirement, but is not desired for obvious reasons.

- "In the presence of an active process with a higher identity the leader will resign at some time":

$$\Box\Big(\bigwedge_{\substack{1\leqslant i,j\leqslant N\\ i<j}} ((\mathit{leader}_i \;\wedge\; \neg\, \mathit{leader}_j \;\wedge\; \mathit{active}_j) \rightarrow \Diamond\, \neg\, \mathit{leader}_i)\Big)$$

  For reasons of efficiency it is assumed not to be desirable that a leader eventually resigns in the presence of an inactive process that may participate at some unknown time in the future. Therefore we require j to be an active process.

- "A new leader will be an improvement over the previous one". This property requires that successive leaders have an increasing identity. In particular, a process that resigns once will not become a leader anymore.

$$\Box\Big(\bigwedge_{1\leqslant i,j\leqslant N}(\mathit{leader}_i \ \wedge \ \neg \bigcirc \mathit{leader}_i \ \wedge \ \bigcirc \Diamond \mathit{leader}_j) \rightarrow (i<j)\Big)$$

  Here, we use "$i < j$" as an atomic proposition that compares the identifiers of processes P_i and P_j and evaluates to true if and only if the process identifier of P_i is smaller than that of P_j. Assuming that the identity of P_i is i (for $i = 1, \ldots, N$), the above property can also be specified by the LTL formula

$$\Box \neg \Big(\bigwedge_{\substack{1\leqslant i,j\leqslant N\\ i\geqslant j}}(\mathit{leader}_i \ \wedge \ \neg \bigcirc \mathit{leader}_i \ \wedge \ \bigcirc \Diamond \mathit{leader}_j)\Big).$$

■

Example 5.14. Specifying the Input/Output Behavior of Sequential Programs

The typical requirements on sequential programs such as partial correctness and termination can "in principle" be represented in LTL. Let us briefly describe what termination and partial correctness mean. Assume that a sequential program *Prog* computes a function of the type $f : \mathit{Inp} \rightarrow \mathit{Outp}$, i.e., *Prog* takes as input a value $i \in \mathit{Inp}$ and terminates either by reporting an output value $o \in \mathit{Outp}$ or does not terminate. *Prog* is called *terminating* if the computation of *Prog* halts for each input value $i \in \mathit{Inp}$. *Prog* is *partially correct* if for any input value $i \in \mathit{Inp}$, whenever *Prog* terminates then the output value o equals $f(i)$. How can termination and partial correctness be expressed by means of LTL formulae?

Termination can be specified by a formula of the form $\mathit{init} \ \rightarrow \ \Diamond \mathit{halt}$ where *init* is the labeling for the initial states and *halt* is an atomic proposition characterizing exactly those states that stand for termination. (Without loss of generality it can be assumed that transition systems have no terminal states, i.e., this means terminating states either are equipped with a self-loop, or have a transition leading to a *trap*-state with a self-loop and no other outgoing transitions.)

Partial correctness can be represented by a formula of the form

$$\Box(\ \mathit{halt} \ \longrightarrow \ \Diamond(y = f(x)))$$

where y is the output variable and x the input variable which is assumed not to change during program execution. Additional initial conditions such as expressed by the formula *init* can be added as premise as follows:

$$\mathit{init} \ \longrightarrow \ \Box(\mathit{halt} \ \longrightarrow \ \Diamond(y = f(x))).$$

It should be stressed that this is an extremely simplified representation. In practice, predicate logic concepts are needed to precisely formulate partial correctness. And even in cases where propositional logic formulae of the above form can be used to exactly describe termination and partial correctness, the algorithmic proof of the LTL formulae is very difficult or even impossible. (Recall the undecidability of the halting problem.) ■

Remark 5.15. Specifying Timed Properties with LTL for Synchronous Systems

For *synchronous* systems, LTL can be used as a formalism to specify "real-time" properties that refer to a discrete time scale. Recall that in synchronous systems, the involved processes proceed in a lock step fashion, i.e., at each discrete time instance each process performs a (sometimes idle) step. In this kind of system, the next-step operator $\bigcirc$ has a "timed" interpretation: $\bigcirc\varphi$ states that "at the next time instant φ holds". By putting applications of $\bigcirc$ in sequence, we obtain, e.g.:

$$\bigcirc^k \varphi \stackrel{\text{def}}{=} \underbrace{\bigcirc \bigcirc \ldots \bigcirc}_{k\text{-times}} \varphi \qquad \text{"}\varphi \text{ holds after (exactly) } k \text{ time instants"}.$$

Assertions like "φ will hold within at most k time instants" are obtained by

$$\Diamond^{\leqslant k} \varphi \;=\; \bigvee_{0 \leqslant i \leqslant k} \bigcirc^i \varphi.$$

Statements like "φ holds now and will hold during the next k instants" can be represented as follows:

$$\Box^{\leqslant k}\varphi \;=\; \neg \Diamond^{\leqslant k} \neg \varphi \;=\; \neg \bigvee_{0 \leqslant i \leqslant k} \bigcirc^i \neg \varphi.$$

For the modulo 4 counter of Example 5.11 (page 240) we in fact already implicitly used LTL-formulae as real-time specifications. For example, the formula expressing that once the output is y=1, the next three steps the output is y=0:

$$\Box(\, y \;\longrightarrow\; (\bigcirc \neg y \;\wedge\; \bigcirc \bigcirc\; \neg y \;\wedge\; \bigcirc \bigcirc \bigcirc \neg y)\;)$$

can be abbreviated as $\Box(y \;\longrightarrow\; \bigcirc \Box^{\leqslant 2}\; \neg y)$.

It should, however, be noted that the temporal interpretation of the next-step operator is only appropriate for synchronous systems. Every transition in these systems represents the cumulative effect of the actions possible within a single time instant. For asynchronous systems (for which the transition system representation is time-abstract), the next-step operator cannot be interpreted as a real-time modality. In fact, for asynchronous systems the next-step operator should be used with care. The phase changes of a traffic light, for example, can be described by

$$\varphi \;=\; \Box(\mathit{green} \rightarrow \bigcirc \mathit{yellow}) \;\wedge\; \Box(\mathit{yellow} \rightarrow \bigcirc \mathit{red}) \;\wedge\; \ldots$$

For the interleaving of two independent traffic lights (Example 2.17 on page 36) and the formulae φ_1, φ_2 (where the indexed atomic propositions $green_i$, $yellow_i$, etc., are used),

$$TrLight_1 \,|||\, TrLight_2 \not\models \varphi_1 \wedge \varphi_2.$$

This stems from the fact that, e.g., the first traffic light does not change its location when the second traffic light changes its phase. To avoid this problem, the until operator can be used instead, e.g.,

$$\varphi' \;=\; \Box(green \rightarrow (green \,\mathsf{U}\, yellow)) \;\wedge\; \Box(yellow \rightarrow (yellow \,\mathsf{U}\, red)) \;\wedge\; \ldots$$

This differs from the synchronous product operator $\otimes$, where

$$TrLight_1 \otimes TrLight_2 \models \varphi.$$

■

Remark 5.16. Other Notations and Variants of LTL

Many variants and notations have been introduced for LTL. Alternative notations for the temporal modalities are X for $\bigcirc$ (ne*X*t), F for $\Diamond$ (*F*inally), and G for $\Box$ (*G*lobally). All operators from LTL refer to the future (including the current state). Consequently, operators are known as future operators. LTL can, however, also be extended with *past* operators. This can be useful for specifying some properties more easily (and succinctly) in terms of the past than in terms of the future. For instance, $\Box^{-1}\, a$ ("always in the past") means that a is valid now and in any state in the past. $\Diamond^{-1}\, a$ ("sometime in the past") means that either a is valid in the current state or in some state in the past and $\bigcirc^{-1}\, a$ means that a holds in the previous state, provided such state exists. For example, the property "every red light phase is preceded by a yellow one" can be described by

$$\Box(red \;\rightarrow\; \bigcirc^{-1}\, yellow).$$

The main reason for introducing past operators is to simplify the specification of several properties. The expressive power of the logic is, however, not affected by the addition of past operators when a discrete notion of time is taken (as we do). Thus, for any property which contains one or more past operators, an LTL-formula with only future temporal operators exists expressing the same thing. More information is described in Section 5.4.

■

5.1.4 Equivalence of LTL Formulae

For any type of logic, a clear separation between syntax and semantics is an essential aspect. On the other hand, two formulae are intuitively identified whenever they have the

duality law	*idempotency law*
$\neg \bigcirc \varphi \equiv \bigcirc \neg\varphi$	$\Diamond\Diamond\varphi \equiv \Diamond\varphi$
$\neg\Diamond\varphi \equiv \Box\neg\varphi$	$\Box\Box\varphi \equiv \Box\varphi$
$\neg\Box\varphi \equiv \Diamond\neg\varphi$	$\varphi \,\mathsf{U}\, (\varphi \,\mathsf{U}\, \psi) \equiv \varphi \,\mathsf{U}\, \psi$
	$(\varphi \,\mathsf{U}\, \psi) \,\mathsf{U}\, \psi \equiv \varphi \,\mathsf{U}\, \psi$
absorption law	*expansion law*
$\Diamond\Box\Diamond\varphi \equiv \Box\Diamond\varphi$	$\varphi \,\mathsf{U}\, \psi \equiv \psi \vee (\varphi \wedge \bigcirc(\varphi \,\mathsf{U}\, \psi))$
$\Box\Diamond\Box\varphi \equiv \Diamond\Box\varphi$	$\Diamond\psi \equiv \psi \vee \bigcirc\Diamond\psi$
	$\Box\psi \equiv \psi \wedge \bigcirc\Box\psi$
distributive law	
$\bigcirc(\varphi \,\mathsf{U}\, \psi) \equiv (\bigcirc\varphi) \,\mathsf{U}\, (\bigcirc\psi)$	
$\Diamond(\varphi \vee \psi) \equiv \Diamond\varphi \vee \Diamond\psi$	
$\Box(\varphi \wedge \psi) \equiv \Box\varphi \wedge \Box\psi$	

Figure 5.7: Some equivalence rules for LTL.

same truth-value under all interpretations. For example, it seems useless to distinguish between $\neg\neg a$ and a, although both formulae are syntactically different.

Definition 5.17. Equivalence of LTL Formulae

LTL formulae φ_1, φ_2 are *equivalent*, denoted $\varphi_1 \equiv \varphi_2$, if $\mathit{Words}(\varphi_1) = \mathit{Words}(\varphi_2)$. ■

As LTL subsumes propositional logic, equivalences of propositional logic also hold for LTL, e.g., $\neg\neg\varphi \equiv \varphi$ and $\varphi \wedge \varphi \equiv \varphi$. In addition, there exist a number of equivalence rules for temporal modalities. They include the equivalence laws indicated in Figure 5.7. We explain some of these equivalence laws. The *duality rule* $\neg \bigcirc \varphi \equiv \bigcirc \neg\varphi$ shows that the next-step operator $\bigcirc$ is dual to itself. It results from the observation that

$$
\begin{array}{ll}
 & A_0\, A_1\, A_2 \ldots \models \neg \bigcirc \varphi \\
\text{iff} & A_0\, A_1\, A_2 \ldots \not\models \bigcirc \varphi \\
\text{iff} & A_1\, A_2 \ldots \not\models \varphi \\
\text{iff} & A_1\, A_2 \ldots \models \neg\varphi \\
\text{iff} & A_0\, A_1\, A_2 \ldots \models \bigcirc \neg\varphi.
\end{array}
$$

The first *absorption law* is explained by the fact that "infinitely often φ" is equal to "from

a certain point of time on, φ is true infinitely often".

The distributive laws for $\Diamond$ and disjunction, or $\Box$ and conjunction, respectively, are dual to each other. They can be regarded as the temporal logic analogon to the distributive laws for $\exists$ and $\vee$ or $\forall$ and $\wedge$ in predicate logic. It should be noted, however, that $\Diamond$ does not distribute over conjunction (as existential quantification), and $\Box$ does not distribute over disjunction (like universal quantification):

$$\Diamond(a \wedge b) \not\equiv \Diamond a \wedge \Diamond b \quad \text{and} \quad \Box(a \vee b) \not\equiv \Box a \vee \Box b.$$

The formula $\Diamond(a \wedge b)$ ensures that a state will be reached for which a and b hold, while $\Diamond a \wedge \Diamond b$ ensures that eventually an a-state and eventually a b-state will be reached. According to the latter formula, a and b need not to be satisfied at the same time. Figure 5.8 depicts a transition system that satisfies $\Diamond a \wedge \Diamond b$, but not $\Diamond(a \wedge b)$.

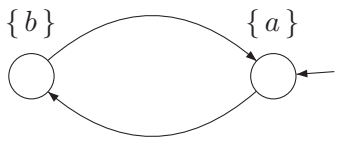


Figure 5.8: $TS \not\models \Diamond(a \wedge b)$ and $TS \models \Diamond a \wedge \Diamond b$.

The expansion laws play an important role. They describe the temporal modalities U, $\Diamond$, and $\Box$ by means of a *recursive* equivalence. These equivalences all have the same global structure: they assert something about the current state, and about the direct successor state. The assertion about the current state is done without the need to use temporal modalities whereas the assertion about the next state is done using the $\bigcirc$ operator. The expansion law for until, for instance, can be considered as follows. Formula $\phi \,\mathsf{U}\, \psi$ is a solution of the equivalence

$$\kappa \equiv \psi \vee (\varphi \wedge \bigcirc \kappa).$$

current state — first suffix

Let us explain the expansion law for the until operator in more detail. Let $A_0\, A_1\, A_2 \ldots$ be an infinite word over the alphabet 2^{AP}, such that $A_0\, A_1\, A_2 \ldots \models \varphi \,\mathsf{U}\, \psi$. By the definition of "until", there exists a $k \geqslant 0$, such that

$$A_i\, A_{i+1}\, A_{i+2} \ldots \models \varphi, \quad \text{for all } 0 \leqslant i < k \quad \text{and} \quad A_k\, A_{k+1}\, A_{k+2} \ldots \models \psi.$$

Distinguish between $k = 0$ and $k > 0$. If $k = 0$, then $A_0\, A_1\, A_2 \ldots \models \psi$ and thus $A_0\, A_1\, A_2 \ldots \models \psi \vee \ldots$. If $k > 0$, then

$$A_0\,A_1\,A_2\ldots \models \varphi \quad \text{and} \quad A_1\,A_2\ldots \models \varphi \,\mathsf{U}\, \psi.$$

From this it immediately follows that

$$A_0\,A_1\,A_2\ldots \models \varphi \;\wedge\; \bigcirc\,(\varphi \,\mathsf{U}\, \psi).$$

Gathering the results for $k = 0$ and $k > 0$ yields

$$A_0\,A_1\,A_2\ldots \models \psi \;\vee (\varphi \;\wedge\; \bigcirc\,(\varphi \,\mathsf{U}\, \psi)).$$

For the reverse direction, a similar argument can be provided.

The expansion law for $\Diamond\psi$ is a special case of the expansion law for until:

$$\Diamond\psi \;=\; \text{true}\,\mathsf{U}\,\psi \;\equiv\; \psi \;\vee\; \underbrace{(\text{true} \;\wedge\; \bigcirc\,(\text{true}\,\mathsf{U}\,\psi))}_{\equiv \bigcirc(\text{true}\,\mathsf{U}\,\psi) = \bigcirc\Diamond\psi} \;\equiv\; \psi \;\vee\; \bigcirc\Diamond\psi.$$

The expansion law for $\Box\psi$ now results from the duality of $\Diamond$ and $\Box$, the duality of $\vee$ and $\wedge$ (i.e., de Morgan's law) and from the duality law for $\bigcirc$

$$\begin{array}{lll} & \Box\psi & \\ = & & (\text{* definition of } \Box \text{ *}) \\ & \neg\Diamond\neg\psi & \\ \equiv & & (\text{* expansion law for } \Diamond\text{*}) \\ & \neg(\neg\psi \;\vee\; \bigcirc\Diamond\neg\psi) & \\ \equiv & & (\text{* de Morgan's law *}) \\ & \neg\neg\psi \;\wedge\; \neg\bigcirc\;\Diamond\neg\psi & \\ \equiv & & (\text{* self-duality of } \bigcirc\text{*}) \\ & \psi \;\wedge\; \bigcirc\neg\Diamond\neg\psi & \\ \equiv & & (\text{* definition of } \Box \text{ *}) \\ & \psi \;\wedge\; \bigcirc\Box\psi & \end{array}$$

It is important to realize that none of the indicated expansion laws represents a complete recursive characterization of the temporal operator at hand. For example, the formulae $\varphi = \text{false}$ and $\varphi = \Box a$ both satisfy the recursive "equation" $\varphi \;\equiv\; a \wedge \bigcirc\varphi$ since false $\equiv$ $a \wedge \bigcirc\,\text{false}$ and $\Box a \;\equiv\; a \wedge \bigcirc\Box a$. However, $\varphi \,\mathsf{U}\, \psi$ and $\Diamond\varphi$ are the *least* solutions of the expansion law indicated for "until" and"eventually", respectively. Similarly, $\Box\varphi$ is the *greatest* solution of the expansion law for "always". The precise meaning of these statements will be explained by considering the until operator as example:

Lemma 5.18. Until is the Least Solution of the Expansion Law

For LTL formulae φ and ψ, $Words(\varphi \,\mathsf{U}\, \psi)$ is the least LT property $P \subseteq (2^{AP})^\omega$ such that:

$$Words(\psi) \cup \{A_0\, A_1\, A_2 \ldots \in Words(\varphi) \;\mid\; A_1\, A_2 \ldots \in P\} \;\subseteq\; P \qquad (*)$$

Moreover, $Words(\varphi \,\mathsf{U}\, \psi)$ agrees with the set

$$Words(\psi) \cup \{A_0\, A_1\, A_2 \ldots \in Words(\varphi) \;\mid\; A_1\, A_2 \ldots \in Words(\varphi \,\mathsf{U}\, \psi)\}.$$

The formulation "least LT property satisfying condition (*)" means that the following conditions hold:

(1) $P = Words(\varphi \,\mathsf{U}\, \psi)$ satisfies (*).

(2) $Words(\varphi \,\mathsf{U}\, \psi) \subseteq P$ for all LT properties P satisfying condition (*).

Proof: Condition (1) follows immediately from the expansion law $\varphi \,\mathsf{U}\, \psi \;\equiv\; \psi \vee (\varphi \wedge \bigcirc(\varphi \,\mathsf{U}\, \psi))$. In fact, the expansion law even yields that $\subseteq$ in (*) can be replaced by equality, i.e., $Words(\varphi \,\mathsf{U}\, \psi)$ agrees with

$$Words(\psi) \cup \{A_0\, A_1\, A_2 \ldots \in Words(\varphi) \;\mid\; A_1\, A_2 \ldots \in Words(\varphi \,\mathsf{U}\, \psi)\}.$$

To prove condition (2), P is assumed to be an LT property that satisfies (*). We show that $Words(\varphi \,\mathsf{U}\, \psi) \subseteq P$. Since P fulfills (*), we have:

(i) $Words(\psi) \subseteq P$,

(ii) If $B_0\, B_1\, B_2 \ldots \in Words(\varphi)$ and $B_1\, B_2 \ldots \in P$ then $B_0\, B_1\, B_2 \ldots \in P$.

Let $A_0\, A_1\, A_2 \ldots \in Words(\varphi \,\mathsf{U}\, \psi)$. Then, there exists an index $k \geqslant 0$ such that

(iii) $A_i\, A_{i+1}\, A_{i+2} \ldots \in Words(\varphi)$, for all $0 \leqslant i < k$,

(iv) $A_k\, A_{k+1}\, A_{k+2} \ldots \in Words(\psi)$.

We now derive

$$
\begin{array}{lll}
 & A_k\, A_{k+1}\, A_{k+2}\, A_{k+3} \ldots \in P & \text{due to (iv) and (i)} \\
\Longrightarrow & A_{k-1}\, A_k\, A_{k+1}\, A_{k+2} \ldots \in P & \text{due to (ii) and (iii)} \\
\Longrightarrow & A_{k-2}\, A_{k-1}\, A_k\, A_{k+1} \ldots \in P & \text{due to (ii) and (iii)} \\
 & \vdots & \\
\Longrightarrow & A_0\, A_1\, A_2\, A_3 \ldots \in P & \text{due to (ii) and (iii).}
\end{array}
$$

■

5.1.5 Weak Until, Release, and Positive Normal Form

Any LTL formula can be transformed into a canonical form, the so-called *positive normal form* (PNF). This canonical form is characterized by the fact that negations only occur adjacent to atomic propositions. PNF formulae in propositional logic are constructed from the constants true and false, the literals a and $\neg a$, and the operators $\wedge$ and $\vee$. For instance, $\neg a \wedge ((\neg b \wedge c) \vee \neg a)$ is in PNF, while $\neg(a \wedge \neg b)$ is not. The well-known disjunctive and conjunctive normal forms are special cases of the PNF for propositional logic.

In order to transform any LTL formula into PNF, for each operator a dual operator needs to be incorporated into the syntax of PNF formulae. The propositional logical primitives of the positive normal form for LTL are the constant true and its dual constant false $= \neg$true, as well as conjunction $\wedge$ and its dual, $\vee$. De Morgan's rules $\neg(\varphi \vee \psi) \equiv \neg\varphi \wedge \neg\psi$ and $\neg(\varphi \wedge \psi) \equiv \neg\varphi \vee \neg\psi$ yield the duality of conjunction and disjunction. According to the duality rule $\neg \bigcirc \varphi \equiv \bigcirc \neg\varphi$, the next-step operator is a dual of itself. Therefore, no extra operator is necessary for $\bigcirc$. Now consider the until operator. First we observe that

$$\neg(\varphi \,\mathsf{U}\, \psi) \equiv ((\varphi \wedge \neg\psi) \,\mathsf{U}\, (\neg\varphi \wedge \neg\psi)) \vee \Box(\varphi \wedge \neg\psi).$$

The first disjunct on the right-hand side asserts that φ stops to hold "too early", i.e., before ψ becomes valid. The second disjunct states that φ always holds but never ψ. Clearly, in both cases, $\neg(\varphi \,\mathsf{U}\, \psi)$ holds.

This observation provides the motivation to introduce the operator W (called *weak until* or *unless*) as the dual of U. It is defined by:

$$\varphi \,\mathsf{W}\, \psi \stackrel{\text{def}}{=} (\varphi \,\mathsf{U}\, \psi) \vee \Box\varphi.$$

The semantics of $\varphi \,\mathsf{W}\, \psi$ is similar to that of $\varphi \,\mathsf{U}\, \psi$, except that $\varphi \,\mathsf{U}\, \psi$ requires a state to be reached for which ψ holds, whereas this is not required for $\varphi \,\mathsf{W}\, \psi$. Until U and weak until W are dual in the following sense:

$$
\begin{aligned}
\neg(\varphi \,\mathsf{U}\, \psi) &\equiv (\varphi \wedge \neg\psi) \,\mathsf{W}\, (\neg\varphi \wedge \neg\psi) \\
\neg(\varphi \,\mathsf{W}\, \psi) &\equiv (\varphi \wedge \neg\psi) \,\mathsf{U}\, (\neg\varphi \wedge \neg\psi)
\end{aligned}
$$

The reason that weak until is not a standard operator for LTL is that U and W have the same expressiveness, since

$$\begin{aligned} \Box\psi &\equiv \psi \,\mathsf{W}\, \text{false}, \\ \varphi \,\mathsf{U}\, \psi &\equiv (\varphi \,\mathsf{W}\, \psi) \wedge \underbrace{\Diamond\psi}_{\equiv\, \neg\Box\neg\psi}. \end{aligned}$$

That is to say, weak until is relevant only if restrictions are imposed on the occurrence of negation (as in PNF). It is interesting to observe that W and U satisfy the same expansion law:

$$\varphi \,\mathsf{W}\, \psi \;\equiv\; \psi \;\vee\; (\varphi \,\wedge\, \bigcirc(\varphi \,\mathsf{W}\, \psi)).$$

For $\psi = \text{false}$ this results in the expansion law for $\Box\varphi$:

$$\Box\varphi \;=\; \varphi \,\mathsf{W}\, \text{false} \;\equiv\; \text{false} \;\vee\; (\varphi \,\wedge\, \bigcirc(\varphi \,\mathsf{W}\, \text{false})) \;\equiv\; \varphi \,\wedge\, \bigcirc\Box\varphi.$$

The semantic difference between U and W is shown by the fact that $\varphi \,\mathsf{W}\, \psi$ is the *greatest* solution of

$$\kappa \;\equiv\; \psi \;\vee\; (\varphi \,\wedge\, \bigcirc\kappa).$$

This result is proven below. Recall $\varphi \,\mathsf{U}\, \psi$ is the least solution of this equivalence, see Lemma 5.18 on page 251.

Lemma 5.19. Weak-Until is the Greatest Solution of the Expansion Law

For LTL formulae φ and ψ, $\mathit{Words}(\varphi \,\mathsf{W}\, \psi)$ is the greatest LT property $P \subseteq (2^{AP})^\omega$ such that:

$$\mathit{Words}(\psi) \,\cup\, \{A_0A_1A_2\ldots \in \mathit{Words}(\varphi) \;\mid\; A_1A_2\ldots \in P\} \;\supseteq\; P \qquad (*)$$

Moreover, $\mathit{Words}(\varphi \,\mathsf{W}\, \psi)$ agrees with the LT property

$$\mathit{Words}(\psi) \,\cup\, \{A_0A_1A_2\ldots \in \mathit{Words}(\varphi) \;\mid\; A_1A_2\ldots \in \mathit{Words}(\varphi \,\mathsf{W}\, \psi)\}.$$

The formulation "greatest LT property with the indicated condition (*) is to be understood in the following sense:

(1) $P \supseteq \mathit{Words}(\varphi \,\mathsf{W}\, \psi)$ satisfies (*).

(2) $\mathit{Words}(\varphi \,\mathsf{W}\, \psi) \supseteq P$ for all LT properties P satisfying condition (*).

Proof: The fact that condition (1) is satisfied, even with equality rather than $\supseteq$, i.e., $\mathit{Words}(\varphi \,\mathsf{W}\, \psi)$ agrees with

$$\mathit{Words}(\psi) \cup \{A_0 A_1 A_2 \ldots \in \mathit{Words}(\varphi) \;\mid\; A_1 A_2 \ldots \in \mathit{Words}(\varphi \,\mathsf{W}\, \psi)\},$$

is an immediate conclusion from the expansion law $\varphi \,\mathsf{W}\, \psi \;\equiv\; \psi \vee (\varphi \wedge \bigcirc(\varphi \,\mathsf{W}\, \psi))$.

For proving condition (2), P is assumed to be an LT property satisfying (*). In particular, for all words $B_0\, B_1\, B_2\, B_3 \ldots \in (2^{AP})^\omega \setminus \mathit{Words}(\psi)$ we have:

(i) If $B_0\, B_1\, B_2\, B_3 \ldots \notin \mathit{Words}(\varphi)$, then $B_0\, B_1\, B_2\, B_3 \ldots \notin P$.

(ii) If $B_1\, B_2\, B_3 \ldots \notin P$, then $B_0\, B_1\, B_2\, B_3 \ldots \notin P$.

Now we demonstrate that

$$(2^{AP})^\omega \setminus \mathit{Words}(\varphi \,\mathsf{W}\, \psi) \;\subseteq\; (2^{AP})^\omega \setminus P.$$

Let $A_0\, A_1\, A_2 \ldots \in (2^{AP})^\omega \setminus \mathit{Words}(\varphi \,\mathsf{W}\, \psi)$. Then $A_0\, A_1\, A_2 \ldots \not\models \varphi \,\mathsf{W}\, \psi$ and thus

$$A_0\, A_1\, A_2 \ldots \models \neg(\varphi \,\mathsf{W}\, \psi) \;\equiv\; (\varphi \wedge \neg\psi)\,\mathsf{U}\,(\neg\varphi \wedge \neg\psi).$$

Thus there exists $k \geqslant 0$, such that:

(iii) $A_i\, A_{i+1}\, A_{i+2} \ldots \models \varphi \wedge \neg\psi$, for all $0 \leqslant i < k$,

(iv) $A_k\, A_{k+1}\, A_{k+2} \ldots \models \neg\varphi \wedge \neg\psi$.

In particular, none of the words $A_i\, A_{i+1}\, A_{i+2} \ldots$ belongs to $\mathit{Words}(\psi)$ for $0 \leqslant i \leqslant k$. We obtain:

$$\begin{array}{lll}
 & A_k\, A_{k+1}\, A_{k+2}\, A_{k+3} \ldots \notin P & \text{due to (i) and (iv)} \\
\Longrightarrow & A_{k-1}\, A_k\, A_{k+1}\, A_{k+2} \ldots \notin P & \text{due to (ii) and (iii)} \\
\Longrightarrow & A_{k-2}\, A_{k-1}\, A_k\, A_{k+1} \ldots \notin P & \text{due to (ii) and (iii)} \\
 & \vdots & \\
\Longrightarrow & A_0\, A_1\, A_2\, A_3 \ldots \notin P & \text{due to (ii) and (iii)}\ .
\end{array}$$

Thus, $(2^{AP})^\omega \setminus \mathit{Words}(\varphi \,\mathsf{W}\, \psi) \subseteq (2^{AP})^\omega \setminus P$, or equivalently, $\mathit{Words}(\varphi \,\mathsf{W}\, \psi) \supseteq P$. ∎

We are now ready to introduce the positive normal form for LTL which permits negation only on the level of literals and – to ensure the full expressiveness of LTL – uses the dual Boolean connectors $\wedge$ and $\vee$, the self-dual next-step operator $\bigcirc$, and U and W:

Definition 5.20. Positive Normal Form for LTL (Weak-Until PNF)

For $a \in AP$, the set of LTL formulae in *weak-until positive normal form* (weak-until PNF, for short, or simply PNF) is given by:

$$\varphi \ ::= \ \text{true} \ \Big| \ \text{false} \ \Big| \ a \ \Big| \ \neg a \ \Big| \ \varphi_1 \wedge \varphi_2 \ \Big| \ \varphi_1 \vee \varphi_2 \ \Big| \ \bigcirc \varphi \ \Big| \ \varphi_1 \,\mathsf{U}\, \varphi_2 \ \Big| \ \varphi_1 \,\mathsf{W}\, \varphi_2.$$

■

Due to the law $\Box\varphi \equiv \varphi \,\mathsf{W}\, \text{false}$, $\Box$ can also be considered as permitted operator of the W-positive normal form. As before, $\Diamond\varphi = \text{true}\,\mathsf{U}\,\varphi$. An example LTL formula in PNF is

$$\Diamond(a \,\mathsf{U}\, \Box b) \ \vee \ (a \wedge \neg c) \,\mathsf{W}\, (\Diamond(\neg a \,\mathsf{U}\, b)).$$

The LTL formulae $\neg(a \,\mathsf{U}\, b)$ and $c \vee \neg(a \wedge \Diamond b)$ are not in weak-until PNF.

The previous considerations were aimed to rewrite any LTL formula into weak-until PNF. This is done by successively "pushing" negations "inside" the formula at hand. This is facilitated by the following transformations:

$$\begin{array}{lcl}
\neg\text{true} & \rightsquigarrow & \text{false} \\
\neg\text{false} & \rightsquigarrow & \text{true} \\
\neg\neg\varphi & \rightsquigarrow & \varphi \\
\neg(\varphi \wedge \psi) & \rightsquigarrow & \neg\varphi \vee \neg\psi \\
\neg \bigcirc \varphi & \rightsquigarrow & \bigcirc \neg\varphi \\
\neg(\varphi \,\mathsf{U}\, \psi) & \rightsquigarrow & (\varphi \wedge \neg\psi) \,\mathsf{W}\, (\neg\varphi \wedge \neg\psi).
\end{array}$$

These rewrite rules are lifted to the derived operators as follows:

$$\neg(\varphi \ \vee \ \psi) \ \rightsquigarrow \ \neg\varphi \wedge \neg\psi \quad \text{and} \quad \neg\Diamond\varphi \ \rightsquigarrow \Box\neg\varphi \quad \text{and} \quad \neg\Box\varphi \ \rightsquigarrow \ \Diamond\neg\varphi.$$

Example 5.21. Positive Normal Form

Consider the LTL formula $\neg\Box((a \,\mathsf{U}\, b) \ \vee \ \bigcirc c)$. This formula is not in PNF, but can be transformed into an equivalent LTL formula in weak-until PNF as follows:

$$\begin{array}{ll}
 & \neg\Box((a \,\mathsf{U}\, b) \ \vee \ \bigcirc c) \\
\equiv & \Diamond\neg((a \,\mathsf{U}\, b) \ \vee \ \bigcirc c) \\
\equiv & \Diamond(\neg(a \,\mathsf{U}\, b) \wedge \neg \bigcirc c) \\
\equiv & \Diamond((a \wedge \neg b) \,\mathsf{W}\, (\neg a \wedge \neg b) \ \wedge \ \bigcirc \neg c)
\end{array}$$

■

Theorem 5.22. Existence of Equivalent Weak-Until PNF Formulae

For each LTL formula there exists an equivalent LTL formula in weak-until PNF.

The main drawback of the rewrite rules introduced above is that the length of the resulting LTL formula (in weak-until PNF) may be exponential in the length of the original nonPNF LTL formula. This is due to the rewrite rule for until where φ and ψ are duplicated. Although this can be slightly improved by adopting the law:

$$\neg(\varphi \,\mathsf{U}\, \psi) \;\equiv\; (\neg\psi)\,\mathsf{W}\,(\neg\varphi \wedge \neg\psi)$$

an exponential growth in length is not avoided here too due to the duplication of ψ in the right-hand side.

To avoid this exponential blowup in transforming an LTL formula in PNF, another temporal modality is used that is dual to the until operator: the so-called binary *release*-operator, denoted R. It is defined by

$$\varphi \,\mathsf{R}\, \psi \quad \stackrel{\text{def}}{=} \quad \neg(\neg\varphi \,\mathsf{U}\, \neg\psi).$$

Its intuitive interpretation is as follows. Formula $\varphi \,\mathsf{R}\, \psi$ holds for a word if ψ always holds, a requirement that is released as soon as φ becomes valid. Thus, the formula $\text{false}\,\mathsf{R}\,\varphi$ is valid if φ always holds, since the release condition (false) is a contradiction. Formally, for a given word $\sigma = A_0\, A_1 \ldots \in (2^{AP})^\omega$:

$$\begin{array}{ll}
& \sigma \models \varphi \,\mathsf{R}\, \psi \\
\text{iff} & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(* definition of } \mathsf{R} \text{ *)} \\
& \neg\exists j \geqslant 0.\ (\sigma[j..] \models \neg\psi \wedge \forall i < j.\ \sigma[i..] \models \neg\varphi) \\
\text{iff} & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(* semantics of negation *)} \\
& \neg\exists j \geqslant 0.\ (\sigma[j..] \not\models \psi \ \wedge\ \forall i < j.\ \sigma[i..] \not\models \varphi) \\
\text{iff} & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(* duality of } \exists \text{ and } \forall \text{ *)} \\
& \forall j \geqslant 0.\ \neg\Big(\sigma[j..] \not\models \psi \ \wedge\ \forall i < j.\ \sigma[i..] \not\models \varphi\Big) \\
\text{iff} & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(* de Morgan's law *)} \\
& \forall j \geqslant 0.\ \Big(\neg(\sigma[j..] \not\models \psi) \ \vee\ \neg\forall i < j.\ \sigma[i..] \not\models \varphi\Big) \\
\text{iff} & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(* semantics of negation *)} \\
& \forall j \geqslant 0.\ \Big(\sigma[j..] \models \psi \ \vee\ \exists i < j.\ \sigma[i..] \models \varphi\Big) \\
\text{iff} & \\
& \forall j \geqslant 0.\ \sigma[j..] \models \psi \quad \text{or} \quad \exists i \geqslant 0.\ (\sigma[i..] \models \varphi) \wedge \forall k \leqslant i.\ \sigma[k..] \models \psi).
\end{array}$$

The always operator is obtained from the release operator by:

$$\Box\varphi \;\equiv\; \text{false}\ \mathsf{R}\ \varphi.$$

The weak-until and the until operator are obtained by:

$$\varphi \,\mathsf{W}\, \psi \;\equiv\; (\neg\varphi \;\vee\; \psi) \,\mathsf{R}\, (\varphi \;\vee\; \psi), \qquad \varphi \,\mathsf{U}\, \psi \;\equiv\; \neg(\neg\varphi \,\mathsf{R}\, \neg\psi).$$

Vice versa, $\varphi \,\mathsf{R}\, \psi \;\equiv\; (\neg\varphi \wedge \psi) \,\mathsf{W}\, (\varphi \wedge \psi)$. The expansion law (see Exercise 5.8) for release reads as follows:

$$\varphi \,\mathsf{R}\, \psi \;\equiv\; \psi \wedge (\varphi \;\vee\; \bigcirc\, (\varphi \,\mathsf{R}\, \psi)).$$

We now revisit the notion of PNF, which is defined using the R-operator rather than W:

Definition 5.23. Positive Normal Form (release PNF)

For $a \in AP$, LTL formulae in release positive normal form (release PNF, or simply PNF) are given by

$$\varphi \;::=\; \text{true} \;\Big|\; \text{false} \;\Big|\; a \;\Big|\; \neg a \;\Big|\; \varphi_1 \wedge \varphi_2 \;\Big|\; \varphi_1 \vee \varphi_2 \;\Big|\; \bigcirc\, \varphi \;\Big|\; \varphi_1 \,\mathsf{U}\, \varphi_2 \;\Big|\; \varphi_1 \,\mathsf{R}\, \varphi_2.$$

■

The following transformation rules push negations inside and serve to transform a given LTL formula into an equivalent LTL formula in PNF:

$$\begin{array}{lcl}
\neg\text{true} & \rightsquigarrow & \text{false} \\
\neg\neg\varphi & \rightsquigarrow & \varphi \\
\neg(\varphi \wedge \psi) & \rightsquigarrow & \neg\varphi \vee \neg\psi \\
\neg \bigcirc\, \varphi & \rightsquigarrow & \bigcirc\, \neg\varphi \\
\neg(\varphi \,\mathsf{U}\, \psi) & \rightsquigarrow & \neg\varphi \,\mathsf{R}\, \neg\psi
\end{array}$$

In each rewrite rule the size of the resulting formula increases at most by an additive constant.

Theorem 5.24. Existence of Equivalent Release PNF Formulae

For any LTL formula φ there exists an equivalent LTL formula φ' in release PNF with $|\varphi'| = \mathcal{O}(|\varphi|)$.

5.1.6 Fairness in LTL

In Chapter 3, we have seen that typically some fairness constraints are needed to verify liveness properties. Three types of fairness constraints (for sets of actions) have been distinguished, namely unconditional, strong, and weak fairness. Accordingly, the satisfaction

relation for LT properties (denoted $\models$) has been adapted to the fairness assumption $\mathcal{F}$ (denoted $\models_{\mathcal{F}}$), where $\mathcal{F}$ is a triple of (sets of) fairness constraints. This entails that only fair paths are considered while determining the satisfaction of a property. In this section, this approach is adopted in the context of LTL. That is to say, rather than determining for transition system TS and LTL formula φ whether $TS \models \varphi$, we focus on the fair executions of TS. The main difference with the action-based fairness assumptions (and constraints) is that we now focus on *state-based* fairness.

Definition 5.25. LTL Fairness Constraints and Assumptions

Let Φ and Ψ be propositional logic formulae over AP.

1. An *unconditional LTL fairness constraint* is an LTL formula of the form

$$ufair \;=\; \Box\Diamond\Psi.$$

2. A *strong LTL fairness condition* is an LTL formula of the form

$$sfair \;=\; \Box\Diamond\Phi \;\longrightarrow\; \Box\Diamond\Psi.$$

3. A *weak LTL fairness constraint* is an LTL formula of the form

$$wfair \;=\; \Diamond\Box\Phi \;\longrightarrow\; \Box\Diamond\Psi.$$

An *LTL fairness assumption* is a conjunction of LTL fairness constraints (of any arbitrary type). ■

For instance, a strong LTL fairness assumption denotes a conjunction of strong LTL fairness constraints, i.e., a formula of the form

$$sfair \;=\; \bigwedge_{0<i\leqslant k} (\Box\Diamond\Phi_i \;\longrightarrow\; \Box\Diamond\Psi_i)$$

for propositional logical formulae Φ_i and Ψ_i over AP. Weak and unconditional LTL fairness assumptions are defined in a similar way.

In their most general form, LTL fairness assumptions are (as in Definition 3.46, page 133) a conjunction of unconditional, strong, and weak fairness assumptions:

$$\mathit{fair} \;=\; \mathit{ufair} \;\wedge\; \mathit{sfair} \;\wedge\; \mathit{wfair}.$$

where *ufair*, *sfair*, and *wfair* are unconditional, strong, and weak LTL fairness assumptions, respectively. As in the case of action-based fairness assumptions, the rule of thumb for imposing fairness assumptions is: strong (or unconditional) fairness assumptions are useful for solving contentions, and weak fairness is often sufficient for resolving the nondeterminism that results from interleaving.

In the sequel, we adopt the same notations as for action-based fairness assumptions. Let $\mathit{FairPaths}(s)$ denote the set of all fair paths starting in s and $\mathit{FairTraces}(s)$ the set of all traces induced by fair paths starting in s. Formally, for fixed formula *fair*,

$$\begin{aligned} \mathit{FairPaths}(s) &= \{\, \pi \in \mathit{Paths}(s) \mid \pi \models \mathit{fair} \,\}, \\ \mathit{FairTraces}(s) &= \{\, \mathit{trace}(\pi) \mid \pi \in \mathit{FairPaths}(s) \,\}. \end{aligned}$$

These notions can be lifted to transition systems in the obvious way yielding $\mathit{FairPaths}(TS)$ and $\mathit{FairTraces}(TS)$. To identify the fairness assumption *fair*, we may write $\mathit{FairPaths}_{\mathit{fair}}(\cdot)$ or $\mathit{FairTraces}_{\mathit{fair}}(\cdot)$.

Definition 5.26. Satisfaction Relation for LTL with Fairness

For state s in transition system TS (over AP) without terminal states, LTL formula φ, and LTL fairness assumption *fair* let

$$\begin{aligned} s \models_{\mathit{fair}} \varphi \quad &\text{iff} \quad \forall \pi \in \mathit{FairPaths}(s).\, \pi \models \varphi \quad \text{and} \\ TS \models_{\mathit{fair}} \varphi \quad &\text{iff} \quad \forall s_0 \in I.\, s_0 \models_{\mathit{fair}} \varphi. \end{aligned}$$

■

TS satisfies φ under the LTL fairness assumption *fair* if φ holds for all *fair* paths that originate from some initial state.

Example 5.27. Mutual Exclusion with Randomized Arbiter (Fairness)

Consider the following approach to two-process mutual exclusion. A randomized arbiter, see the program graphs in Figure 5.9, decides which process is acquiring access to the critical section. It does so by tossing coins. We abstract from the probabilities, and model the coin tossing by a nondeterministic choice between the alternatives "heads" and "tails". It is assumed that the two contending processes communicate with the arbiter via the actions enter_1 and enter_2. The critical section is released by synchronizing over

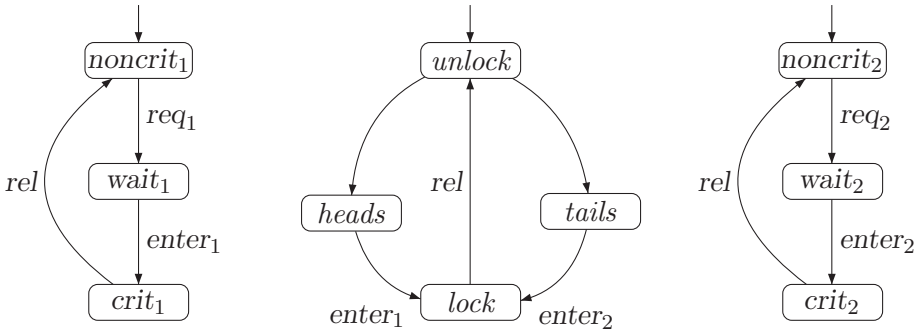


Figure 5.9: Mutual exclusion with a randomized arbiter.

the action *release*. For the sake of simplicity, we refrain from indicating which process is releasing the critical section.

The property "process P_1 is in its critical section infinitely often" *cannot* be established, since, for instance, the underlying transition system representation does not exclude an execution in which only the second process may perform an action while P_1 is entirely ignored. Thus:

$$TS_1 \parallel Arbiter \parallel TS_2 \not\models \Box\Diamond\, crit_1.$$

If a coin is assumed to be fair enough such that both events "heads" and "tails" occur with positive probability, the unrealistic case of one of the two alternatives never happening can be ignored by means of the unconditional LTL fairness assumption:

$$fair \;=\; \Box\Diamond heads \,\wedge \Box\Diamond tails.$$

It is not difficult to check that now:

$$TS_1 \parallel Arbiter \parallel TS_2 \;\models_{fair}\; \Box\Diamond crit_1 \;\wedge\; \Box\Diamond crit_2.$$

■

Example 5.28. Communication Protocol (Fairness)

Consider the alternating bit protocol as described in Example 2.32 (page 57). For the sake of convenience, the program graph of the sender of the alternating bit protocol is repeated in Figure 5.10. The liveness property $\Box\Diamond start$ states that the protocol returns infinitely often to its initial state. In this initial state the action *snd_msg(0)* is enabled. It follows that

$$ABP \;\not\models\; \Box\Diamond start$$

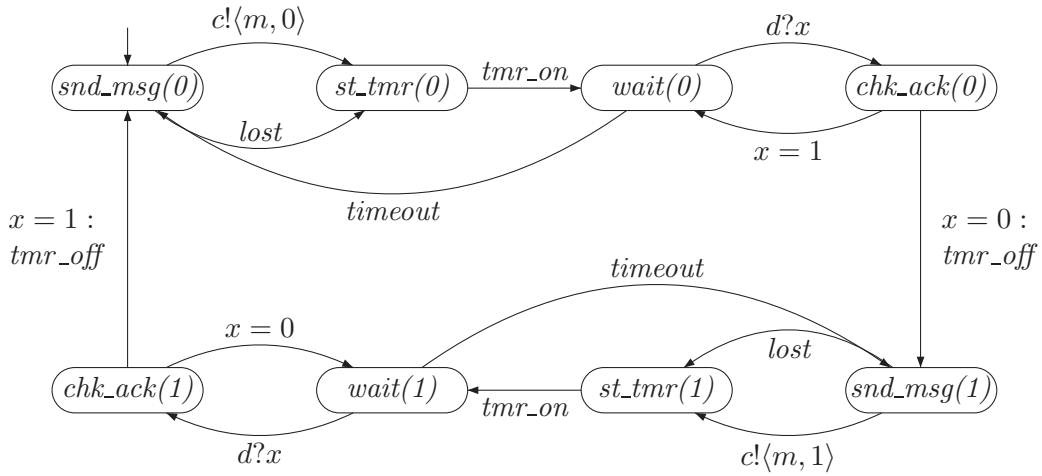


Figure 5.10: Program graph of ABP sender S.

since the unrealistic scenario in which (after some finite time) each message with alternating bit 1 is lost cannot be excluded. This corresponds to the path

$$\ldots\ldots\, s_i \xrightarrow{lost} s_{i+1} \xrightarrow{tmr\_on} s_{i+2} \xrightarrow{timeout} s_{i+3} \xrightarrow{lost} \ldots\ldots$$

Suppose we impose the strong LTL fairness assumption

$$sfair \;=\; \bigwedge_{b=0,1} \; \bigwedge_{k<cap(c)} (\Box\Diamond\,(send(b) \wedge |c| = k) \;\rightarrow\; \Box\Diamond\,|c| = k+1).$$

Here, $|c| = n$ stands for the atomic proposition that holds in the states $\langle \ell, \eta, \xi \rangle$ in which channel c contains exactly n elements, i.e., the length of the word $\xi(c)$ equals n. Thus, *sfair* describes (from the state-based point of view) that the loss of a transmitted message is not continuously possible. We now obtain

$$ABP \;\models_{sfair}\; \Box\Diamond start.$$

Note that it is essential to impose a strong fairness assumption on *send*(b); as this action is not continuously enabled, a weak fairness assumption is insufficient. ■

In Section 3.5 (page 126 ff.), fairness was introduced using sets of actions; e.g., an execution is unconditionally A-fair for a set of actions A, whenever each action $\alpha \in A$ occurs infinitely often. LTL-fairness, however, is defined on atomic propositions, i.e., from a state-based perspective. Is there any relationship between these two—at first sight, rather different—approaches toward fairness?

The advantage of the action-based formulation of fairness assumptions is that many useful (and realizable) fairness assumptions can easily be expressed. Using the state-based

perspective, this may be less intuitive. For instance, the enabling of a process (or, more generally, of a certain action) is not necessarily a property that can be determined from the (atomic propositions in a) state. This discrepancy is, however, not always present.

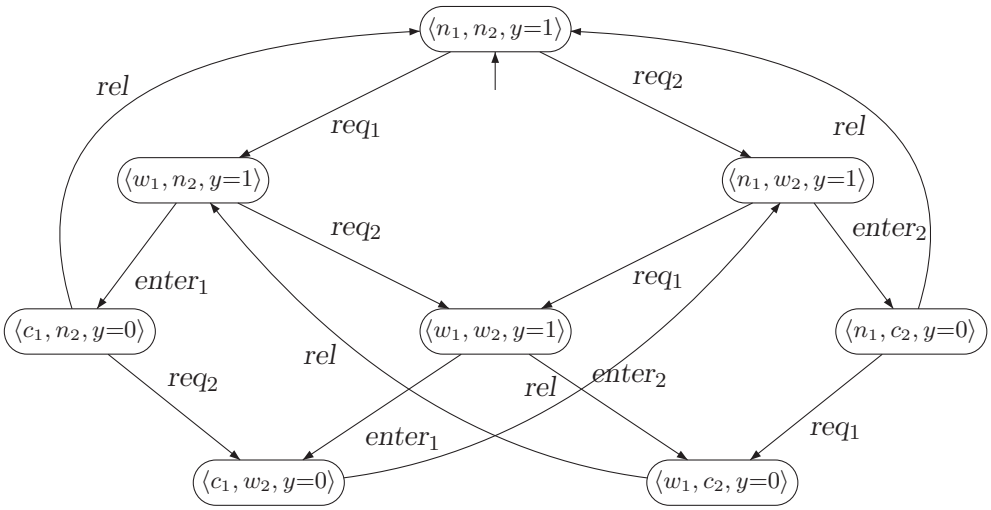


Figure 5.11: Semaphore-based mutual exclusion algorithm.

Example 5.29. State-Based vs. Action-Based Fairness

To exemplify this, consider the semaphore-based two-process mutual exclusion protocol (see Figure 5.11) together with the action-based strong-fairness assumption

$$\mathcal{F}_{strong} \;=\; \{\,\{\,enter_1\,\},\{\,enter_2\,\}\,\}.$$

Let us try to state the same constraint by means of a (state-based) LTL fairness assumption. Observe first that the action $enter_1$ is executable if and only if process P_1 is in the local state $wait_1$ and process P_2 is not in its critical section. Besides, on executing action $enter_1$, process P_1 moves to its critical section. Thus, strong fairness for $\{\,enter_1\,\}$ can be described by the LTL fairness assumption:

$$sfair_1 \;=\; \Box\Diamond(wait_1 \wedge \neg crit_2) \rightarrow \Box\Diamond crit_1.$$

The assumption $sfair_2$ is defined analogously. It now follows that $sfair \;=\; sfair_1 \wedge sfair_2$ describes $\mathcal{F}_{strong}$.

$\mathcal{F}_{strong}$ requires each process to enter its critical section infinitely often when it infinitely often gets the opportunity to do so. This does not forbid a process to never leave its noncritical section. To avoid this unrealistic scenario, the weak fairness constraint

$$\mathcal{F}_{weak} \;=\; \{\{\,req_1\,\},\{\,req_2\,\}\}$$

requires that any process infinitely often requests to enter the critical section when it continuously is able to do so. This (action-based) weak fairness constraint can be formulated as (state-based) LTL fairness assumption in a similar way as above. Observe that the request action of P_i is executable if and only if process P_i is in the local state $noncrit_i$. Weak fairness for $\{ req_i \}$ thus correponds to the LTL fairness assumption:

$$wfair_i \;=\; \Diamond\Box noncrit_i \rightarrow \Box\Diamond wait_i.$$

Let $fair \;=\; sfair \wedge wfair$ where $wfair \;=\; wfair_1 \wedge wfair_2$. It then follows that

$$TS_{Sem} \;\models_{fair}\; \Box\Diamond crit_1 \;\wedge\; \Box\Diamond crit_2.$$

■

In many cases, it is possible to replace action-based fairness by state-based LTL fairness assumptions. This, however, requires the possibility to deduce from the state label the possible enabled actions and the last executed action. It turns out that action-based fairness assumptions can always be "translated" into analogous LTL fairness assumptions. This works as follows. The intuition is to make a copy of each noninitial state s such that it is recorded which action was executed to enter state s. Such copy is made for every possible action via which the state can be entered. The copied state $\langle s, \alpha \rangle$ indicates that state s has been reached by performing α as last action.

Formally, this works as follows. For transition system $TS = (S, Act, \rightarrow, I, AP, L)$ let

$$TS' \;=\; (S', Act', \rightarrow', I', AP', L')$$

where $Act' = Act \uplus \{ begin \}$, $I' = I \times \{ begin \}$ and $S' \;=\; I' \;\cup\; (S \times Act)$. The transition relation in TS' is defined by the rules:

$$\frac{s \xrightarrow{\alpha} s'}{\langle s, \beta \rangle \xrightarrow{\alpha}{}' \langle s', \alpha \rangle} \qquad \text{and} \qquad \frac{s_0 \xrightarrow{\alpha} s \quad s_0 \in I}{\langle s_0, begin \rangle \xrightarrow{\alpha}{}' \langle s, \alpha \rangle}$$

The state labeling is defined as follows. Let

$$AP' \;=\; AP \cup \{ enabled(\alpha), taken(\alpha) \mid \alpha \in Act \}$$

with the labeling function

$$L'(\langle s, \alpha \rangle) \;=\; L(s) \cup \{ taken(\alpha) \} \cup \{ enabled(\beta) \mid \beta \in Act(s) \}$$

for $\langle s, \alpha \rangle \in S \times Act$ and

$$L'(\langle s_0, begin \rangle) \;=\; L(s_0) \cup \{enabled(\beta) \mid \beta \in Act(s_0) \}.$$

It can easily be established that

$$Traces_{AP}(TS) \;=\; Traces_{AP}(TS').$$

Strong fairness for a set of actions $A \subseteq Act$ can now be described by the strong LTL fairness assumption:

$$sfair_A \;=\; \Box\Diamond enabled(A) \;\rightarrow\; \Box\Diamond taken(A)$$

where

$$enabled(A) = \bigvee_{\alpha\in A} enabled(\alpha) \quad \text{and} \quad taken(A) = \bigvee_{\alpha\in A} taken(\alpha).$$

Unconditional and weak action-based fairness assumptions for TS can be transformed into LTL fairness assumptions for TS' in a similar way. For action-based fairness assumption $\mathcal{F}$ for TS and *fair* the corresponding LTL fairness assumption for TS', it follows that the set of fair traces coincides:

$$\begin{aligned} &\{trace_{AP}(\pi) \mid \pi \in Paths(TS), \pi \text{ is } \mathcal{F}\text{-fair}\} \\ =\; &\{trace_{AP}(\pi') \mid \pi' \in Paths(TS'), \pi' \models fair\}. \end{aligned}$$

Stated differently, $FairTraces_{\mathcal{F}}(TS) = FairTraces_{fair}(TS')$ where in TS' only atomic propositions in AP are considered. In particular, for every LT property P over AP,

$$TS \models_{\mathcal{F}} P \;\text{ iff }\; TS' \models_{fair} P.$$

Conversely, a (state-based) LTL fairness assumption cannot always be represented as action-based fairness assumption. This fact follows from the fact that strong or weak LTL fairness assumptions need not be realizable, while any action-based strong or weak fairness assumptions can be realized by a scheduler. In this sense, *state-based LTL fairness assumptions are more general than action-based fairness assumptions.*

The following theorem shows that the satisfaction relation $\models_{fair}$ as defined in Definition 5.26 has a strong relation to the usual satisfaction relation $\models$.

Theorem 5.30. Reduction of $\models_{fair}$ to $\models$

For transition system TS without terminal states, LTL formula φ, and LTL fairness assumption fair:

$$TS \models_{fair} \varphi \qquad \textit{if and only if} \qquad TS \models (fair \rightarrow \varphi).$$

Proof: $\Rightarrow$: Assume $TS \models_{fair} \varphi$. Then, for any path $\pi \in Paths(TS)$ either $\pi \models fair \wedge \varphi$ or $\pi \models \neg fair$. Thus, $\pi \models (fair \rightarrow \varphi)$, and consequently $TS \models (fair \rightarrow \varphi)$.

$\Leftarrow$: by a similar reasoning. ■

Example 5.31. On-The-Fly Garbage Collection

An important characteristic of pointer-manipulating programs is that certain parts of the dynamically changing data structure, such as a list or a tree, may become inaccessible. That is to say, this part of the data structure is not "reachable" by dereferencing one of the program variables. In order to be able to recycle these inaccessible memory cells, so-called *garbage collecting* algorithms are employed. These algorithms are key parts of operating systems, and play a major role in compilers. In this example, we focus on *on-the-fly* garbage collection algorithms. These algorithms attempt to identify and recycle inaccessible memory cells *concurrently* with the running programs that may manipulate the dynamically changing data structure. Typical requirements for such garbage collection algorithms are

Safety: An accessible (i.e., reachable) memory cell is never collected.

Liveness: Any unreachable storage cell is eventually collected.

The memory cells. The memory is considered to consist of a fixed number N of memory cells. The number of memory cells is static, i.e., the dynamic allocation and deallocation of cells (as in the `C`-statement `malloc`) is not considered. The memory cells are organized in a directed graph, whose structure may change during the computation; this happens, e.g., when carrying out an assignment $v := w$ where v and w point to a memory cell. For the sake of simplicity, it is assumed that each cell (= vertex) has at most one pointer (= edge) to another cell. Let $son(i)$ denote the immediate successor of cell i. Cells that do not have an outgoing reference are equipped with a self-reference. Thus, each cell has exactly one outgoing edge in the graph representation. Vertices are numbered. There is a fixed set of *root* vertices. Root cells are never considered as garbage. A memory cell is reachable if it is reachable in the graph from a root vertex. Cells that are not reachable from a root vertex are *garbage* cells. The memory is partitioned into three fragments: the accessible cells (i.e., cells that are reachable by the running processes), the free cells (i.e., cells that are unused and that can be assigned to running processes), and the unreachable cells. As for the garbage collection algorithm, it is only of importance which cells are unreachable. The distinction between free and accessible cells is of no importance, and will be neglected in the remainder.

Modeling the garbage collector. The entire system is modeled as the parallel composition of the garbage collector (process *Collector*), and a process that models the behaviour of the running processes manipulating the shared data structure. Thus:

$$\mathit{Mutator} \parallel \mathit{Collector}$$

For simplicity, we abstain from a detailed transition system representation and describe the mutator and the collector by pseudocode algorithms. As the garbage collecting algorithm should work correctly for any arbitrary set of concurrently running processes, the mutator is modeled by means of a single process that can nondeterministically choose two arbitrary reachable cells i and k and change the current reference $i \to j$ into $i \to k$. This corresponds to the assignment "$son(i) := k$". Note that if i was the only cell pointing to j, then after the assignment $son(i) := k$, the memory cell j has become garbage.

Algorithm 9 Mutator and garbage collector (naive version)

```
                                                        (* mutator *)
while true do
  Nondeterministically choose two reachable nodes i and k;
  son(i) := k
od
```

```
                                                        (* garbage collector *)
while true do
  label all reachable cells;                  (* e.g., by depth-first search *)
  for all cells i do
    if i is not labeled then collect i
  od
  add the collected cells to the list of free memory cells
od
```

Now let us consider the garbage collector. A naive technique could be based on a DFS or BFS which labels cells and, subsequently, collects all unlabeled cells as garbage (see Algorithm 9). The collected cells are added to the list of free memory cells, and thus are turned into reachable cells. This simple idea works correctly when the actions of the mutator are not interlocked with those of the garbage collector. As an on-the-fly algorithm, however, this naive idea fails due to the possible interleavings between the mutator and the collector. This is illustrated in Figure 5.12 where six memory configurations are depicted. Circles denote memory cells and edges represent the pointer structure. Nonshaded cells are not visited, gray ones are visited, and cells with a thick border have been completely processed. Due to the changes made by the mutator during the garbage collection phase, one of the memory cells remains unlabeled although it is accessible. The collector would now collect the unlabeled but accessible cell.

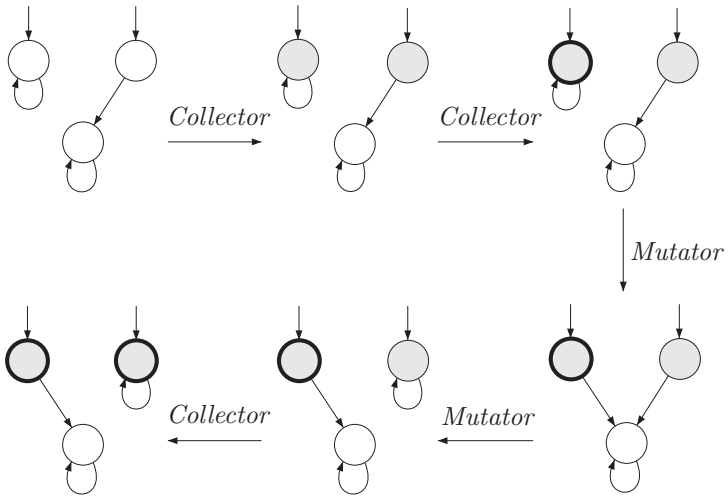


Figure 5.12: Failure of the naive garbage collector.

An alternative is to use a labeling algorithm that successively passes through all cells and labels their sons if the appropriate cell is labeled itself. Initially, only root cells are labeled. This technique is iterated as long as the number of labeled cells does not change anymore. The auxiliary variables M and M_{old} are used to this purpose. M counts the number of labeled cells in the current stage of labeling. M_{old} stands for the number of labeled cells in the previous stage of labeling. Additionally, the mutator supports the labeling process by labeling cell k as soon as a reference is redirected to k (see Algorithm 10). It is not difficult to show that the participation of the mutator is necessary to obtain a correct garbage collection algorithm. If it does not participate, there can always be found some interference pattern between the mutator and the collector such that reachable cells are made inaccessible by the mutator, causing a flaw in the garbage collection. Figure 5.13 demonstrates the functioning of the collector for four linearly linked reachable cells, where it is assumed that the labeling steps of the collector are not interrupted by the mutator.

The shading represents the labeling by the collector. The thick border of a node indicates that this node was considered in the for loop. Thus, in the last step, all four nodes were processed in the for loop. Therefore, the result of the first iteration is $M_{old} = 1$ (number of root cells) and $M = 3$, since exactly three cells were labeled. The labels arising at the end of the subsequent iterations are indicated in Figure 5.14.

Figure 5.15 shows an example that indicates that the mutator has to label cells, as otherwise it cannot be ensured that the collector identifies all reachable cells. The leftmost figure indicates the starting memory configuration. The situation after the collector has labeled the root cell and has processed the top left node in the FOR loop is depicted in

Algorithm 10 Ben Ari's on-the-fly garbage collection algorithm

```
                                                          (* mutator *)
while true do
  let  i and k be reachable memory cells;
  label k;
  son(i) := k
od
                                                          (* garbage collector *)
while true do
  label all root cells;
  M := number of root cells;
  repeat
    M_old := M;
    for all node i do
      if i is labeled then
        if son(i) is unlabeled then label son(i); M := M+1;  fi
      fi
    od
  until M = M_old;

  for all cell i do
    if i is labeled  then  delete label for cell i
    else collect cell i                                   (* cell i is garbage *)
    fi
  od
  add the collected cells to the list of free cells
od
```

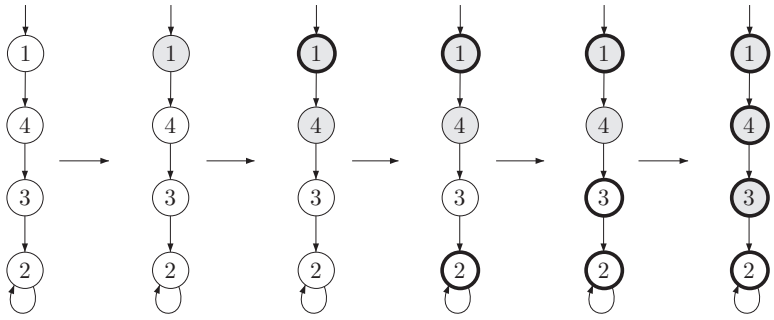


Figure 5.13: Example of Ben Ari's on-the-fly garbage collector (one iteration).

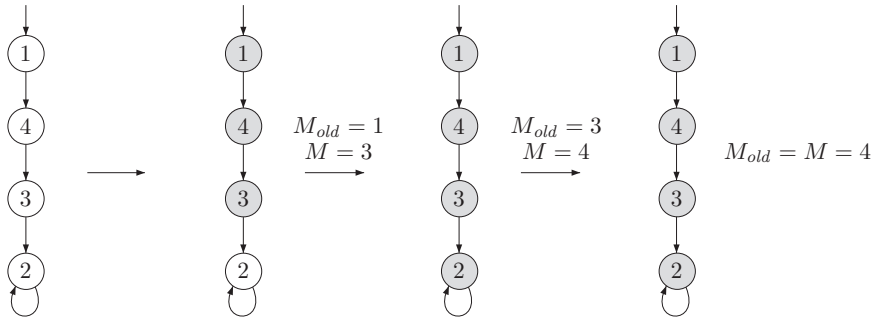


Figure 5.14: Example of Ben-Ari's on-the-fly garbage collector (all iterations).

the next figure. The third and fourth figures show a possible modification of the pointer structure by the mutator. The collector then would investigate the upper cell on the right and label it. Since this cell does not have an unlabeled successor, the first round of the collector is finished. As the obtained memory configuration is symmetric to the initial one, the whole procedure might be repeated, which leads to the initial configuration of Figure 5.15. In the second round, the collector also just labels the two upper cells. Since the number of labeled cells has not increased and assuming that the mutator does not label the lower cell, this cell would be wrongly regarded as garbage (and thus be collected).

The examples illustrate the functioning of the algorithm but are by no means meant as proof of correctness. Due to the enormous size of the state space and the extremely high degree of nondeterminism, the analysis of the garbage collection algorithm is extremely difficult. Note that the states of the transition system, which results from the parallel composition of mutator and collector, consist of control components plus a representation of the current labels and of another component that gives information about the current

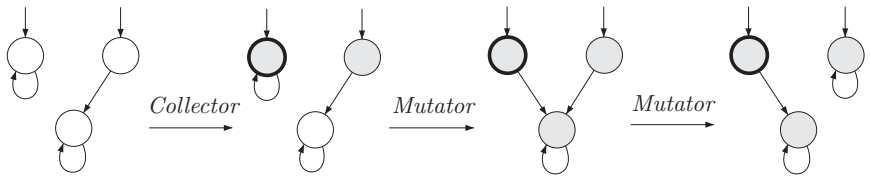


Figure 5.15: Labeling memory cells by the mutator is necessary.

memory configuration. For N nodes, there exist 2^N possible labels and N^N possible graphs. The high degree of nondeterminism results from the mutator that can change an arbitrary pointer. It is not useful to restrict the mutator (and thus reducing the degree of nondeterminism), since this would impose restrictions on the applicability of the garbage collecting algorithm.

To conclude, we will formalize the requirements on the concurrent garbage collection algorithm. To that end, let atomic proposition $\mathit{collect}(i)$ hold for cell i in a state if and only if cell i has been collected in that state. Similarly, $\mathit{accessible}(i)$ holds in those states in which cell i is reachable from a root cell. Given these atomic propositions, the safety property "an accessible memory cell is never collected" can be specified as

$$\bigwedge_{0<i\leqslant N} \Box(\ \mathit{collect}(i) \rightarrow \neg \mathit{accessible}(i)\ ).$$

The liveness property "any unreachable storage cell is eventually collected" is described as LTL formula

$$\bigwedge_{0<i\leqslant N} \Box(\ \neg \mathit{accessible}(i)\ \rightarrow\ \Diamond \mathit{collect}(i)\ ).$$

It turns out that Ben Ari's concurrent garbage collection algorithm satisfies the safety property, but refutes the liveness property. The latter happens, e.g., in the pathological case where only the mutator acts infinitely often. To rule out this unrealistic behavior, weak process fairness can be imposed. ■

5.2 Automata-Based LTL Model Checking

In this section we address the model-checking problem for LTL. The starting point is a finite transition system TS and an LTL formula φ that formalizes a requirement on TS. The problem is to check whether $TS \models \varphi$. If φ is refuted, an error trace needs to be provided for debugging purposes. The considerations in Section 2.1 show that transition

systems are typically huge. Therefore, a manual proof of $TS \models \varphi$ is extremely difficult. Instead, verification tools are desirable, which enable a fully automated analysis of the transition system.

In general, not just a single requirement but several requirements are relevant. These requirements can be represented by separate formulae, such as $\varphi_1, \ldots, \varphi_k$, and be combined into $\varphi_1 \wedge \ldots \wedge \varphi_k$ to obtain a specification of all requirements. Alternatively, the requirements φ_i can be treated separately. This is often more efficient than considering them together. Moreover, the decomposition of the entire requirement specification into several requirements is advised if errors are to be expected or if the validity of φ_i is known by a prior analysis.

An LTL model-checking algorithm is a decision procedure which for a transition system TS and LTL formula φ returns the answers "yes" if $TS \models \varphi$, and "no" (plus a counterexample) if $TS \not\models \varphi$. The counterexample consists of an appropriate finite prefix of an infinite path in TS where φ does not hold. Theorem 5.30 shows that special measures to handle fairness assumptions are unnecessary as fairness assumptions can be encoded in the LTL formula to be checked. (For reasons of efficiency, however, it is advised to use special algorithms to treat fairness assumptions.)

Throughout this section, TS is assumed to be finite and to have no terminal states. The model-checking algorithm presented in the following is based on the *automata-based approach* as originally suggested by Vardi and Wolper (1986). This approach is based on the fact that each LTL formula φ can be represented by a nondeterministic Büchi automaton (NBA). The basic idea is to try to disprove $TS \models \varphi$ by "looking" for a path π in TS with $\pi \models \neg\varphi$. If such a path is found, a prefix of π is returned as error trace. If no such path is encountered, it is concluded that $TS \models \varphi$.

The essential steps of the model-checking algorithm as summarized in Algorithm 11 and Figure 5.16, rely on the following observations:

$$\begin{aligned} TS \models \varphi \quad & \text{iff} \quad \mathit{Traces}(TS) \subseteq \mathit{Words}(\varphi) \\ & \text{iff} \quad \mathit{Traces}(TS) \cap ((2^{AP})^\omega \setminus \mathit{Words}(\varphi)) = \varnothing \\ & \text{iff} \quad \mathit{Traces}(TS) \cap \mathit{Words}(\neg\varphi) = \varnothing. \end{aligned}$$

Hence, for NBA $\mathcal{A}$ with $\mathcal{L}_\omega(\mathcal{A}) = \mathit{Words}(\neg\varphi)$ we have

$$TS \models \varphi \quad \text{if and only if} \quad \mathit{Traces}(TS) \cap \mathcal{L}_\omega(\mathcal{A}) = \varnothing.$$

Thus, to check whether φ holds for TS one first constructs an NBA for the negation of the input formula φ (representing the "bad behaviors") and then applies the techniques explained in Chapter 4 for the intersection problem.

Algorithm 11 Automaton-based LTL model checking

Input: finite transition system TS and LTL formula φ (both over AP)
Output: "yes" if $TS \models \varphi$; otherwise, "no" plus a counterexample

Construct an NBA $\mathcal{A}_{\neg\varphi}$ such that $\mathcal{L}_\omega(\mathcal{A}_{\neg\varphi}) = \mathit{Words}(\neg\varphi)$
Construct the product transition system $TS \otimes \mathcal{A}$

if there exists a path π in $TS \otimes \mathcal{A}$ satisfying the accepting condition of $\mathcal{A}$ **then**
  return "no" and an expressive prefix of π
else
  return "yes"
fi

It remains to explain how a given LTL formula can be represented by an NBA and how such an NBA can be constructed algorithmically. First observe that for the LTL formula φ, the LTL semantics provided in Definition 5.6 on page 235 yields a language $\mathit{Words}(\varphi) \subseteq (2^{AP})^\omega$. Thus, the alphabet of NBA for LTL formulae is $\Sigma = 2^{AP}$. The next step is to show that $\mathit{Words}(\varphi)$ is ω-regular, and hence, representable by a nondeterministic Büchi automaton.

Example 5.32. NBA for LTL Formulae

Before treating the details of transforming an LTL formula into an NBA, we provide some examples. As in Chapter 4, propositional logic formulae are used (instead of the set notations) for a symbolic representation of the edges of an NBA. These formulae are built by the symbols $a \in AP$, the constant true, and the Boolean connectors, and thus they can be interpreted over *sets* of atomic propositions, i.e., the elements $A \in \Sigma = 2^{AP}$. For instance, if $AP = \{a, b\}$ then $q \xrightarrow{a \vee b} q'$ is a short notation for the three transitions:

$$q \xrightarrow{\{a\}} q',\ q \xrightarrow{\{b\}} q', \text{ and } q \xrightarrow{\{a,b\}} q'.$$

The language of all words $\sigma = A_0 A_1 \ldots \in 2^{AP}$ satisfying the LTL formula $\Box\Diamond \mathit{green}$ ("infinitely often *green*") is accepted by the NBA $\mathcal{A}$ shown in Figure 5.17. Here, AP is a set of atomic propositions containing *green*. Note that $\mathcal{A}$ is in the accept state q_1 if and only if the last consumed symbol (the last set A_i of the input word $A_0 A_1 A_2 \ldots \in (2^{AP})^\omega$) contains the propositional symbol *green*. Therefore, the accepted language $\mathcal{L}_\omega(\mathcal{A})$ is exactly the set of all infinite words $A_0 A_1 A_2 \ldots$ with infinitely many indices i where $\mathit{green} \in A_i$. Thus,

$$\mathcal{L}_\omega(\mathcal{A}) \ = \ \mathit{Words}(\Box\Diamond \mathit{green}).$$

The accepting run that generates the word $\sigma = \{\mathit{green}\}\varnothing\{\mathit{green}\}\varnothing \ldots$ is $(q_0\, q_1)^\omega$.

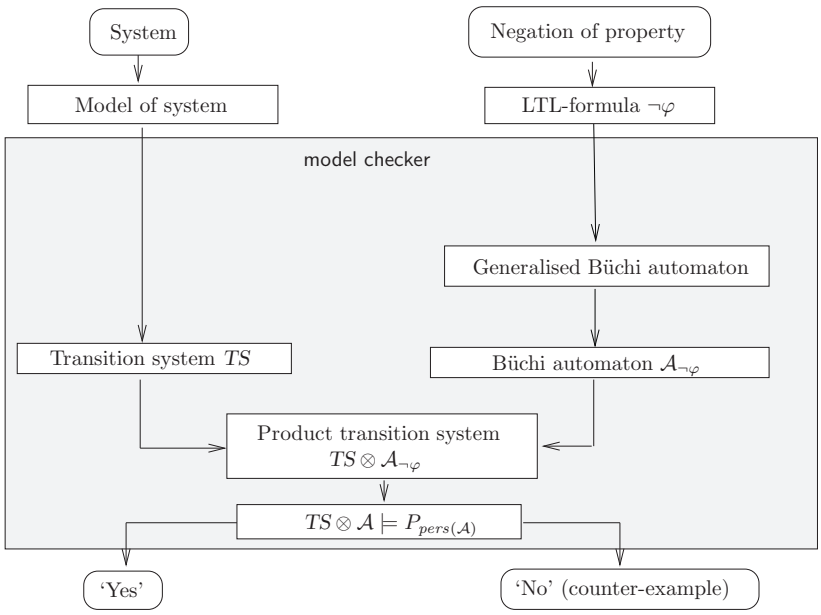


Figure 5.16: Overview of LTL model checking.

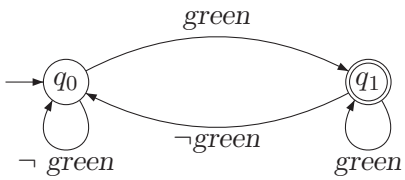


Figure 5.17: NBA for $\Box\Diamond$*green*.

As a second example consider the liveness property: "whenever event a occurs, event b will eventually occur". For example, the property given by the LTL formula $\Box(\mathit{request} \rightarrow \Diamond \mathit{response})$ is of this form. An associated NBA over the alphabet $2^{\{a,b\}}$ where $a = \mathit{request}$ and $b = \mathit{response}$ is shown in Figure 5.18.

The automata in Figures 5.17 and 5.18 are *deterministic*, i.e., they have exactly one run for each input word. To represent temporal properties like "eventually forever (from some moment on)", the concept of nondeterminism is, however, necessary.

The NBA $\mathcal{A}$ shown in Figure 5.19 accepts the language $\mathit{Words}(\Diamond\Box a)$. Here, $AP \supseteq \{ a \}$ and $\Sigma = 2^{AP}$; see also Example 4.51 (page 191). Intuitively, the NBA $\mathcal{A}$ nondeterministically decides (by means of an omniscient oracle) when a continuously holds. Note that state q_2 may be omitted, as there is no accepting run beginning in q_2. (The reader should bear in mind that DBA and NBA are not equally expressive; see Section 4.3.3 on page 188.) ■

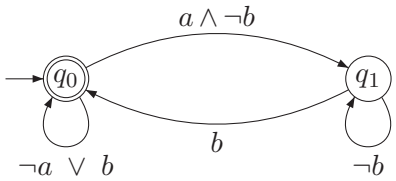


Figure 5.18: NBA for $\Box(a \rightarrow \Diamond b)$.

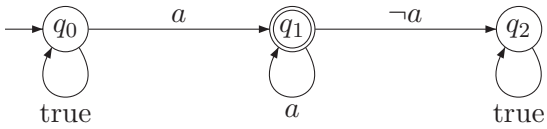


Figure 5.19: NBA for $\Diamond\Box a$.

A key ingredient to the model-checking algorithm for LTL is the construction of an NBA $\mathcal{A}$ satisfying

$$\mathcal{L}_\omega(\mathcal{A}) \;=\; \mathit{Words}(\varphi)$$

for the LTL formula φ. In order to do so, first a generalized NBA is constructed for φ, which subsequently is transformed into an equivalent NBA. For the latter step we employ the recipe as provided in Theorem 4.56 on page 195. For the sake of convenience we recall the definition of generalized NBA; see also Definition 4.52 on page 193.

Definition 5.33. Generalized NBA (GNBA)

A generalized NBA is a tuple $\mathcal{G} = (Q, \Sigma, \delta, Q_0, \mathcal{F})$ where Q, Σ, δ, Q_0 are defined as for NBA (i.e., Q is a finite state space, Σ an alphabet, $Q_0 \subseteq Q$ the set of initial states, and $\delta : Q \times \Sigma \rightarrow 2^Q$ the transition relation) and $\mathcal{F}$ is a (possibly empty) subset of 2^Q. The elements of $\mathcal{F}$ are called *acceptance sets*. The accepted language $\mathcal{L}_\omega(\mathcal{G})$ consists of all infinite words in $(2^{AP})^\omega$ that have at least one infinite run $q_0\, q_1\, q_2 \ldots$ in $\mathcal{G}$ such that for each acceptance set $F \in \mathcal{F}$ there are infinitely many indices i with $q_i \in F$. ■

A GNBA for which $\mathcal{F}$ is a singleton set can be regarded as an NBA. If the set $\mathcal{F}$ of acceptance sets in $\mathcal{G}$ is empty, the language $\mathcal{L}_\omega(\mathcal{G})$ consists of all infinite words that have an infinite run in $\mathcal{G}$. Hence, if $\mathcal{F} = \varnothing$, then $\mathcal{G}$ can be viewed as an NBA for which all states are accepting.

Let us consider how to construct a GNBA over the alphabet 2^{AP} for a given LTL formula φ (over AP), i.e., a GNBA $\mathcal{G}_\varphi$ with $\mathcal{L}_\omega(\mathcal{G}_\varphi) = \mathit{Words}(\varphi)$. Assume φ only contains the operators $\wedge$, $\neg$, $\bigcirc$ and U, i.e., the derived operators $\vee$, $\rightarrow$, $\Diamond$, $\Box$, W, and so on are assumed to be expressed in terms of the basic operators. Since the special case $\varphi = \text{true}$ is trivial, it may be assumed that $\varphi \neq \text{true}$.

The basic idea for the construction of $\mathcal{G}_\varphi$ is as follows. Let $\sigma = A_0\,A_1\,A_2\ldots \in \mathit{Words}(\varphi)$. The sets $A_i \subseteq AP$ are expanded by subformulae ψ of φ such that an infinite word $\bar{\sigma} = B_0\,B_1\,B_2\ldots$ with the following property arises:

$$\psi \in B_i \qquad \text{if and only if} \qquad \underbrace{A_i\,A_{i+1}\,A_{i+2}\ldots}_{\sigma^i} \models \psi.$$

For technical reasons, the subformulae ψ of φ are considered as well as their negation $\neg\psi$. Consider, e.g.,

$$\varphi = a\ \mathsf{U}\,(\neg a \wedge b) \quad \text{and} \quad \sigma = \{\,a\,\}\,\{\,a,b\,\}\,\{\,b\,\}\ldots.$$

In this case, B_i is a subset of the set of formulae

$$\underbrace{\{\,a, b, \neg a, \neg a \wedge b, \varphi\,\}}_{\text{subformulae of } \varphi} \cup \underbrace{\{\,\neg b, \neg(\neg a \wedge b), \neg\varphi\,\}}_{\text{their negation}}.$$

The set $A_0 = \{\,a\,\}$ is extended with the formulae $\neg b$, $\neg(\neg a \wedge b)$, and φ, since all these formulae hold in $\sigma^0 = \sigma$, and all other subformulae in the above set are refuted by σ. We thus obtain

$$B_0 \;=\; \{\,a, \neg b, \neg(\neg a \wedge b), \varphi\,\}.$$

The set $A_1 = \{\,a, b\,\}$ is extended with $\neg(\neg a \wedge b)$ and φ, as these are the only subformulae in the above set that hold in $\sigma^1 = \{\,a, b\,\}\,\{\,b\,\}\ldots$. The $A_2 = \{\,b\,\}$ is extended with $\neg a$, $\neg a \wedge b$ and φ as they hold in $\sigma^2 = \{\,b\,\}\ldots$. This yields a word of the form:

$$\bar{\sigma} \;=\; \{\,a, \neg b, \neg(\neg a \wedge b), \varphi\,\}\,\{\,a, b, \neg(\neg a \wedge b), \varphi\,\}\,\{\,\neg a, b, \neg a \wedge b, \varphi\,\}\ldots$$

As σ is infinite, this procedure is of course not effective. The example is just meant to explain the intuition behind the construction of the states in the GNBA.

The GNBA $\mathcal{G}_\varphi$ is constructed such that the sets B_i constitute its *states*. Moreover, the construction ensures that $\bar{\sigma} = B_0\,B_1\,B_2\ldots$ is a *run* for $\sigma = A_0\,A_1\,A_2\ldots$ in $\mathcal{G}_\varphi$. The accepting conditions for $\mathcal{G}_\varphi$ are chosen such that the run $\bar{\sigma}$ is accepting if and only if $\sigma \models \varphi$. Thus, we have to encode the meaning of the logical operators into the states, transitions, and acceptance sets of $\mathcal{G}_\varphi$. The meaning of propositional logic operators $\wedge$, $\neg$, and the constant true will be encoded in the states by requiring consistent formula sets B_i. The semantics of the next-step operator relies on a nonlocal condition and will be encoded in the transition relation. The meaning of the until operator is split according to the expansion law into local conditions (encoded in the states) and a next-step condition (encoded in the transitions). Since the expansion law does not provide a full characterization of until, a further condition is imposed which expresses the fact that the meaning of until yields the least solution of the expansion law (see Lemma 5.18 on page 251). This will be encoded by the acceptance sets of $\mathcal{G}_\varphi$.

As explained above, the formula sets are subsets of subformulae of φ and their negation.

Definition 5.34. Closure of φ

The *closure* of LTL formula φ is the set $\mathit{closure}(\varphi)$ consisting of all subformulae ψ of φ and their negation $\neg\psi$ (where ψ and $\neg\neg\psi$ are identified). ■

For instance, for $\varphi = a \,\mathsf{U}\, (\neg a \wedge b)$, the set $\mathit{closure}(\varphi)$ consists of the formulae

$$a, b, \neg a, \neg b, \neg a \wedge b, \neg(\neg a \wedge b), \varphi, \neg\varphi.$$

It is not difficult to assess that $|\mathit{closure}(\varphi)| \in \mathcal{O}(|\varphi|)$. A set of formulae $B \subseteq \mathit{closure}(\varphi)$ is called *elementary* if B is the set of all formulae $\psi \in \mathit{closure}(\varphi)$ with $\pi \models \psi$ for a path π. For this, B should not contain propositional logic contradictions and it must be *locally consistent* with respect to the until operator. Since for any path π and formula ψ, either $\pi \models \psi$ or $\pi \models \neg\psi$, it is additionally required that elementary sets of formulae are *maximal*. The precise definition of these three conditions is provided in Figure 5.20 on page 277.

Definition 5.35. Elementary Sets of Formulae

$B \subseteq \mathit{closure}(\varphi)$ is *elementary* if it is consistent with respect to propositional logic, maximal, and locally consistent with respect to the until operator. ■

The requirements for local consistency result from the expansion law

$$\varphi_1 \,\mathsf{U}\, \varphi_2 \;\equiv\; \varphi_2 \;\vee\; (\varphi_1 \;\wedge\; \bigcirc\, (\varphi_1 \,\mathsf{U}\, \varphi_2)).$$

Due to the required maximality and propositional logic consistency, we have

$$\psi \in B \text{if and only if} \quad \neg\psi \notin B$$

for all elementary sets B and subformulae ψ of φ. Further, due to maximality and local consistency, we have

$$\varphi_1, \varphi_2 \notin B \quad \text{implies} \quad \varphi_1 \,\mathsf{U}\, \varphi_2 \notin B.$$

Hence, if $\varphi_1, \varphi_2 \notin B$ then $\{\, \neg\varphi_1, \neg\varphi_2, \neg(\varphi_1 \,\mathsf{U}\, \varphi_2) \,\} \subseteq B$; here, it is assumed that $\varphi_1 \,\mathsf{U}\, \varphi_2$ is a subformula of φ.

Example 5.36. Elementary Sets of Formulae

Let $\varphi = a \,\mathsf{U}\, (\neg a \wedge b)$. The set $B = \{\, a, b, \varphi \,\} \subseteq \mathit{closure}(\varphi)$ is consistent with respect to propositional logic and locally consistent with respect to the until operator. It is, however, not maximal, since for $\neg a \wedge b \in \mathit{closure}(\varphi)$:

$$\neg a \wedge b \notin B \qquad \text{and} \qquad \neg(\neg a \wedge b) \notin B.$$

1. B is *consistent* with respect to propositional logic, i.e., for all $\varphi_1 \wedge \varphi_2$, $\psi \in \mathit{closure}(\varphi)$:
   - $\varphi_1 \wedge \varphi_2 \in B \;\Leftrightarrow\; \varphi_1 \in B$ and $\varphi_2 \in B$
   - $\psi \in B \;\Rightarrow\; \neg\psi \notin B$
   - $\mathrm{true} \in \mathit{closure}(\varphi) \;\Rightarrow\; \mathrm{true} \in B$.
2. B is *locally consistent* with respect to the until operator, i.e., for all $\varphi_1 \mathsf{U} \varphi_2 \in \mathit{closure}(\varphi)$:
   - $\varphi_2 \in B \;\Rightarrow\; \varphi_1 \mathsf{U} \varphi_2 \in B$
   - $\varphi_1 \mathsf{U} \varphi_2 \in B$ and $\varphi_2 \notin B \;\Rightarrow\; \varphi_1 \in B$.
3. B is *maximal*, i.e., for all $\psi \in \mathit{closure}(\varphi)$:
   - $\psi \notin B \;\Rightarrow\; \neg\psi \in B$.

Figure 5.20: Properties of elementary sets of formulae.

The set of formulae $\{ a, b, \neg a \wedge b, \varphi \}$ contains the propositional logic "contradiction" a and $\neg a \wedge b$ and therefore is not elementary. The set

$$\{\neg a, \neg b, \neg(\neg a \wedge b), \varphi\}$$

is consistent with respect to propositional logic but contains a local inconsistency with respect to the until operator U, since $a \mathsf{U} (\neg a \wedge b) \in B$ and $\neg a \wedge b \notin B$, but $a \notin B$. This means that

$$\pi \models \neg a, \quad \pi \models \neg(\neg a \wedge b), \quad \text{and} \quad \pi \models \varphi$$

are impossible for any path π.

The following sets are elementary:

$$\begin{array}{lcl}
B_1 & = & \{ \quad a, \quad b, \quad \neg(\neg a \wedge b), \quad \varphi \; \}, \\
B_2 & = & \{ \quad a, \quad b, \quad \neg(\neg a \wedge b), \quad \neg\varphi \; \}, \\
B_3 & = & \{ \quad a, \quad \neg b, \quad \neg(\neg a \wedge b), \quad \varphi \; \}, \\
B_4 & = & \{ \quad a, \quad \neg b, \quad \neg(\neg a \wedge b), \quad \neg\varphi \; \}, \\
B_5 & = & \{ \quad \neg a, \quad \neg b, \quad \neg(\neg a \wedge b), \quad \neg\varphi \; \}, \\
B_6 & = & \{ \quad \neg a, \quad b, \quad \neg a \wedge b, \quad \varphi \; \}.
\end{array}$$

■

The proof of the following theorem shows how to construct for an arbitrary LTL formula φ a GNBA $\mathcal{G}_\varphi$ with $\mathcal{L}_\omega(\mathcal{G}_\varphi) = \mathit{Words}(\varphi)$. This construction is one of the initial steps of the

LTL model checking algorithm; see Figure 5.16 (page 273). Subsequently, the resulting GNBA $\mathcal{G}_\varphi$ is transformed into an NBA $\mathcal{A}_\varphi$ by means of the technique indicated in the proof of Theorem 4.56 (page 195).

Theorem 5.37. GNBA for LTL Formula

For any LTL formula φ (over AP) there exists a GNBA $\mathcal{G}_\varphi$ over the alphabet 2^{AP} such that

(a) $\mathit{Words}(\varphi) = \mathcal{L}_\omega(\mathcal{G}_\varphi)$.

(b) $\mathcal{G}_\varphi$ *can be constructed in time and space* $2^{\mathcal{O}(|\varphi|)}$.

(c) The number of accepting sets of $\mathcal{G}_\varphi$ *is bounded above by* $\mathcal{O}(|\varphi|)$.

Proof: Let φ be an LTL formula over AP. Let $\mathcal{G}_\varphi = (Q, 2^{AP}, \delta, Q_0, \mathcal{F})$ where

- Q is the set of all elementary sets of formulae $B \subseteq \mathit{closure}(\varphi)$,
- $Q_0 \;=\; \{\, B \in Q \mid \varphi \in B \,\}$,
- $\mathcal{F} \;=\; \{\, F_{\varphi_1 \,\mathsf{U}\, \varphi_2} \mid \varphi_1 \,\mathsf{U}\, \varphi_2 \in \mathit{closure}(\varphi) \}$ where

$$F_{\varphi_1 \,\mathsf{U}\, \varphi_2} \;=\; \{\, B \in Q \mid \varphi_1 \,\mathsf{U}\, \varphi_2 \notin B \text{ or } \varphi_2 \in B \,\}.$$

The transition relation $\delta : Q \times 2^{AP} \to 2^Q$ is given by:

- If $A \neq B \cap AP$, then $\delta(B, A) \;=\; \varnothing$.
- If $A = B \cap AP$, then $\delta(B, A)$ is the set of all elementary sets of formulae B' satisfying
  - (i) for every $\bigcirc \psi \in \mathit{closure}(\varphi)$: $\bigcirc \psi \in B \;\;\Leftrightarrow\;\; \psi \in B'$, and
  - (ii) for every $\varphi_1 \,\mathsf{U}\, \varphi_2 \in \mathit{closure}(\varphi)$:

$$\varphi_1 \,\mathsf{U}\, \varphi_2 \in B \;\;\Leftrightarrow\;\; (\varphi_2 \in B \;\vee\; (\varphi_1 \in B \wedge \varphi_1 \,\mathsf{U}\, \varphi_2 \in B')).$$

The constraints (i) and (ii) reflect the semantics of the next-step and the until operator, respectively. Rule (ii) is justified by the expansion law:

$$\varphi_1 \,\mathsf{U}\, \varphi_2 \;\;\equiv\;\; \varphi_2 \vee (\varphi_1 \wedge \bigcirc\, (\varphi_1 \,\mathsf{U}\, \varphi_2)).$$

To model the semantics of U, an acceptance set F_ψ is introduced for every subformula $\psi = \varphi_1 \,\mathsf{U}\, \varphi_2$ of φ. The underlying idea is to ensure that in every run $B_0\, B_1\, B_2 \ldots$ for which $\psi \in B_0$, we have $\varphi_2 \in B_j$ (for some $j \geqslant 0$) and $\varphi_1 \in B_i$ for all $i < j$. The requirement that a word σ satisfies $\varphi_1 \,\mathsf{U}\, \varphi_2$ only if φ_2 will actually eventually become true is ensured by the accepting set $F_{\varphi_1 \,\mathsf{U}\, \varphi_2}$.

Let us first consider the claim (b). States in the GNBA $\mathcal{G}_\varphi$ are elementary sets of formulae in $\mathit{closure}(\varphi)$. Let $\mathrm{subf}(\varphi)$ denote the set of all subformulae of φ. The number of states in $\mathcal{G}_\varphi$ is bounded by $2^{|\mathrm{subf}(\varphi)|}$, the number of possible formula sets. (The elementary formula sets B can be represented by bit vectors containing a single bit per subformula ψ of φ which indicates whether ψ or $\neg\psi$ belongs to B.) As $|\mathrm{subf}(\varphi)| \leqslant 2{\cdot}|\varphi|$, the number of states in the GNBA $\mathcal{G}_\varphi$ is bounded by $2^{\mathcal{O}(|\varphi|)}$. Claim (c) follows directly from the fact that the number of accept sets is equal to the number of until-subformulae in φ.

It remains to show that (a) $\mathcal{L}_\omega(\mathcal{G}_\varphi) = \mathit{Words}(\varphi)$. We prove set inclusion in both directions.

$\supseteq$: Let $\sigma = A_0\, A_1\, A_2 \ldots \in \mathit{Words}(\varphi)$. Then, $\sigma \in (2^{AP})^\omega$ and $\sigma \models \varphi$. The elementary set B_i of formulae is defined as follows:

$$B_i \;=\; \{\, \psi \in \mathit{closure}(\varphi) \mid A_i A_{i+1} \ldots \models \psi \,\} \tag{5.1}$$

Obviously, B_i is an elementary set of formulae, i.e., $B_i \in Q$. We now prove that $B_0\, B_1\, B_2 \ldots$ is an accepting run for σ. Observe that $B_{i+1} \in \delta(B_i, A_i)$ for all $i \geqslant 0$, since for all i:

- $A_i = B_i \cap AP$
- for $\bigcirc \psi \in \mathit{closure}(\varphi)$:

$$\begin{array}{ll} & \bigcirc \psi \in B_i \\ \text{iff} & \qquad\qquad \text{(* equation (5.1) for } B_i \text{ *)} \\ & A_i\, A_{i+1} \ldots \models \bigcirc \psi \\ \text{iff} & \qquad\qquad \text{(* semantics of } \bigcirc \text{ *)} \\ & A_{i+1}\, A_{i+2} \ldots \models \psi \\ \text{iff} & \qquad\qquad \text{(* equation (5.1) for } B_{i+1} \text{ *)} \\ & \psi \in B_{i+1} \end{array}$$

- for $\varphi_1 \mathsf{U} \varphi_2 \in \mathit{closure}(\varphi)$:

$$
\begin{array}{ll}
 & \varphi_1 \mathsf{U} \varphi_2 \in B_i \\
\text{iff} & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(* equation (5.1) for } B_i \text{ *)} \\
 & A_i\, A_{i+1} \ldots \models \varphi_1 \mathsf{U} \varphi_2 \\
\text{iff} & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{(* semantics of until *)} \\
 & A_i\, A_{i+1} \ldots \models \varphi_2 \quad \text{or} \\
 & A_i\, A_{i+1} \ldots \models \varphi_1 \text{ and } (A_{i+1}\, A_{i+2} \ldots \models \varphi_1 \mathsf{U} \varphi_2 \qquad\qquad\qquad\qquad\quad) \\
 & \\
\text{iff} & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(* equation (5.1) for } B_i \text{ and } B_{i+1} \text{ *)} \\
 & \\
 & \varphi_2 \in B_i \text{ or } (\varphi_1 \in B_i \text{ and } \varphi_1 \mathsf{U} \varphi_2 \in B_{i+1}) \qquad\qquad\qquad\qquad\qquad\qquad\quad .
\end{array}
$$

This shows that $B_0\, B_1\, B_2 \ldots$ is a run of $\mathcal{G}_\varphi$. It remains to prove that this run is accepting, i.e., for each subformula $\varphi_{1,j} \mathsf{U} \varphi_{2,j}$ in $\mathit{closure}(\varphi)$, $B_i \in F_j$ for infinitely many i. By contraposition. Assume there are finitely many i such that $B_i \in F_j$. We have:

$$B_i \notin F_j = F_{\varphi_{1,j} \mathsf{U} \varphi_{2,j}} \;\Rightarrow\; \varphi_{1,j} \mathsf{U} \varphi_{2,j} \in B_i \text{ and } \varphi_{2,j} \notin B_i.$$

As $B_i = \{\, \psi \in \mathit{closure}(\varphi) \mid A_i\, A_{i+1} \ldots \models \psi \,\}$, it follows that if $B_i \notin F_j$, then:

$$A_i\, A_{i+1} \ldots \models \varphi_{1,j} \mathsf{U} \varphi_{2,j} \quad \text{and} \quad A_i\, A_{i+1} \ldots \not\models \varphi_{2,j}.$$

Thus, $A_k\, A_{k+1} \ldots \models \varphi_{2,j}$ for some $k > i$. By definition of the formula sets B_i, it then follows that $\varphi_{2,j} \in B_k$, and by definition of F_j, $B_k \in F_j$. Thus, $B_i \in F_j$ for finitely many i, then $B_k \in F_j$ for infinitely many k. Contradiction.

Thus, $B_0\, B_1\, B_2 \ldots$ is an accepting run of $\mathcal{G}_\varphi$, and hence, $A_0\, A_1\, A_2 \ldots \in \mathcal{L}_\omega(\mathcal{G}_\varphi)$.

$\subseteq$: Let $\sigma = A_0\, A_1\, A_2 \ldots \in \mathcal{L}_\omega(\mathcal{G}_\varphi)$, i.e., there is an accepting run $B_0\, B_1\, B_2 \ldots$, say, for σ in $\mathcal{G}_\varphi$. Since

$$\delta(B, A) = \varnothing \quad \text{for all pairs } (B, A) \text{ with } A \neq B \cap AP,$$

it follows that $A_i = B_i \cap AP$ for $i \geqslant 0$. Thus

$$\sigma \;=\; (B_0 \cap AP)\; (B_1 \cap AP)\; (B_2 \cap AP) \ldots$$

Our proof obligation now becomes $(B_0 \cap AP)\; (B_1 \cap AP)\; (B_2 \cap AP) \ldots \;\models\; \varphi$. We prove the following more general proposition:

For $B_0 B_1 B_2 \ldots$ a sequence with $B_i \in Q$ satisfying

(i) for all $i \geqslant 0 : B_{i+1} \in \delta(B_i, A_i)$, and

(ii) for all $F \in \mathcal{F} : \overset{\infty}{\exists}\, j \geqslant 0.\ B_j \in F$,

we have for all $\psi \in \mathit{closure}(\varphi)$:

$$\psi \in B_0 \quad \Leftrightarrow \quad A_0\, A_1\, A_2 \ldots \models \psi.$$

The proof of this claim is by structural induction on the structure of ψ.

Base case: The statement for $\psi = \text{true}$ or $\psi = a$ with $a \in AP$ follows directly from equation (5.1) and the definition of closure.

Induction step: Based on the induction hypothesis that the claim holds for $\psi', \varphi_1, \varphi_2 \in \mathit{closure}(\varphi)$, it is proven that for the formulae

$$\psi = \bigcirc \psi',\ \psi = \neg\psi',\ \psi = \varphi_1 \wedge \varphi_2 \text{ and } \psi = \varphi_1 \,\mathsf{U}\, \varphi_2$$

the claim also holds. We provide the detailed proof for $\psi = \varphi_1 \,\mathsf{U}\, \varphi_2$. Let $A_0\, A_1\, A_2 \ldots \in (2^{AP})^\omega$ and $B_0\, B_1\, B_2 \ldots \in Q^\omega$ satisfying the constraints (i) and (ii). It is now shown that:

$$\psi \in B_0 \quad \text{iff} \quad A_0\, A_1\, A_2 \ldots \models \psi.$$

This goes as follows.

$\Leftarrow$: Assume $A_0\, A_1\, A_2 \ldots \models \psi$ where $\psi = \varphi_1 \,\mathsf{U}\, \varphi_2$. Then, there exists $j \geqslant 0$ such that

$$A_j\, A_{j+1} \ldots \models \varphi_2 \quad \text{and} \quad A_i\, A_{i+1} \ldots \models \varphi_1 \text{ for } 0 \leqslant i < j.$$

From the induction hypothesis (applied to φ_1 and φ_2) it follows that

$$\varphi_2 \in B_j \quad \text{and} \quad \varphi_1 \in B_i \quad \text{for } 0 \leqslant i < j.$$

By induction on j we obtain: $\varphi_1 \,\mathsf{U}\, \varphi_2 \in B_j, B_{j-1} \ldots, B_0$.

$\Rightarrow$: Assume $\varphi_1 \,\mathsf{U}\, \varphi_2 \in B_0$. Since B_0 is elementary, $\varphi_1 \in B_0$ or $\varphi_2 \in B_0$. Distinguish between $\varphi_2 \in B_0$ and $\varphi_2 \notin B_0$. If $\varphi_2 \in B_0$, it follows from the induction hypothesis $A_0\, A_1 \ldots \models \varphi_2$, and thus $A_0\, A_1 \ldots \models \varphi_1 \,\mathsf{U}\, \varphi_2$. This remains the case $\varphi_2 \notin B_0$. Then $\varphi_1 \in B_0$ and $\varphi_1 \,\mathsf{U}\, \varphi_2 \in B_0$. Assume $\varphi_2 \notin B_j$ for all $j \geqslant 0$. From the definition of the transition relation δ, we obtain using an inductive argument (successively applied to $\varphi_1 \in B_j$, $\varphi_2 \notin B_j$ and $\varphi_1 \,\mathsf{U}\, \varphi_2 \in B_j$ for $j \geqslant 0$):

$$\varphi_1 \in B_j \quad \text{and} \quad \varphi_1 \,\mathsf{U}\, \varphi_2 \in B_j \quad \text{for all } j \geqslant 0.$$

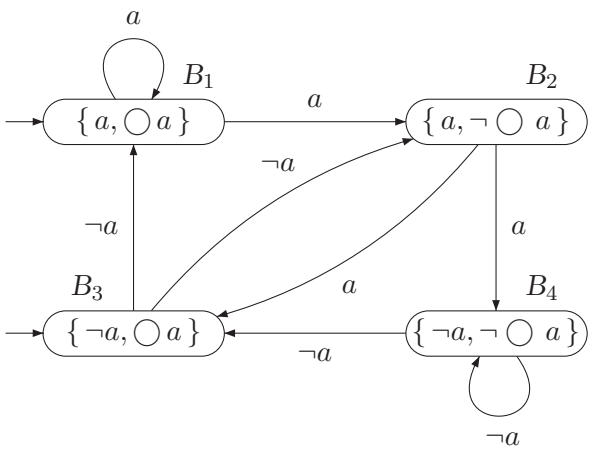


Figure 5.21: A generalised Büchi automaton for the LTL formula $\bigcirc a$.

As $B_0\, B_1\, B_2 \ldots$ satisfies constraint (ii), it follows that

$$B_j \in F_{\varphi_1 \,\mathsf{U}\, \varphi_2} \text{for infinitely many } j \geqslant 0.$$

On the other hand, we have

$$\underbrace{\varphi_2 \notin B_j \quad \text{and} \quad \varphi_1 \,\mathsf{U}\, \varphi_2 \in B_j}_{\text{iff} \quad B_j \notin F_{\varphi_1 \,\mathsf{U}\, \varphi_2}}$$

for all j. Contradiction! Thus, $\varphi_2 \in B_j$ for some $j \geqslant 0$. Without loss of generality, assume $\varphi_2 \notin B_0, \ldots, B_{j-1}$, i.e., let j be the smallest index such that $\varphi_2 \in B_j$. The induction hypothesis for $0 \leqslant i < j$ yields

$$\varphi_1 \in B_i \quad \text{and} \quad \varphi_1 \,\mathsf{U}\, \varphi_2 \in B_i \quad \text{for all } 0 \leqslant i < j.$$

From the induction hypothesis applied to φ_1 and φ_2 it follows that

$$A_j\, A_{j+1} \ldots \models \varphi_2 \quad \text{and} \quad A_i\, A_{i+1} \ldots \models \varphi_1 \text{ for } 0 \leqslant i < j.$$

We conclude that $A_0\, A_1\, A_2 \ldots \models \varphi_1 \,\mathsf{U}\, \varphi_2$. ■

Example 5.38. Construction of a GNBA (Next Step)

Consider $\varphi = \bigcirc a$. The GNBA $\mathcal{G}_\varphi$ (see Figure 5.21) is obtained as indicated in the proof of Theorem 5.37, . The states of the automaton are the elementary sets of formulae contained in

$$\mathit{closure}(\varphi) \;=\; \{\, a, \bigcirc a, \neg a, \neg \bigcirc a \,\}.$$

The state space Q consists of the following elementary sets

$$\begin{array}{lcllcl} B_1 & = & \{\, a, \bigcirc a \,\}, & B_2 & = & \{\, a, \neg\bigcirc a \,\}, \\ B_3 & = & \{\, \neg a, \bigcirc a \,\}, & B_4 & = & \{\, \neg a, \neg\bigcirc a \,\}. \end{array}$$

The initial states of $\mathcal{G}_\varphi$ are the elementary sets $B \in Q$ with $\varphi = \bigcirc a \in B$. Thus $Q_0 = \{\, B_1, B_3 \,\}$. Let us justify some of the transitions. For state B_1, $B_1 \cap \{\, a \,\} = \{\, a \,\}$, so $\delta(B_1, \varnothing) = \varnothing$. In addition, $\delta(B_1, \{\, a \,\}) = \{\, B_1, B_2 \,\}$ since $\bigcirc a \in B_1$ and B_1 and B_2 are the only states that contain a. As $B_2 \cap \{\, a \,\} = \{\, a \,\}$, we get $\delta(B_2, \varnothing) = \varnothing$. Moreover, $\delta(B_2, \{\, a \,\}) = \{\, B_3, B_4 \,\}$. This follows from the fact that for $\neg\bigcirc\, \psi \in \mathit{closure}(\varphi)$, and any direct successor B' of B we have

$$\neq \bigcirc\, \psi \in B \text{ if and only if } \psi \notin B'.$$

(This follows by the definition of δ, local consistency and maximality.) Since $\neg\bigcirc\, a \in B_2$, and B_3 and B_4 are the only states that do not contain a, we have $\delta(B_2, \{\, a \,\}) = \{\, B_3, B_4 \,\}$. Hence, $\delta(B_4, \{\, a \,\}) = \varnothing$ since $B_4 \cap \{\, a \,\} = \varnothing \neq \{\, a \,\}$. Using a similar reasoning as above, we obtain $\delta(B_4, \varnothing) = \{\, B_3, B_4 \,\}$. The outgoing transitions of B_3 are determined analogously. The set $\mathcal{F}$ is empty as $\varphi = \bigcirc a$ does not contain an until operator. Since $\mathcal{F} = \varnothing$, every infinite run in the GNBA $\mathcal{G}_\varphi$ is accepting. As each infinite run is either of the form $B_1\, B_1 \ldots$, $B_1\, B_2 \ldots$, $B_3\, B_1 \ldots$, or $B_3\, B_2 \ldots$, and $\varphi = \bigcirc a \in B_1, B_2$, it follows that indeed all runs satisfy $\bigcirc a$. ■

Example 5.39. Construction of a GNBA (Until)

Consider $\varphi = a \,\mathsf{U}\, b$ and let $AP = \{\, a, b \,\}$. Then

$$\mathit{closure}(\varphi) = \{\, a, b, \neg a, \neg b,\ a \,\mathsf{U}\, b, \neg(a \,\mathsf{U}\, b) \,\}.$$

The construction in the proof of Theorem 5.37 yields the GNBA $\mathcal{G}_\varphi$ illustrated in Figure 5.22. In order not to blur the figure, transition labels have been omitted. (The label of transition $B \to B'$ equals the propositional logic formula characterising the set $B \cap AP$.) The states correspond to the elementary sets of $\mathit{closure}(\varphi)$

$$\begin{array}{lcl} B_1 & = & \{\, a, \quad b, \quad \varphi \,\}, \\ B_2 & = & \{\, \neg a, \quad b, \quad \varphi \,\}, \\ B_3 & = & \{\, a, \quad \neg b, \quad \varphi \,\}, \\ B_4 & = & \{\, \neg a, \quad \neg b, \quad \neg\varphi \,\}, \\ B_5 & = & \{\, a, \quad \neg b, \quad \neg\varphi \,\}. \end{array}$$

The initial states are the sets $B_i \in Q$ with $\varphi \in B_i$; thus, $Q_0 = \{\, B_1, B_2, B_3 \,\}$. The set $\mathcal{F} = \{\, F_\varphi \,\}$ of the accepting sets is a singleton, since φ contains a single until operator.

The set F_φ is given by

$$F_\varphi \;=\; \{\, B \in Q \mid \varphi \notin B \;\vee\; b \in B \,\} \;=\; \{\, B_1, B_2, B_4, B_5 \,\}.$$

Since the accepting set is a singleton set, the GNBA $\mathcal{G}_\varphi$ can be understood as an NBA with the accepting set F_φ.

Let us justify some of the transitions. We have $B_1 \cap AP = \{\, a, b \,\}$ and $a \,\mathsf{U}\, b \in B_1$. As:

$$b \in B_1 \vee (a \in B_1 \wedge a \,\mathsf{U}\, b \in B')$$

holds for any elementary set $B' \subseteq \mathit{closure}(a \,\mathsf{U}\, b)$, we have that $\delta(B_1, \{\, a, b \,\})$ contains all states. These are the only outgoing transitions of state B_1. A similar reasoning applies to $\delta(B_2, \{\, b \,\})$. Consider state B_3. Then $B_3 \cap AP = \{\, a \,\}$. The states B' that are possible direct successors of B_3 under $\{\, a \,\}$ are those satisfying

$$a \in B_3 \wedge a \,\mathsf{U}\, b \in B'.$$

Thus, $\delta(B_3, \{\, a \,\}) = \{\, B_1, B_2, B_3 \,\}$. Finally, consider state B_5. This state only has successors for input symbol $B_5 \cap AP = \{\, a \,\}$. For any $\varphi_1 \,\mathsf{U}\, \varphi_2 \in \mathit{closure}(\varphi)$ it can be shown that for any successor B' of B:

$$\varphi_1 \,\mathsf{U}\, \varphi_2 \notin B \quad \text{iff} \quad \varphi_2 \notin B \;\wedge\; (\varphi_1 \notin B \;\vee\; \varphi_1 \,\mathsf{U}\, \varphi_2 \notin B').$$

(The proof of this fact is left as an exercise.) Applying this to the state B_5 yields that all states not containing φ are possible successors of B_5. For example, the run $B_3\, B_3\, B_1\, B_4^\omega$ is accepting. This run corresponds to the word $\{\, a \,\}\{\, a \,\}\{\, a, b \,\}\varnothing^\omega$ which indeed satisfies $a \,\mathsf{U}\, b$. The word $\{\, a \,\}^\omega$ does not satisfy $a \,\mathsf{U}\, b$. It has exactly one run in $\mathcal{G}_\varphi$, namely B_3^ω. As B_3 is not a final state, this run is not accepting, i.e., $\{\, a \,\}^\omega \notin \mathcal{L}_\omega(\mathcal{G}_\varphi)$. ■

Remark 5.40. Simplified Representation of the Automata States

Any state of the GNBA for an LTL formula φ contains either ψ or its negation $\neg\psi$ for every subformula ψ of φ. This is somewhat redundant. It suffices to represent state $B \in \mathit{closure}(\varphi)$ by the propositional symbols $a \in B \cap AP$, and the formulae $\bigcirc \psi$ or $\varphi_1 \,\mathsf{U}\, \varphi_2 \in B$. ■

Having constructed a GNBA $\mathcal{G}_\varphi$ for a given LTL formula φ, an NBA for φ can be obtained by the transformation "GNBA $\rightsquigarrow$ NBA" described in Theorem 4.56 on page 195. Recall that this transformation for GNBA with two or more acceptance sets generates a copy of $\mathcal{G}_\varphi$ for each acceptance set of $\mathcal{G}_\varphi$. In our case, the number of copies that we need is given by the number of until subformulae of φ. We obtain the following result:

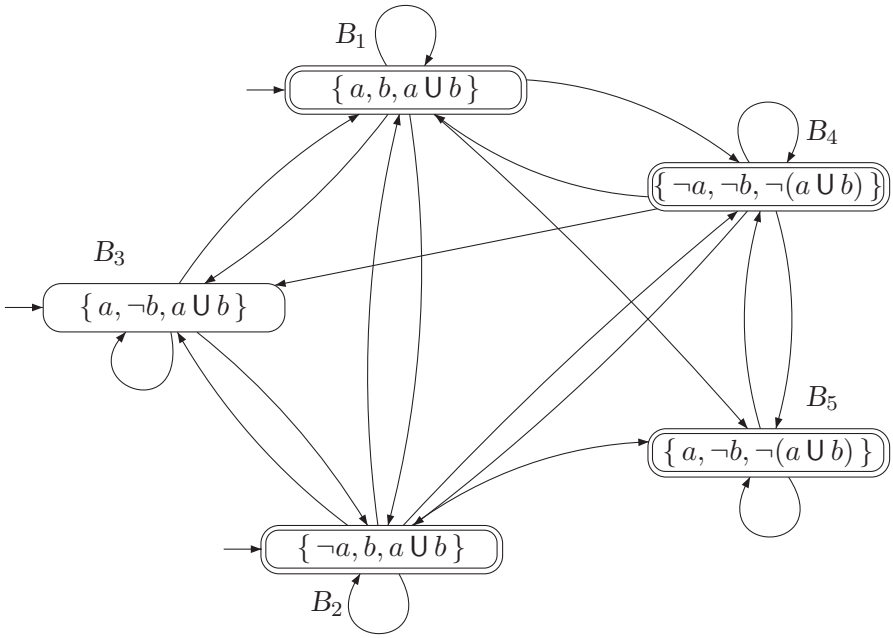


Figure 5.22: A generalised Büchi automaton for $a \mathsf{U} b$.

Theorem 5.41. Constructing an NBA for an LTL Formula

For any LTL formula φ (over AP) there exists an NBA $\mathcal{A}_\varphi$ with $Words(\varphi) = \mathcal{L}_\omega(\mathcal{A}_\varphi)$ which can be constructed in time and space $2^{\mathcal{O}(|\varphi|)}$.

Proof: From Theorem 5.37 (page 278), it follows that a GNBA $\mathcal{G}_\varphi$ can be constructed which has at most $2^{|\varphi|}$ states. As the number of accepting states in $\mathcal{G}_\varphi$ equals the number of until-subformulas in φ, GNBA $\mathcal{G}_\varphi$ has at most $|\varphi|$ accepting states. Transforming the GNBA into an equivalent NBA (as described in the proof of Theorem 4.56, page 195), yields an NBA with at most $|\varphi|$ copies of the state space of $\mathcal{G}_\varphi$. Thus, the number of states in the NBA is at most $2^{|\varphi|} \cdot |\varphi| = 2^{|\varphi| + \log |\varphi|}$ states. This yields the claim. ■

There are various algorithms in the literature for associating an automaton for infinite words to an LTL formula. The presented algorithm is one of the conceptually simplest algorithms, but often yields unnecessarily large GNBAs. For example, for the LTL formulae $\bigcirc a$ and $a \mathsf{U} b$, an NBA with two states suffices. (It is left to the reader to provide these NBAs.) Several optimizations are possible to improve the size of the resulting GNBA, but the exponential blowup cannot be avoided. This is formally stated in the following theorem:

Theorem 5.42. Lower Bound for NBA from LTL Formulae

There exists a family of LTL formulae φ_n with $|\varphi_n| = \mathcal{O}(poly(n))$ such that every NBA for φ_n has at least 2^n states.

Proof: Let AP be an arbitrary nonempty set of atomic propositions, that is, $|2^{AP}| \geqslant 2$. Consider the family of languages:

$$\mathcal{L}_n \;=\; \{\, A_1 \ldots A_n\, A_1 \ldots A_n\, \sigma \mid A_i \subseteq AP \wedge \sigma \in (2^{AP})^\omega \,\}, \qquad \text{for } n \geqslant 0.$$

It is not difficult to check that $\mathcal{L}_n \;=\; \mathit{Words}(\varphi_n)$ where

$$\varphi_n \;=\; \bigwedge_{a \in AP} \; \bigwedge_{0 \leqslant i < n} (\bigcirc^i a \longleftrightarrow \bigcirc^{n+i} a).$$

Here, $\bigcirc^j$ stands for the j-fold application of the next-step operator $\bigcirc$, i.e., $\bigcirc^1 \varphi = \bigcirc \varphi$ and $\bigcirc^{n+1} \varphi = \bigcirc \bigcirc^n \varphi$. It follows that φ_n is an LTL formula of polynomial length. More precisely, $|\varphi_n| \;\in\; \mathcal{O}(|AP| \cdot n)$.

However, any NBA $\mathcal{A}$ with $\mathcal{L}_\omega(\mathcal{A}) = \mathcal{L}_n$ has at least 2^n states. Essentially, this is justified by the following consideration. Since the words

$$A_1 \ldots A_n\, A_1 \ldots A_n\, \varnothing\, \varnothing\, \varnothing \ldots$$

are accepted by $\mathcal{A}$, $\mathcal{A}$ contains for every word $A_1 \ldots A_n$ of length n, a state $q(A_1 \ldots A_n)$, which can be reached from an initial state by consuming the prefix $A_1 \ldots A_n$. Starting from $q(A_1 \ldots A_n)$ it is possible to visit an accept state infinitely often by accepting the suffix $A_1 \ldots A_n\, \varnothing\, \varnothing\, \varnothing \ldots$. If $A_1 \ldots A_n \;\neq\; A'_1 \ldots A'_n$ then

$$A_1 \ldots A_n\, A'_1 \ldots A'_n\, \varnothing\, \varnothing\, \varnothing \ldots \notin \mathcal{L}_n \;=\; \mathcal{L}_\omega(\mathcal{A}).$$

Therefore, the states $q(A_1 \ldots A_n)$ are all pairwise different. As there are $|2^{AP}|$ possible combinations for $A_1 \ldots A_n$, $\mathcal{A}$ has at least $(|2^{AP}|)^n \;\geqslant\; 2^n$ states. ■

Remark 5.43. Büchi Automata are More Expressive Than LTL

The results so far show that for every LTL formula φ an NBA can be constructed that accepts exactly the infinite sequences satisfying φ. We state without proof that the reverse, however, is *not* true. It can be shown that for, e.g., the LT property

$$P \;=\; \Big\{ A_0\, A_1\, A_2 \ldots \in (2^{\{\, a \,\}})^\omega \mid a \in A_{2i} \text{ for } i \geqslant 0 \Big\},$$

which requires a to hold in every even position, there is no LTL formula φ with $\mathit{Words}(\varphi) = P$. On the other hand, there exists an NBA $\mathcal{A}$ with $\mathcal{L}_\omega(\mathcal{A}) = P$. (It is left to the reader to provide such an NBA.) ■

5.2.1 Complexity of the LTL Model-Checking Problem

Let us summarize the results of the previous sections in order to provide an overview of LTL model checking. Subsequently, we discuss the complexity of the LTL model-checking problem.

As explained before, the essential idea behind the automata-based model-checking algorithm for LTL is based upon the following relations:

$$\begin{aligned} TS \models \varphi \quad &\text{iff} \quad \mathit{Traces}(TS) \subseteq \mathit{Words}(\varphi) \\ &\text{iff} \quad \mathit{Traces}(TS) \subseteq (2^{AP})^\omega \setminus \mathit{Words}(\neg\varphi) \\ &\text{iff} \quad \mathit{Traces}(TS) \cap \underbrace{\mathit{Words}(\neg\varphi)}_{\mathcal{L}_\omega(\mathcal{A}_{\neg\varphi})} = \varnothing \\ &\text{iff} \quad TS \otimes \mathcal{A}_{\neg\varphi} \models \Diamond\Box\, \neg F. \end{aligned}$$

Here, NBA $\mathcal{A}_{\neg\varphi}$ accepts $\mathit{Words}(\neg\varphi)$ and F is its set of accept states. The algorithm to transform an LTL formula φ into an NBA may give rise to an NBA whose state space size is exponential in the length of φ. The NBA $\mathcal{A}_{\neg\varphi}$ can thus be constructed in exponential time:

$$\mathcal{O}(2^{|\varphi|} \cdot |\varphi|) \;=\; \mathcal{O}(2^{|\varphi| + \log|\varphi|}).$$

This complexity bound, together with the fact that the state space of $\mathcal{A}$ is exponential in $|\varphi|$, yields an upper bound for the time- and space-complexity of LTL model checking (see Algorithm 11, page 272):

$$\mathcal{O}(\,|TS| \cdot 2^{|\varphi|}\,).$$

Remark 5.44. LTL Model Checking with Fairness

As a consequence of Theorem 5.30 (see page 264), the model-checking problem for LTL with fairness assumptions can be reduced to the model-checking problem for plain LTL. So, in order to check the formula φ under fairness assumption *fair*, it suffices to verify the formula $\mathit{fair} \rightarrow \varphi$ with an LTL model-checking algorithm. This approach, however, has as its main drawback that the length $|\mathit{fair}|$ can have an exponential influence on the run-time of the algorithm. This is due to the construction of an NBA for the negated formula, i.e., $\neg(\mathit{fair} \rightarrow \varphi)$, whose size is exponential in $|\neg(\mathit{fair} \rightarrow \varphi)| \;=\; |\mathit{fair}| + |\varphi|$. To avoid this additional exponential blowup, a modified persistence check (see Algorithm 8 on page 211) can be exploited to analyze the product transition system $TS \otimes \mathcal{A}_{\neg\varphi}$ (instead of $TS \otimes \mathcal{A}_{\neg(\mathit{fair} \rightarrow \varphi)}$). This can be done using standard graph algorithms. The reader is referred to Exercise 5.22 (on page 308) for more details. ■

An interesting aspect of the LTL model-checking algorithm is that it can be executed *on-the-fly*, i.e., while constructing the NBA $\mathcal{A}_{\neg\varphi}$. This may avoid the need for constructing the entire automaton $\mathcal{A}_{\neg\varphi}$. This on-the-fly procedure works as follows. Suppose we are given a high-level description of the transition system TS, e.g., by means of a syntactic description of the concurrent processes (as in SPIN's input language PROMELA). The generation of the reachable states of TS can proceed in parallel with the construction of the relevant fragment of $\mathcal{A}_{\neg\varphi}$. Simultaneously, the reachable fragment of the product transition system $TS \otimes \mathcal{A}_{\neg\varphi}$ is constructed in a DFS-manner. (This, in fact, yields the outermost DFS in the nested DFS for checking persistence in $TS \otimes \mathcal{A}_{\neg\varphi}$.) So the entire LTL model-checking procedure can be interleaved with the generation of the relevant fragments of TS and $\mathcal{A}_{\neg\varphi}$. In this way, the product transition system $TS \otimes \mathcal{A}_{\neg\varphi}$ is constructed "on demand", so to speak. A new vertex is only considered if no accepting cycle has been encountered yet in the partially constructed product transition system $TS \otimes \mathcal{A}_{\neg\varphi}$. When generating the successors of a state in $\mathcal{A}_{\neg\varphi}$, it suffices to only consider the successors matching the current state TS (rather than all possible successors). It is thus possible that an accepting cycle is found, i.e., a violation of φ (with corresponding counterexample), without the need for generating the entire automaton $\mathcal{A}_{\neg\varphi}$.

This on-the-fly generation of $\mathit{Reach}(TS)$, $\mathcal{A}_{\neg\varphi}$, and $TS \otimes \mathcal{A}_{\neg\varphi}$ is adopted in practical LTL model checkers (such as SPIN) and for many examples yields an efficient verification procedure. From a theoretical point of view, though, the LTL model-checking problem remains computationally hard and is "probably" not efficiently solvable. It is shown in the sequel of this section that the LTL model-checking problem is PSPACE-complete. We assume some familiarity with basic notions of complexity theory and the complexity classes coNP and PSPACE; see, e.g., the textbooks [160, 320] and the Appendix.

Before proving the PSPACE-hardness of the LTL model-checking problem, a weaker result is provided. To that end, let us recall the so-called *Hamiltonian path problem*. Consider a finite directed graph $G = (V, E)$ with set V of vertices and set $E \subseteq V \times V$ of edges. The Hamiltonian path problem is a decision problem that yields an affirmative answer when G has a Hamiltonian path, i.e., a path which passes through every vertex in V exactly once.

The next ingredient needed for the following result is the *complement* of the LTL model-checking problem. This decision problem takes as input a finite transition system TS and an LTL formula φ and asks whether $TS \not\models \varphi$. Thus, whenever the LTL model-checking problem provides an affirmative answer, its complement yields "no", and whenever the result of the LTL model-checking problem is negative, its complement yields "yes".

Lemma 5.45.

The Hamiltonian path problem is polynomially reducible to the complement of the LTL model-checking problem.

Proof: To establish a polynomial reduction from the Hamiltonian path problem to the complement of the LTL model-checking problem, a mapping is needed from instances of the Hamiltonian path problem onto instances for the complement model-checking problem. That is, we need to map a finite directed graph G onto a pair (TS, φ) where TS is a finite transition system and φ is an LTL formula, such that

(i) G has a Hamiltonian path if and only if $TS \not\models \varphi$, and

(ii) the transformation $G \rightsquigarrow (TS, \varphi)$ can be performed in polynomial time.

The basic concept of the mapping is as follows. Essentially, TS corresponds to G, i.e., the states of TS are the vertices in G and the transitions in TS correspond to the edges in G. For technical reasons, G is slightly modified to ensure that TS has no terminal states. More precisely, the state graph of TS arises from G by inserting a new vertex b to G, which is reachable from every vertex v via an edge and which is only equipped with a self-loop $b \to b$ (i.e., b has no further outgoing edges). Formally, for $G = (V, E)$ the associated transition system is

$$TS = (V \uplus \{ b \}, \{ \tau \}, \to, V, V, L)$$

where $L(v) = \{ v \}$ for any vertex $v \in V$ and $L(b) = \varnothing$. The atomic propositions for TS are thus obtained by taking the identities of the vertices in G. The transition relation $\to$ is defined as follows:

$$\frac{(v, w) \in E}{v \xrightarrow{\tau} w} \quad \text{and} \quad \frac{v \in V \cup \{ b \}}{v \xrightarrow{\tau} b}.$$

This completes the mapping of directed graph G onto transition system TS. It remains to formalize the negation of the existence (i.e., the absence) of a Hamiltonian path by means of an LTL formula. Let

$$\varphi \;=\; \neg \bigwedge_{v \in V} \big(\Diamond v \;\; \wedge \;\; \Box(v \to \bigcirc \Box \neg v) \big).$$

Stated in words, φ asserts that it is not the case that each vertex $v \in V$ is eventually visited and never visited again. Note that the conjunction does not quantify over $b \notin V$.

It is evident that TS and φ can be constructed in polynomial time for a given directed graph G. As a last step, we need to show that G has a Hamiltonian path if and only if $TS \not\models \varphi$.

$\Leftarrow$: Assume $TS \not\models \varphi$. Then there exists a path π in TS such that

$$\pi \;\models\; \bigwedge_{v \in V} \big(\Diamond v \;\; \wedge \;\; \Box(v \to \bigcirc \Box \neg v) \big).$$

As $\pi \models \bigwedge_{v \in V} \Diamond v$, each vertex $v \in V$ occurs at least once in the path π. Since

$$\pi \models \bigwedge_{v \in V} \Box(v \rightarrow \bigcirc \Box \neg v),$$

there is no vertex that occurs more than once in π. Thus, π is of the form $v_1 \ldots v_n b\ b\ b\ \ldots$ where $V = \{ v_1, \ldots, v_n \}$ and $|V| = n$. In particular, $v_1 \ldots v_n$ is a Hamiltonian path in G.

$\Rightarrow$: Each Hamiltonian path $v_1 \ldots v_n$ in G can be extended to a path $\pi = v_1 \ldots v_n b\ b\ b\ \ldots$ in TS. Thus, $\pi \not\models \varphi$ and, hence, $TS \not\models \varphi$. ■

Since the Hamiltonian path problem is known to be NP-complete, it follows from the previous lemma that the LTL model-checking problem is coNP-hard. The following complexity result states that the LTL model-checking problem is PSPACE-hard.

Theorem 5.46. Lower Bound for LTL Model Checking

The LTL model-checking problem is PSPACE*–hard.*

Proof: Let TS be a finite transition system and φ an LTL formula. As a first observation, note that it suffices to show the PSPACE-hardness of the *existential* variant of the LTL model-checking problem. This existential decision problem takes as input TS and φ and yields "yes" if $\pi \models \varphi$ for *some* (initial, infinite) path π in TS, and "no" otherwise. Given that

$$\begin{aligned} TS \models \varphi \quad & \text{if and only if} \quad \pi \models \varphi \text{ for all paths } \pi \\ & \text{if and only if} \quad \text{not } (\pi \models \neg\varphi \text{ for } \textit{some} \text{ path } \pi) \end{aligned}$$

it follows that the output for the instance (TS, φ) of the LTL model-checking problem yields "yes" if and only if the output for the instance $(TS, \neg\varphi)$ of the existential variant of the LTL model-checking problem yields "no". Thus, the PSPACE-hardness of the existential variant of the LTL model-checking problem yields the PSPACE-hardness of the complement of the existential variant of the LTL model checking problem which again induces the PSPACE-hardness of the LTL model-checking problem. Recall that PSPACE = coPSPACE, thus, the complement of any PSPACE-hard problem is PSPACE-hard.

In the remainder of the proof we concentrate on showing the PSPACE–hardness of the existential LTL model-checking problem. This is established by providing a polynomial reduction from any decision problem $K \in$ PSPACE to the existential LTL model-checking problem. Let $\mathcal{M}$ be a polynomial space-bounded deterministic Turing machine that accepts exactly the words $w \in K$. The goal is now to transform $(\mathcal{M}, w)$ by a deterministic

polynomial-time bounded procedure into a pair (TS, φ) such that $\mathcal{M}$ accepts w if and only if TS contains a path π with $\pi \models \varphi$.

The following proof, adopted from [372], has some similarities with Cook's theorem stating the NP-completeness of the satisfiability problem for propositional logic. The rough idea is to encode the initial configurations, possible transitions, and accepting configurations of a given Turing machine $\mathcal{M}$ by LTL formulae. In the sequel, let $\mathcal{M}$ be a deterministic single-tape Turing machine with state-space Q, starting state $q_0 \in Q$, the set F of accept states, the tape alphabet Σ, and the transition function $\delta : Q \times \Sigma \to Q \times \Sigma \times \{ L, R, N \}$. The intuitive meaning of δ is as follows. Suppose $\delta(q, A) = (p, B, L)$. Then, whenever the current state is q and the current symbol in cell i under the cursor is A, then $\mathcal{M}$ changes to state p, overwrites the symbol A by B in cell i, and moves the cursor one position to the left, i.e., it positions the cursor under cell $i{-}1$. For R or N, the cursor moves one position to the right, or does not move at all, respectively. (For our purposes, there is no need to distinguish between the input and the tape alphabet.)

For technical reasons, it is required that the accept states are absorbing, i.e., $\delta(q, A) = (q, A, N)$ for all $q \in F$. Moreover, it is assumed that $\mathcal{M}$ is *polynomially space-bounded*, i.e., there is a polynomial P such that the computation for an input word $A_1 \dots A_n \in \Sigma^*$ of length n visits at most the first $P(n)$ cells on the tape. Without loss of generality, the coefficients of P are assumed to be natural numbers and $P(n) \geqslant n$.

To encode $\mathcal{M}$'s possible behaviors by LTL formulae for an input word of length n, the Turing machine is transformed into a transition system $TS = TS(\mathcal{M}, n)$ with the following state space:

$$S \;=\; \{0, 1, \dots, P(n)\} \cup \{(q, A, i) \mid q \in Q \cup \{ * \},\, A \in \Sigma,\, 0 < i \leqslant P(n)\}.$$

The structure of TS is shown in Figure 5.23. TS consists of $P(n)$ copies of "diamonds". The ith copy starts in state $i{-}1$, ends in state i, and contains the states (q, A, i) where $q \in Q \cup \{ * \}$ and $A \in \Sigma$ "between" state $i{-}1$ and state i. Intuitively, the first component $q \in Q \cup \{ * \}$ of state (q, A, i) in the ith copy indicates whether the cursor points to cell i (in which case $q \in Q$ is the current state) or to some other tape cell (in which case $q = *$). The symbol $A \in \Sigma$ in state (q, A, i) stands for the current symbol in cell i. The path fragments from state 0 to state $P(n)$ serve to represent the possible configurations of $\mathcal{M}$. More precisely, the configuration in which the current content of the tape is $A_1\, A_2 \dots A_{P(n)}$, the current state is q, and the cursor points to cell i is encoded by the path fragment:

$$0\,(*, A_1, 1)\,1\,(*, A_2, 2)\,2\;\dots\;i{-}1\,(q, A_i, i)\,i\,(*, A_{i+1}, i{+}1)\,i{+}1\;\dots\;P(n)$$

Accordingly, the computation of $\mathcal{M}$ for an input word of length n can be described by the concatenation of such path fragments.

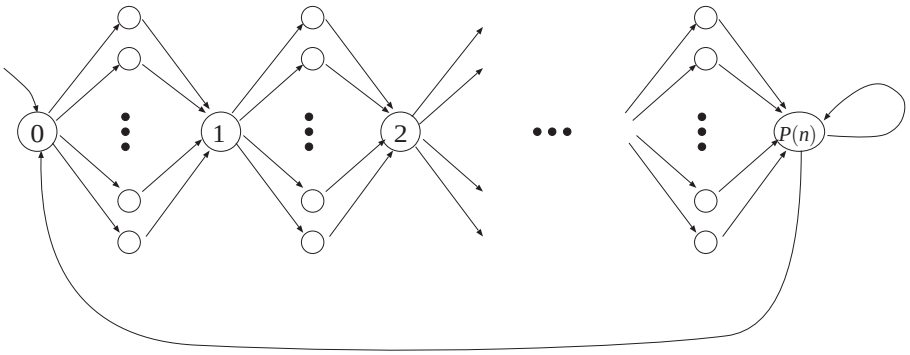


Figure 5.23: Transition system $TS(\mathcal{M}, n)$ for Turing machine $\mathcal{M}$ and input length n.

We use the state identities as atomic propositions. In addition, proposition *begin* is used to identify state 0, while proposition *end* is used for state $P(n)$. That is, $AP = S \cup \{ \mathit{begin}, \mathit{end} \}$ with the obvious labeling function. Let Φ_Q denote the disjunction over all atoms (q, A, i) where $q \in Q$ (i.e., $q \neq *$) and $A \in \Sigma$, $0 < i \leqslant P(n)$. The LTL formulae:

$$\begin{aligned}
\varphi_{\mathit{Conf}} &= \Box\Big(\mathit{begin} \longrightarrow \varphi^1_{\mathit{Conf}} \wedge \varphi^2_{\mathit{Conf}}\Big) \quad \text{where} \\
\varphi^1_{\mathit{Conf}} &= \bigvee_{1 \leqslant i \leqslant P(n)} \bigcirc^{2i-1} \Phi_Q \\
\varphi^2_{\mathit{Conf}} &= \bigwedge_{1 \leqslant i \leqslant P(n)} \Big(\bigcirc^{2i-1} \Phi_Q \longrightarrow \bigwedge_{\substack{1 \leqslant j \leqslant P(n) \\ j \neq i}} \bigcirc^{2j-1} \neg \Phi_Q \Big)
\end{aligned}$$

characterize any path π in TS such that all path fragments of π that lead from 0 via $1, \ldots, P(n{-}1)$ to $P(n)$ encode a configuration of $\mathcal{M}$. Note that $\varphi^1_{\mathit{Conf}} \wedge \varphi^2_{\mathit{Conf}}$ ensures that the cursor points exactly to one of the positions $1, \ldots, P(n)$. Thus, any path π in TS such that $\pi \models \varphi_{\mathit{Conf}}$ can be viewed as a sequence of configurations in $\mathcal{M}$. However, this sequence need not be a computation of $\mathcal{M}$, since the configurations might not be consecutive according to $\mathcal{M}$'s operational behavior. For this, we need additional constraints that formalize $\mathcal{M}$'s stepwise behavior.

The transition relation δ of $\mathcal{M}$ can be encoded by an LTL formula φ_δ that arises through a conjunction of formulae $\varphi_{q,A}$ describing the semantics of $\delta(q, A)$. Here, q ranges over all states in Q and A over the tape-symbols in Σ. For instance, if $\delta(q, A) = (p, B, L)$ then

$$\varphi_{q,A} = \Box \bigwedge_{1 \leqslant i \leqslant P(n)} \Big(\bigcirc^{2i-1} (q, A, i) \longrightarrow \psi_{(q,A,i,p,B,L)} \Big)$$

where $\psi_{(q,A,i,p,B,L)}$ is defined as

$$\bigwedge_{\substack{1\leqslant j\leqslant P(n)\\ j\neq i, C\in\Sigma}} (\underbrace{\bigcirc^{2j-1} C \leftrightarrow \bigcirc^{2j-1+2P(n)+1} C}_{\text{content of all cells } j\neq i \text{ unchanged}}) \;\wedge\; \underbrace{\bigcirc^{2i-1+2P(n)+1} B}_{\text{overwrite } A \text{ by } B \text{ in cell } i} \wedge \underbrace{\bigcirc^{2i-1+2P(n)+1-2} p}_{\substack{\text{move to state } p\\ \text{and cursor to cell } i-1}}.$$

Here, C denotes the disjunction of the atoms (r, C, j) where $r \in Q \cup \{*\}$ and $1 \leqslant j \leqslant P(n)$, and p for the disjunction of all atoms (p, D, j) where $D \in \Sigma$ and $1 \leqslant j \leqslant P(n)$.

The starting configuration of $\mathcal{M}$ for a given input word $w = A_1 \ldots A_n \in \Sigma^*$ is given by the formula

$$\varphi^w_{start} \;=\; \bigcirc q_0 \;\wedge \bigwedge_{1\leqslant i\leqslant n} \bigcirc^{2i-1} A_i \;\wedge \bigwedge_{n<i\leqslant P(n)} \bigcirc^{2i-1} \sqcup\,.$$

The first conjunct $\bigcirc q_0$ asserts that $\mathcal{M}$ starts in its starting state; the other conjuncts assert that $A_1 \ldots A_n \sqcup^{P(n)-n}$ is the content of the tape where $\sqcup$ denotes the blank symbol. The accepting configurations are formalized by the formula

$$\varphi_{accept} = \Diamond \bigvee_{q\in F} q.$$

For a given input word $w = A_1 \ldots A_n$ of length n for $\mathcal{M}$, let

$$\varphi_w \;=\; \varphi^w_{start} \;\wedge\; \varphi_{Conf} \;\wedge\; \varphi_\delta \;\wedge\; \varphi_{accept}.$$

Note that the length of φ_w is polynomial in the size of $\mathcal{M}$ and the length n of w. Thus, $TS = TS(\mathcal{M}, n)$ and φ_w can be constructed from $(\mathcal{M}, w)$ in polynomial time. Moreover, it also follows that there exists a path $\pi \models \varphi_w$ in TS if and only if $\mathcal{M}$ accepts the input word w. ■

For real applications, the aforementioned theoretical complexity result is less dramatic than it seems at first sight, since the complexity is *linear* in the size of the transition system and exponential in formula length. In practice, typical requirement specifications yield short LTL formulae. In fact, the exponential growth in the length of the formula is not decisive for the practical application of LTL model checking. Instead, the linear dependency on the size of the transition system is the critical factor. As has been discussed at the end of Chapter 2 (page 77), the size of transition systems may be huge even for relatively simple systems—the state-space explosion problem. In later chapters of this monograph, various techniques will be treated to combat this state-space explosion problem.

We mentioned before that the LTL model-checking problem is PSPACE-complete. The previous result showed PSPACE-hardness. It remains to show that it belongs to the complexity class PSPACE.

To prove that the LTL model-checking problem is in PSPACE, we resort (again) to the existential variant of the LTL model-checking problem. This is justified by the fact that PSPACE – as any other deterministic complexity class – is closed under complementation. That is, the LTL model-checking problem is in PSPACE iff its complement is in PSPACE. This is equivalent to the statement that the existential variant of the LTL model-checking problem is solvable by a polynomial space-bounded algorithm. By Savitch's theorem (PSPACE agrees with NPSPACE), it suffices to provide a *nondeterministic polynomial space-bounded* algorithm that solves the existential LTL model-checking problem.

Lemma 5.47.

The existential LTL model-checking problem is solvable by a nondeterministic space-bounded algorithm.

Proof: In the sequel, let φ be an LTL formula and $TS = (S, Act, \to, I, AP, L)$ a finite transition system. The goal is to check nondeterministically whether TS has a path π with $\pi \models \varphi$, while the memory requirements are bounded by $\mathcal{O}(\,\mathrm{poly}(\mathit{size}(TS), |\varphi|)\,)$. The techniques discussed in Section 5.2 (page 270 ff) suggest to build an NBA $\mathcal{A}_\varphi$ for φ, construct the product transition system $TS \otimes \mathcal{A}_\varphi$ and check whether this product contains a reachable cycle containing an accept state of $\mathcal{A}_\varphi$. We now modify this approach to obtain an NPSPACE algorithm. Instead of an NBA for φ, we deal here with the GNBA $\mathcal{G}_\varphi$ for φ. Recall that states in this automaton are elementary subsets of the closure of φ (see Definition 5.34 on page 276). The goal is to guess nondeterministically a finite path $u_0\, u_1\, \ldots\, u_{n-1}\, v_0\, v_1\, \ldots\, v_{m-1}$ in $TS \otimes \mathcal{G}_\varphi$ and to check whether the components of $\mathcal{G}_\varphi$ in the infinite path

$$u_0\, u_1\, \ldots\, u_{n-1}\, (v_0\, v_1\, \ldots\, v_{m-1})^\omega$$

constitute an accepting run in $\mathcal{G}_\varphi$. This, of course, requires that v_0 is a successor of v_{m-1}. The states u_i, v_j in the infinite path in $TS \otimes \mathcal{G}_\varphi$ are pairs consisting of a state in TS and an elementary set of formulae. For the lengths of the prefix $u_0 \ldots u_{n-1}$ and the cycle $v_0\, v_1 \ldots v_{m-1}\, v_m$ we can deal with $n \leqslant k$ and $m \leqslant k \cdot |\varphi|$ where k is the number of reachable states in $TS \otimes \mathcal{G}_\varphi$. (Note that $|\varphi|$ is an upper bound for the number of acceptance sets in $\mathcal{G}_varphi$.) An upper bound for the value k is given by

$$K \;=\; N_{TS} \cdot 2^{N_\varphi}$$

where N_{TS} denotes the number of states in TS and $N_\varphi = |\mathit{closure}(\varphi)|$. Note that

$$K \;=\; \mathcal{O}(\,\mathit{size}(TS) \cdot \exp(|\varphi|)\,).$$

The algorithm now works as follows. We first nondeterministically choose two natural numbers n, m with $n \leqslant K$ and $m \leqslant K \cdot |\varphi|$ (by guessing $\lceil \log K \rceil \;=\; \mathcal{O}(\log(\mathit{size}(TS)) \cdot |\varphi|)$

bits for n and $\lceil \log K \rceil + \lceil \log |\varphi| \rceil = \mathcal{O}(\log(\mathit{size}(TS)) \cdot |\varphi|)$ bits for m). Then the algorithm guesses nondeterministically a sequence $u_0 \ldots u_{n-1}$, $u_n \ldots u_{n+m}$ where the u_i's are pairs $\langle s_i, B_i \rangle$ consisting of a state s_i in TS and a subset B_i of $\mathit{closure}(\varphi)$. For each such state $u_i = \langle s_i, B_i \rangle$, the algorithm checks whether

1. s_i is a successor of s_{i-1}, provided that $i \geqslant 1$,
2. B_i is elementary,
3. $B_i \cap AP = L(s_i)$,
4. $B_i \in \delta(B_{i-1}, L(s_i))$, provided that $i \geqslant 1$.

For $i = 0$, the algorithm checks whether $s_0 \in I$ is an initial state of TS and whether $B_0 \in \delta(B, L(s_0))$ for some elementary set B which contains φ. Here, δ denotes the transition relation of the GNBA $\mathcal{G}_\varphi$. (Recall that the sets B where $\varphi \in B$ are the initial states in the GNBA for φ.) Conditions 1-4 are local and simple to check. If one of these four conditions is violated for some i, the algorithm rejects and halts. Otherwise $u_0 \ldots u_{n+m}$ is a finite path in $TS \otimes \mathcal{G}_\varphi$. We finally check whether u_n agrees with the last state u_{n+m}. Again, the algorithm rejects and halts if this condition does not hold. Otherwise, $u_0 \ldots u_{n-1}(u_n \ldots u_{n+m-1})^\omega$ is an infinite path in $TS \otimes \mathcal{G}_\varphi$, and it finally amounts to check whether the acceptance condition of $\mathcal{G}_\varphi$ is fulfilled. This means we have to verify that whenever $\psi_1 \,\mathsf{U}\, \psi_2 \in B_i$ for some $i \in \{n, \ldots, n+m-1\}$, then there is some $j \in \{n, \ldots, n+m-1\}$ such that $\psi_2 \in B_j$. If this condition holds, then the algorithm terminates with the answer "yes".

This algorithm is correct since if TS has a path where φ holds, then there is a computation of the above algorithm that returns "yes". Otherwise, i.e., if TS does not have a path where φ holds, then all computations of the algorithm are rejecting.

It remains to explain how the sketched algorithm can be realized such that the memory requirements are polynomial in the size of TS and length of φ. Although the length $n+m$ of the path $u_0 \ldots u_{n+m}$ might be exponentially in the length of φ (note that, e.g., $n = K$ is possible and that K grows exponentially in the length of φ), this procedure can be realized with only polynomial space requirements. This is due to the observation that for checking the above conditions 1-4 for $u_i = \langle s_i, B_i \rangle$, we only need state $u_{i-1} = \langle s_{i-1}, B_{i-1} \rangle$. Thus, there is no need to store all states u_j for $0 \leqslant j \leqslant n+m$. Instead, the actual and previous states are sufficient. Moreover, to verify that $\mathcal{G}_\varphi$'s acceptance condition holds, we only need to remember the subformulae $\psi_1 \,\mathsf{U}\, \psi_2$ of φ that are contained in some of the sets $B_n, \ldots B_{n+m-1}$, and the subformulae ψ_2 that appear on the right hand side of an until subformula of φ and are contained in some of the sets $B_n, \ldots B_{n+m-1}$. This additional

information requires $\mathcal{O}(|\varphi|)$ space. Thus, the above nondeterministic algorithm can be realized in a polynomially space-bounded way. ■

By Lemma 5.47 and Theorem 5.46 we get:

Theorem 5.48.

The LTL model-checking problem is PSPACE*-complete.*

5.2.2 LTL Satisfiability and Validity Checking

The last section of this chapter considers the satisfiability problem and the validity problem for LTL. The satisfiability problem is: for a given LTL formula φ, does there exist a model for which φ holds? That is, do we have $\mathit{Words}(\varphi) \neq \varnothing$? Satisfiability can be solved by constructing an NBA $\mathcal{A}_\varphi$ for LTL formula φ. In this way, the existence of an infinite word $\sigma \in \mathit{Words}(\varphi) = \mathcal{L}_\omega(\mathcal{A}_\varphi)$ can be established. The emptiness problem for NBA $\mathcal{A}$, i.e., whether $\mathcal{L}_\omega(\mathcal{A}) = \varnothing$ or not, can be solved by means of a technique similar to persistence checking, see Algorithm 12. In addition to an affirmative response, a prefix of a word $\sigma \in \mathcal{L}_\omega(\mathcal{A}) = \mathit{Words}(\varphi)$ can be provided similar to a counterexample for model checking LTL.

Algorithm 12 Satisfiability checking for LTL

Input: LTL formula φ over AP
Output: "yes" if φ is satisfiable. Otherwise "no".

Construct an NBA $\mathcal{A} = (Q, 2^{AP}, \delta, Q_0, F)$ with $\mathcal{L}_\omega(\mathcal{A}) = \mathit{Words}(\varphi)$

(* Check whether $\mathcal{L}_\omega(\mathcal{A}) = \varnothing$. *)

Perform a nested DFS to determine whether there exists a state $q \in F$ reachable from $q_0 \in Q_0$ and that lies on a cycle

If so, then return "yes". Otherwise, "no".

Given that satisfiability for LTL can be tackled using similar techniques as for model checking LTL, let us now consider the validity problem. Formula φ is *valid* whenever φ holds under all interpretations, i.e., $\varphi \equiv$ true. For LTL formula φ over AP we have φ is valid if and only if $\mathit{Words}(\varphi) = (2^{AP})^\omega$. The validity of φ can be established by using the

observation that φ is valid if and only if $\neg\varphi$ is not satisfiable. Hence, to algorithmically check whether φ is obtained, one constructs an NBA for $\neg\varphi$ and applies the satisfiablity algorithm (see Algorithm 12) to $\neg\varphi$.

The outlined LTL satisfiability algorithm has a runtime that is exponential in the length of φ. The following result shows that an essentially more efficient technique cannot be achieved as both the validity and satisfiability problems are PSPACE-hard. In fact, both problems are even PSPACE-complete. Membership to PSPACE for the LTL satisfiability problem can be shown by providing a nondeterministic polynomially space-bounded algorithm that guesses a finite run in the GNBA $\mathcal{G}_\varphi$ for the given formula φ and checks whether this finite run is a prefix of an accepting run in $\mathcal{G}_\varphi$. The details are skipped here since they are very similar to the algorithm we provided for the existential LTL model checking problem (see Lemma 5.47 on page 294). The fact that the LTL validity problem belongs to PSPACE can be derived from the observations that φ is valid iff $\neg\varphi$ is not satisfiable and that PSPACE is closed under complementation. We now focus on the proof for the PSPACE-hardness.

Theorem 5.49. LTL Satisfiability and Validity (Lower Bound)

The satisfiability and validity problems for LTL are PSPACE*-hard.*

Proof: Since satisfiability and validity are complementary in the sense that φ is satisfiable if and only if $\neg\varphi$ is not valid, and vice versa, it suffices to show the PSPACE-hardness of the satisfiability problem. This is done by providing a polynomial reduction from the existential variant of the LTL model-checking problem (see the proof of Theorem 5.46 on page 290) to the satisfiability problem for LTL.

Let $TS = (S, Act, \to, I, AP, L)$ be a finite transition system and φ an LTL formula over AP. The goal is to construct an LTL formula ψ such that ψ is satisfiable if and only if there is a path π in TS with $\pi \models \varphi$. Besides, ψ should be constructed in polynomial time.

The atomic propositions in ψ are elements in $AP' = AP \uplus S$. For any state $s \in S$ let

$$\Phi_s = \bigwedge_{a \in L(s)} a \ \wedge \ \bigwedge_{a \notin L(s)} \neg a.$$

The formula Φ_s can be viewed as a characteristic formula for the labeling of s, since $s' \models \Phi_s$ if and only if $L(s) = L(s')$, for any $s' \in S$. However, for the LTL satisfiability problem, there is no fixed transition system and Φ_s can hold also for other states (in another transition system). For $s \in S$ and $T \subseteq S$, let $\Psi_T = \bigvee_{t \in T} t$ be the characteristic formula for the set T. Let

$$\psi_s \ = \ s \to (\Phi_s \ \wedge \ \bigcirc \ \Psi_{Post(s)})$$

assert that in state s, the labels $L(s)$ hold, and that transitions exist to any of its immediate successors $\mathit{Post}(s)$. Let

$$\Xi \;=\; \bigvee_{s \in S} (s \;\wedge \bigwedge_{t \in S \setminus \{\, s \,\}} \neg t).$$

Stated in words, Ξ asserts that exactly one of the atomic propositions $s \in AP'$ holds. These definitions constitute the ingredients for the definition of ψ. For set I of initial states, let

$$\psi \;=\; \Psi_I \;\wedge\; \Box\,\Xi \;\wedge\; \Box\,\Psi_S \;\wedge\; \bigwedge_{s \in S} \Box \psi_s \;\wedge\; \varphi.$$

It follows directly that ψ can be derived from TS and φ in polynomial time. It remains to show that

$$\exists \pi \in \mathit{Paths}(TS).\, \pi \models \varphi \quad \text{if and only if} \quad \psi \text{ is satisfiable.}$$

$\Rightarrow$: Let $\pi = s_0\, s_1\, s_2 \ldots$ be an initial, infinite path in TS with $\pi \models \varphi$. Now consider π as a path in the transition system TS' that agrees with TS but uses the extended labeling function $L'(s) = L(s) \cup \{\, s \,\}$. Then, $\pi \models \Psi_I$, since $s_0 \in I$. In addition, $\pi \models \Box\Xi$ and $\pi \models \Box\psi_s$ since Ξ and ψ_s hold in all states of TS'. Thus, $\pi \models \psi$. Hence, π is a witness for the satisfiability of ψ.

$\Leftarrow$: Assume ψ is satisfiable. Let $A_0\, A_1\, A_2 \ldots$ be an infinite word over the alphabet $2^{AP'}$ with $A_0\, A_1\, A_2 \ldots \in \mathit{Words}(\psi)$. Since

$$A_0\, A_1\, A_2 \ldots \models \Box\Xi$$

there is a unique state sequence $\pi = s_0\, s_1\, s_2 \ldots$ in TS with $s_i \in A_i$ for all $i \geqslant 0$. Since $A_0\, A_1\, A_2 \ldots \models \Psi_I$, we get $s_0 \in I$. As

$$A_0\, A_1\, A_2 \ldots \models \Box \bigwedge_{s \in S} \psi_s,$$

we have $A_i \cap AP = L(s_i)$ and $s_{i+1} \in \mathit{Post}(s_i)$ for all $i \geqslant 0$. This yields that π is a path in TS and $\pi \models \varphi$. ∎

5.3 Summary

- Linear Temporal Logic (LTL) is a logic for formalizing path-based properties.
- LTL formulae can be transformed algorithmically into nondeterministic Büchi automata (NBA). This transformation can cause an exponential blowup.

- The presented algorithm for the construction of an NBA for a given LTL formula φ relies on first constructing a GNBA for φ, which is then transformed into an equivalent NBA.
- The GNBA for φ encodes the semantics of propositional logic and the semantics of the next-step operator in its transitions. Based on the expansion law, the meaning of until is split into local requirements (encoded by the states of a GNBA), next-step requirements (encoded by the transitions of the GNBA), and a fairness condition (encoded by the acceptance sets of the GNBA).
- LTL formulae describe ω-regular LT properties, but do not have the same expressiveness as ω-regular languages.
- The LTL model-checking problem can be solved by a nested depth-first search in the product of the given transition system and an NBA for the negated formula.
- The time complexity of the automata-based model-checking algorithm for LTL is linear in the size of the transition system and exponential in the length of the formula.
- Fairness assumptions can be described by LTL formulae. The model-checking problem for LTL with fairness assumptions is reducible to the standard LTL model-checking problem.
- The LTL model-checking problem is PSPACE-complete.
- Satisfiability and validity of LTL formulae can be solved via checking emptiness of nondeterministic Büchi automata. The emptiness check can be determined by a nested DFS that checks the existence of a reachable cycle containing an accept state. Both problems are PSPACE-complete.

5.4 Bibliographic Notes

Linear temporal logic. Based on prior work on modal logics and temporal modalities [270, 345, 244, 230], Pnueli introduced (linear) temporal logics for reasoning about reactive systems in the late seventies in his seminal paper [337]. Since then, a variety of variants and extensions of LTL have been investigated, such as Lamport's Temporal Logic of Actions (TLA) [260] and LTL with past operators [159, 274, 262]. LTL forms the basis for the recently standardized industrial property specification language PSL [136]. The past extension of LTL does not change the expressiveness of LTL, but can be helpful for specification convenience and modular reasoning. For several properties, the use of past operators may lead to (exponentially) more succinct formulae than in ordinary LTL. To

cover the full class of ω-regular LT properties, Vardi and Wolper introduced an extension of LTL by automata formulae [424, 425, 411, 412].

LTL model checking. Vardi and Wolper also developed the automata-based model checking algorithm for LTL presented in this chapter. The presented algorithm to construct an NBA from a given LTL formula is, in our opinion, the simplest and most intuitive one. In the meantime, various alternative techniques have been developed that generate more compact NBAs or that attempt to minimize a given NBA, see e.g. [166, 110, 148, 375, 162, 149, 167, 157, 389, 369]. Alternative LTL model-checking algorithms that do not use Büchi automata, but a so-called tableau for the LTL formula, were presented by Lichtenstein and Pnueli [273] and Clarke, Grumberg, and Hamaguchi [88]. The results about the complexity of LTL model checking and the satisfiability problem are due to Sistla and Clarke [372].

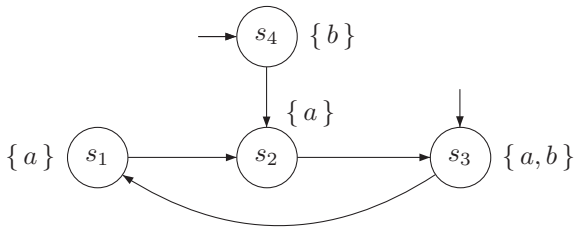
There is a variety of surveys and textbooks; see, e.g.,[245, 138, 173, 283, 158, 284, 92, 219, 379, 365], where several other aspects of LTL and related logics, such as deductive proof systems, alternative model-checking algorithms, or more details about the expressiveness, are treated.

Examples. The garbage collection algorithm presented in Example 5.31 is due to Ben-Ari [41]. Several leader election protocols that fit into the shape of Example 5.13 have been suggested; see, e.g., [280].

LTL model checkers. SPIN is the most well-known LTL model checker and has been developed by Holzmann [209]. Transition systems are described in the modeling language Promela, and LTL formulae are checked using the algorithm advocated by Gerth et al. [166]. LTL model checking using a tableau construction is supported by NuSMV [83].

5.5 Exercises

Exercise 5.1. Consider the following transition system over the set of atomic propositions $\{ a, b \}$:



Indicate for each of the following LTL formulae the set of states for which these formulae are

fulfilled:

(a) $\bigcirc\, a$

(b) $\bigcirc\, \bigcirc\, \bigcirc\, a$

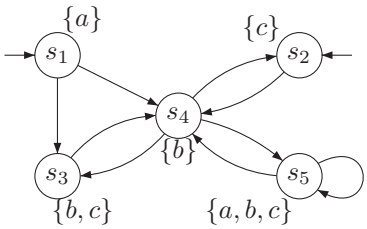
(c) $\Box\, b$

(d) $\Box \Diamond\, a$

(e) $\Box\, (b \,\mathsf{U}\, a)$

(f) $\Diamond\, (a \,\mathsf{U}\, b)$

Exercise 5.2. Consider the transition system TS over the set of atomic propositions $AP = \{\, a, b, c \,\}$:



Decide for each of the LTL formulae φ_i below, whether $TS \models \varphi_i$ holds. Justify your answers! If $TS \not\models \varphi_i$, provide a path $\pi \in \mathit{Paths}(TS)$ such that $\pi \not\models \varphi_i$.

$$
\begin{aligned}
\varphi_1 &= \Diamond \Box\, c \\
\varphi_2 &= \Box \Diamond\, c \\
\varphi_3 &= \bigcirc\, \neg c \rightarrow \bigcirc\, \bigcirc\, c \\
\varphi_4 &= \Box\, a \\
\varphi_5 &= a \,\mathsf{U}\, \Box\, (b \vee c) \\
\varphi_6 &= (\bigcirc\, \bigcirc\, b) \,\mathsf{U}\, (b \vee c)
\end{aligned}
$$

Exercise 5.3. Consider the sequential circuit in Figure 5.24 and let $AP = \{\, x, y, r_1, r_2 \,\}$. Provide LTL formulae for the following properties:

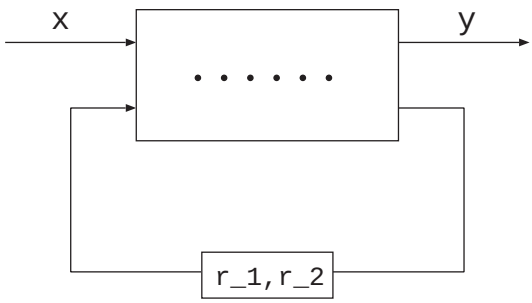


Figure 5.24: Circuit for Exercise 5.3.

(a) "It is impossible that the circuit outputs two successive 1s."

(b) "Whenever the input bit is 1, in at most two steps the output bit will be 1."

(c) "Whenever the input bit is 1, the register bits do not change in the next step."

(d) "Register r_1 has infinitely often the value 1."

Determine which of these properties are satisfied for the initial register evaluation where $r_1 = 0$ and $r_2 = 0$? Justify your answers.

EXERCISE 5.4. Suppose we have two users, *Peter* and *Betsy*, and a single printer device *Printer*. Both users perform several tasks, and every now and then they want to print their results on the *Printer*. Since there is only a single printer, only one user can print a job at a time. Suppose we have the following atomic propositions for *Peter* at our disposal:

- *Peter.request* ::= indicates that *Peter* requests usage of the printer;
- *Peter.use* ::= indicates that *Peter* uses the printer;
- *Peter.release* ::= indicates that *Peter* releases the printer.

For *Betsy*, similar predicates are defined. Specify in LTL the following properties:

(a) Mutual exclusion, i.e., only one user at a time can use the printer.

(b) Finite time of usage, i.e., a user can print only for a finite amount of time.

(c) Absence of individual starvation, i.e., if a user wants to print something, he/she eventually is able to do so.

(d) Absence of blocking, i.e., a user can always request to use the printer

(e) Alternating access, i.e., users must strictly alternate in printing.

EXERCISE 5.5. Consider an elevator system that services $N > 0$ floors numbered 0 through $N-1$. There is an elevator door at each floor with a call-button and an indicator light that signals whether or not the elevator has been called. For simplicity consider $N = 4$. Present a set of atomic propositions – try to minimize the number of propositions – that are needed to describe the following properties of the elevator system as LTL formulae and give the corresponding LTL formulae:

(a) The doors are "safe", i.e., a floor door is never open if the elevator is not present at the given floor.

(b) A requested floor will be served sometime.

(c) Again and again the elevator returns to floor 0.

(d) When the top floor is requested, the elevator serves it immediately and does not stop on the way there.

EXERCISE 5.6. Which of the following equivalences are correct? Prove the equivalence or provide a counterexample that illustrates that the formula on the left and the formula on the right are not equivalent.

(a) $\Box\varphi \rightarrow \Diamond\psi \quad \equiv \quad \varphi \,\mathsf{U}\, (\psi \vee \neg\varphi)$

(b) $\Diamond\Box\varphi \rightarrow \Box\Diamond\psi \quad \equiv \quad \Box\big(\varphi \,\mathsf{U}\, (\psi \vee \neg\varphi)\big)$

(c) $\Box\Box(\varphi \vee \neg\psi) \;\equiv\; \neg\Diamond(\neg\varphi \wedge \psi)$

(d) $\Diamond(\varphi \wedge \psi) \;\equiv\; \Diamond\varphi \;\wedge\; \Diamond\psi$

(e) $\Box\varphi \;\wedge\; \bigcirc\Diamond\varphi \;\equiv\; \Box\varphi$

(f) $\Diamond\varphi \;\wedge\; \bigcirc\Box\varphi \;\equiv\; \Diamond\varphi$

(g) $\Box\Diamond\varphi \rightarrow \Box\Diamond\psi \;\equiv\; \Box\big(\varphi \rightarrow \Diamond\psi\big)$

(h) $\neg(\varphi_1 \,\mathsf{U}\, \varphi_2) \;\equiv\; \neg\varphi_2 \,\mathsf{W}\, (\neg\varphi_1 \wedge \neg\varphi_2)$

(i) $\bigcirc\Diamond\varphi_1 \equiv \Diamond\bigcirc\varphi_2$

(j) $(\Diamond\Box\varphi_1) \wedge (\Diamond\Box\varphi_2) \;\equiv\; \Diamond(\Box\varphi_1 \wedge \Box\varphi_2)$

(k) $(\varphi_1 \,\mathsf{U}\, \varphi_2) \,\mathsf{U}\, \varphi_2 \;\equiv\; \varphi_1 \,\mathsf{U}\, \varphi_2$

EXERCISE 5.7. Let φ and ψ be LTL formulae. Consider the following new operators:

(a) "At next" $\varphi \,\mathsf{N}\, \psi$: at the next time where ψ holds, φ also holds.

(b) "While" $\varphi \,\mathsf{W}\, \psi$: φ holds as least as long as ψ does.

(c) "Before" $\varphi \,\mathsf{B}\, \psi$: if ψ holds sometime, φ does so before.

Make the definitions of these informally explained operators precise by providing LTL formulae that formalize their intuitive meanings.

EXERCISE 5.8. We consider the release operator R which was defined by $\varphi \,\mathsf{R}\, \psi \stackrel{\text{def}}{=} \neg(\neg\varphi \,\mathsf{U}\, \neg\psi)$; see Section 5.1.5 on page 252 ff.

(a) Prove the expansion law $\varphi_1 \,\mathsf{R}\, \varphi_2 \;\equiv\; \varphi_2 \wedge (\varphi_1 \vee \bigcirc(\varphi_1 \wedge \varphi_2))$.

(b) Prove that $\varphi \,\mathsf{R}\, \psi \equiv (\neg\varphi \wedge \psi)\,\mathsf{W}\,(\varphi \wedge \psi)$.

(c) Prove that $\varphi_1 \,\mathsf{W}\, \varphi_2 \;\equiv\; (\neg\varphi_1 \vee \varphi_2)\,\mathsf{R}\,(\varphi_1 \vee \varphi_2)$.

(d) Prove that $\varphi_1 \,\mathsf{U}\, \varphi_2 \;\equiv\; \neg(\neg\varphi_1 \,\mathsf{R}\, \neg\varphi_2)$.

EXERCISE 5.9. Consider the LTL formula

$$\varphi \;=\; \neg\Big((\Box a) \rightarrow \big((a \wedge \neg c)\,\mathsf{U}\,\neg(\bigcirc b)\big)\Big) \wedge \neg(\neg a \vee \bigcirc \Diamond c).$$

Transform φ into an equivalent LTL formula in PNF

(a) using the weak-until operator W,

(b) using the release operator R.

EXERCISE 5.10. Provide an example for a sequence (φ_n) of LTL formulae such that the LTL formula ψ_n is in weak-until PNF, $\varphi_n \equiv \psi_n$, and ψ_n is exponentially longer than φ_n. Use the transformation rules in Section 5.1.5

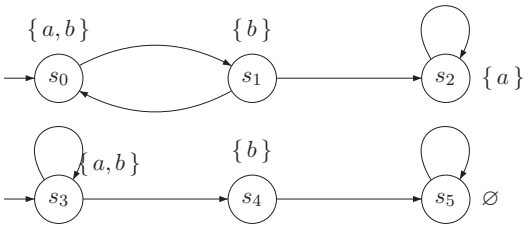


Figure 5.25: Transition system for Exercise 5.11.

EXERCISE 5.11. Consider the transition system TS in Figure 5.25 with the set $AP = \{ a, b, c \}$ of atomic propositions. Note that this is a single transition system with two initial states. Consider the LTL fairness assumption

$$\mathit{fair} \;=\; \big(\Box\Diamond(a \wedge b) \;\rightarrow\; \Box\Diamond\neg c\big) \;\;\wedge\;\; \big(\Diamond\Box(a \wedge b) \rightarrow \Box\Diamond\neg b\big).$$

Questions:

(a) Determine the fair paths in TS, i.e., the initial, infinite paths satisfying *fair*

(b) For each of the following LTL formulae:

$$\begin{aligned}
\varphi_1 &= \Diamond\Box a \\
\varphi_2 &= \bigcirc\neg a \;\longrightarrow\; \Diamond\Box a \\
\varphi_3 &= \Box a \\
\varphi_4 &= b\,\mathsf{U}\;\Box\neg b \\
\varphi_5 &= b\,\mathsf{W}\;\Box\neg b \\
\varphi_6 &= \bigcirc\bigcirc\, b\,\mathsf{U}\,\Box\neg b
\end{aligned}$$

determine whether $TS \models_{fair} \varphi_i$. In case $TS \not\models_{fair} \varphi_i$, indicate a path $\pi \in \mathit{Paths}(TS)$ for which $\pi \not\models \varphi_i$.

EXERCISE 5.12. Let $\varphi = (a \rightarrow \bigcirc \neg b)\,\mathsf{W}\,(a \wedge b)$ and $P = \mathit{Words}(\varphi)$ where $AP = \{\, a, b \,\}$.

(a) Show that P is a safety property.

(b) Define an NFA $\mathcal{A}$ with $\mathcal{L}(\mathcal{A}) = \mathit{BadPref}(P)$.

(c) Now consider $P' = \mathit{Words}\big((a \rightarrow \bigcirc \neg b)\,\mathsf{U}\,(a \wedge b)\big)$. Decompose P' into a safety property P_{safe} and a liveness property P_{live} such that

$$P' = P_{safe} \cap P_{live}.$$

Show that P_{safe} is a safety and that P_{live} is a liveness property.

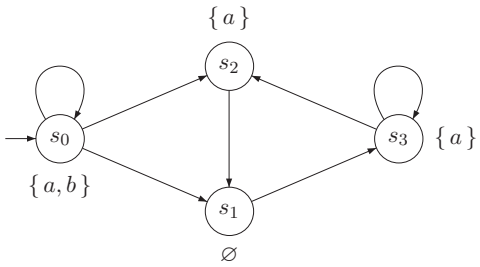


Figure 5.26: Transition system for Exercise 5.14.

EXERCISE 5.13. Provide an NBA for each of the following LTL formulae:

$$\Box(a \vee \neg \bigcirc b) \quad \text{and} \quad \Diamond a \;\vee\; \Box\Diamond(a \leftrightarrow b) \quad \text{and} \quad \bigcirc\bigcirc(a \vee \Diamond\Box b).$$

EXERCISE 5.14. Consider the transition system TS in Figure 5.26 with the atomic propositions $\{\, a, b \,\}$. Sketch the main steps of the LTL model-checking algorithm applied to TS and the LTL formulae

$$\varphi_1 \;=\; \Box\Diamond a \rightarrow \Box\Diamond b \quad \text{and} \quad \varphi_2 \;=\; \Diamond(a \wedge \bigcirc a).$$

To that end, carry out the following steps:

(a) Depict an NBA A_i for $\neg\varphi_i$.

(b) Depict the reachable fragment of the product transition system $TS \otimes \mathcal{A}_i$.

(c) Explain the main steps of the nested DFS in $TS \otimes \mathcal{A}_i$ by illustrating the order in which the states are visited during the "outer" and "inner" DFS.

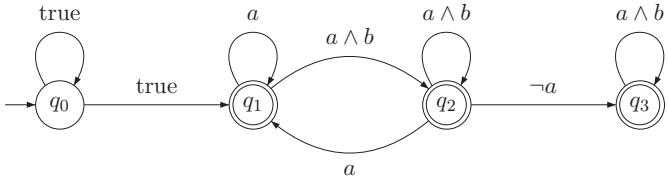


Figure 5.27: GNBA for Exercise 5.15.

(d) If $TS \not\models \varphi_i$, provide the counterexample resulting from the nested DFS.

Exercise 5.15. Consider the GNBA $\mathcal{G}$ in Figure 5.27 with the alphabet $\Sigma = 2^{\{a,b\}}$ and the set $\mathcal{F} = \{ \{q_1, q_3\}, \{q_2\} \}$ of accepting sets.

(a) Provide an LTL formula φ with $\mathit{Words}(\varphi) = \mathcal{L}_\omega(\mathcal{G})$. Justify your answer.

(b) Depict the NBA $\mathcal{A}$ with $\mathcal{L}_\omega(\mathcal{A}) = \mathcal{L}_\omega(\mathcal{G})$.

Exercise 5.16. Depict a GNBA $\mathcal{G}$ over the alphabet $\Sigma = 2^{\{a,b,c\}}$ such that

$$\mathcal{L}_\omega(\mathcal{G}) \;=\; \mathit{Words}\big(\; (\Box\Diamond a \rightarrow \Box\Diamond b) \;\wedge\; \neg a \;\wedge\; (\neg a \,\mathsf{W}\, c) \; \big).$$

Exercise 5.17. Let $\psi = \Box\, (a \leftrightarrow \bigcirc \neg a)$ and $AP = \{a\}$.

(a) Show that ψ can be transformed into the following equivalent basic LTL formula

$$\varphi = \neg \left[\mathrm{true}\, \mathsf{U}\, \left(\neg\, (a \wedge \bigcirc \neg a) \wedge \neg\, (\neg a \wedge \neg \bigcirc\, \neg a) \right) \right].$$

The basic LTL syntax is given by the following context-free grammar:

$$\varphi ::= \mathrm{true} \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc\, \varphi \mid \varphi_1\, \mathsf{U}\, \varphi_2.$$

(b) Compute all elementary sets with respect to $\mathit{closure}(\varphi)$ (*Hint: There are six elementary sets.*)

(c) Construct the GNBA $\mathcal{G}_\varphi$ with $\mathcal{L}_\omega(\mathcal{G}_\varphi) = \mathit{Words}(\varphi)$. To that end:

(i) Define its set of initial states and its acceptance component.

(ii) For each elementary set B, define $\delta(B, B \cap AP)$.

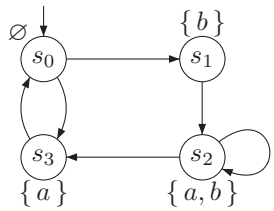
Exercise 5.18. Let $AP = \{a\}$ and $\varphi = (a \wedge \bigcirc\, a)\, \mathsf{U}\, \neg a$ an LTL formula over AP.

(a) Compute all elementary sets with respect to φ.
(*Hint: There are five elementary sets.*).

(b) Construct the GNBA $\mathcal{G}_\varphi$ such that $\mathcal{L}_\omega(\mathcal{G}_\varphi) = \mathit{Words}(\varphi)$.

Exercise 5.19. Consider the formula $\varphi = a \,\mathsf{U}\, (\neg a \wedge b)$ and let $\mathcal{G}$ be the GNBA for φ that is obtained through the construction explained in the proof of Theorem 4.56. What are the initial states in $\mathcal{G}$? What are the accept states in $\mathcal{G}$? Provide an accepting run for the word $\{a\}\{a\}\{a,b\}\{b\}^\omega$. Explain why there are no accepting runs for the words $\{a\}^\omega$ and $\{a\}\{a\}\{a,b\}^\omega$. (*Hint: The answers to these questions can be given without depicting* $\mathcal{G}$.)

Exercise 5.20.
We consider the LTL formula $\varphi = \Box\,(a \rightarrow (\neg b \,\mathsf{U}\, (a \wedge b)))$ over the set $AP = \{\, a, b \,\}$ of atomic propositions and we want to check $TS \models \varphi$ for TS outlined on the right.



(a) To check $TS \models \varphi$, convert $\neg\varphi$ into an equivalent LTL formula ψ which is constructed according to the following grammar:

$$\Phi ::= \text{true} \mid \text{false} \mid a \mid b \mid \Phi \wedge \Phi \mid \neg\Phi \mid \bigcirc\, \Phi \mid \Phi \,\mathsf{U}\, \Phi.$$

Then construct $\mathit{closure}(\psi)$.

(b) Give the elementary sets w.r.t. $\mathit{closure}(\psi)$!

(c) Construct the GNBA $\mathcal{G}_\psi$.

(d) Construct an NBA $\mathcal{A}_{\neg\varphi}$ *directly* from $\neg\varphi$, i.e., without relying on $\mathcal{G}_\psi$. (*Hint: Four states suffice.*)

(e) Construct $TS \otimes \mathcal{A}_{\neg\varphi}$.

(f) Use the nested DFS algorithm to check $TS \models \varphi$. Therefore, sketch the algorithm's main steps and interpret its outcome!

Exercise 5.21. The construction of a $\mathcal{G}$ from a given LTL formula in the proof of Theorem 4.56 assumes an LTL formula that only uses the basic temporal modalities $\bigcirc$ and U. The derived operators $\Diamond$, $\Box$, W and R can be treated by syntactic replacements of their definitions. Alternatively, and more efficient, is to treat them as basic modalities and to allow for formulae $\Diamond\psi$, $\Box\psi$, $\varphi_1 \,\mathsf{W}\, \varphi_2$ and $\varphi_1 \,\mathsf{R}\, \varphi_2$ as elements of elementary sets of formulae and redefine the components of the constructed GNBA.

Explain which modifications are necessary for such a "direct" treatment of $\Diamond$ (eventually), $\Box$ (always), W (weak-until), and R (release). That is, which additional conditions do the elementary sets, the transition function δ, and the set $\mathcal{F}$ of accepting sets have to fulfill?

EXERCISE 5.22. Let $TS = (S, Act, \rightarrow, S_0, AP, L)$ be a finite transition system without terminal states and let $wfair = \Diamond\Box b_1 \rightarrow \Box\Diamond b_2$ be a weak LTL fairness assumption with $b_1, b_2 \in AP$. Explain how the nested DFS can be modified to check directly whether $TS \models_{wfair} \Diamond\Box a$ (where $a \in AP$), that is, without using the transformation $TS \models_{wfair} \Diamond\Box a$ iff $TS \models (wfair \rightarrow \Diamond\Box a)$.

EXERCISE 5.23. Which of the following LTL formulae φ_i are representable by a deterministic Büchi automaton?

$$\varphi_1 \;=\; \Box(a \rightarrow \Diamond b), \quad \varphi_2 \;=\; \neg\varphi_1.$$

Explain your answer.

EXERCISE 5.24. Check for the following LTL formula whether they are (i) satisfiable, and/or (ii) valid:

(a) $\bigcirc \bigcirc a \;\Rightarrow\; \bigcirc a$

(b) $\bigcirc (a \;\vee\; \Diamond a) \;\Rightarrow\; \Diamond a$

(c) $\Box a \;\Rightarrow\; \neg \bigcirc (\neg a \;\wedge\; \Box \neg a)$

(d) $(\Box a) \,\mathsf{U}\, (\Diamond b) \;\Rightarrow\; \Box (a \,\mathsf{U}\, \Diamond b)$

(e) $\Diamond b \;\Rightarrow\; (a \,\mathsf{U}\, b)$

Practical Exercises

EXERCISE 5.25. Consider an arbitrary, but finite, number of identical processes[2], that execute in parallel. Each process consists of a noncritical part and a critical part, usually called the *critical section*. In this exercise we are concerned with the verification of a mutual exclusion protocol, that is, a protocol that should ensure that at any moment of time at most one process (among the N processes in our configuration) is in its critical section. There are many different mutual exclusion protocols developed in the literature. In this exercise we are concerned with Szymanski's protocol [384]. Assume there are N processes for some fixed $N > 0$. There is a global variable, referred to as *flag*, which is an array of length N, such that *flag[i]* is a value between 0 and 4 (for $0 \leqslant i < N$). The idea is that *flag[i]* indicates the status of process i. The protocol executed by process i looks as follows:

l0: **loop forever do**
    begin
    l1: Noncritical section
    l2: *flag[i]* := 1;

[2]Only the identity of a process is unique.

l3: **wait until** (*flag[0]* < 3 and *flag[1]* < 3 and ... and *flag[N-1]* < 3)
l4: *flag[i]* := 3;
l5: **if** (*flag[0]* $= 1$ or *flag[1]* $= 1$ or ... or *flag[N-1]* $= 1$)
then begin
l6: *flag[i]* := 2;
l7: **wait until** (*flag[0]* $= 4$ or *flag[1]* $= 4$ or ... or *flag[N-1]* $= 4$)
end
l8: *flag[i]* := 4;
l9: **wait until** (*flag[0]* < 2 and *flag[1]* < 2 and ... and *flag[i-1]* < 2)
l10: Critical section
l11: **wait until** (*flag[i+1]* $\in \{0, 1, 4\}$) and ... and (*flag[N-1]* $\in \{0, 1, 4\}$)
l12: *flag[i]* := 0;
end.

Before doing any of the exercises listed below, try first to informally understand what the protocol is doing and why it could be correct in the sense that mutual exclusion is ensured. If you are convinced of the fact that the correctness of this protocol is not easy to see — otherwise please inform me — then start with the following questions.

1. Model Szymanski's protocol in Promela. Assume that all tests on the global variable *flag* (such as the one in statement l3) are *atomic*. Look carefully at the indices of the variable *flag* used in the tests. Make the protocol description modular such that the number of processes can be changed easily.
2. Check for several values of N ($N \geqslant 2$) that the protocol indeed ensures mutual exclusion. Report your results for N equal to 4.
3. The code that a process has to go through before reaching the critical section can be divided into several segments. We refer to statement l4 as the *doorway*, to segments l5, l6, and l7, as the *waiting room* and to segments l8 through l12 (which contains the critical section) as the *inner sanctum*. You are requested to check the following basic claims using assertions. Give for each case the changes to your original Promela specification for Szymanski's protocol and present the verification results. In case of negative results, simulate the counterexample by means of guided simulation.
   (a) Whenever some process is in the inner sanctum, the doorway is locked, that is, no process is at location l4.
   (b) If a process i is at l10, l11 or l12, then it has the least index of all the processes in the waiting room and the inner sanctum.
   (c) If some process is at l12, then all processes in the waiting room and in the inner sanctum must have flag value 4.

Exercise 5.26. We assume N processes in a ring topology, connected by unbounded queues. A process can only send messages in a clockwise manner. Initially, each process has a unique identifier *ident* (which is assumed to be a natural number). A process can be either active or

relaying. Initially a process is active. In Peterson's leader election algorithm (1982) each process in the ring carries out the following task:

```
active:
d := ident;
do forever
begin
   /* start phase */
  send(d);
  receive(e);
  if e = ident then announce elected;
  if d > e then send(d) else send(e);
  receive(f);
  if f = ident then announce elected;
  if e ⩾ max(d, f) then d := e else goto relay;
end
relay:
do forever
begin
  receive(d);
  if d = ident then announce elected;
  send(d)
end
```

Solve the following questions concerning the leader election protocol:

1. Model Peterson's leader election protocol in Promela (avoid invalid end states).
2. Verify the following properties:
   (a) There is always at most one leader.
   (b) Eventually always a leader will be elected.
   (c) The elected leader will be the process with the highest number.
   (d) The maximum total amount of messages sent in order to elect a leader is at most $2N\lfloor \log_2 N \rfloor + N$.

Exercise 5.27. This exercise deals with a simple fault-tolerant communication protocol in which processes can fail. A failed process is still able to communicate, i.e., it is able to send and receive messages, but the content of its transmitted messages is unreliable. More precisely, a failed process can send messages with arbitrary content. A failed process is therefore also called *unreliable*.

We are given N reliable processes (i.e., processes that have not failed and that are working as expected) and K unreliable processes, where N is larger than $3 \cdot K$ and K is at least 0. There is no way, a priori, to distinguish the reliable and the unreliable processes. All processes communicate

by means of exchanging messages. Each process has a local variable, which initially has a value, 0 or 1. The following informally described protocol is aimed to be followed by the reliable processes, such that at the end of round $K+1$ we have

- eventually every reliable process has the same value in its local variable, and
- if all reliable processes have the same initial value, then their final value is the same as their common initial value.

The difficulty of this protocol is to establish these constraints in the presence of the K unreliable processes!

Informal description of the protocol The following protocol is due to Berman and Garay [47]. Let the processes be numbered 1 through $N+K$. Processes communicate with each other in "rounds". Each round consists of two phases of message transmissions: In round i, $i > 0$, in the first phase, every process sends its value to all processes (including itself); in the second phase, process i sends the majority value it received in the first phase (for majority to be well-defined we assume that $N+K$ is odd) to all processes. If a process receives N, or more, instances of the same value in its first phase of the round, it sets its local variable to this value; otherwise, it sets its local variable to the value received (from process i) in the second phase of this round.

1. Model this protocol in Promela. Make the protocol description modular such that the number of reliable and unreliable processes can be changed easily. As the state space of your protocol model could be very large, instantiate your model with a single unreliable process and four reliable processes.

   First hint: One of the main causes for the large state space is the model for the unreliable process, so try to keep this model as simple as possible. This can be achieved by, for instance, assuming that an unreliable process can only transmit arbitrary 0 or 1 values (and not any other value) and that a process always starts with a fixed initial value (and not with a randomly selected one). In addition, use atomic broadcast for message transmissions.

   Second hint: It might be convenient (though not necessary) to use a matrix of size $(N+K) \cdot (N+K)$ of channels for the communication structure. As Promela does not support multi-dimensional arrays, you could use the following construct (where M equals $N+K$):

   ```
   typedef Arraychan {
   chan ch[M] = [1] of {bit};    /* M channels of size 1 */
   }

   Arraychan A[M];           /* a matrix A of MxM channels of size 1 */
   ```

   Statement `A[i].ch[j]!0` denotes an output of value `0` over the channel directed process from `i` to `j`. Similarly, statement `A[i].ch[j]?b` denotes the receipt of a bit value stored in variable `b` via the channel directed from process `i` to `j`.

2. Formalize the two constraints of the protocol in LTL and convert these into never claims.

3. Check the two temporal logic properties by SPIN and hand in the verification output generated by SPIN.
4. Show that the requirement $N > 3 \cdot K$ is essential, by, for instance, changing the configuration of your system such that $N \leqslant 3 \cdot K$ and checking that for this configuration the first aforementioned constraint is violated. Create the shortest counterexample (select the shortest trail in the advanced verification options) and perform a guided simulation of this undesired scenario. Hand in the counterexample you found and give an explanation of it.

Chapter 6

Computation Tree Logic

This chapter introduces Computation Tree Logic (CTL), a prominent branching temporal logic for specifying system properties. In particular, the syntax and semantics of CTL are presented, a comparison to Linear Temporal Logic (LTL) is provided, and the issue of fairness in CTL is treated. CTL model checking is explained in detail. First, the core recursive algorithm is presented that is based on a bottom-up traversal of the parse tree of the formula at hand. The foundations of this algorithm are discussed and the adaptations needed to incorporate fairness are detailed. This is followed by an algorithm for the generation of counterexamples. The chapter is concluded by presenting a model-checking algorithm for CTL*, a branching-time logic that subsumes both LTL and CTL.

6.1 Introduction

Pnueli [337] has introduced linear temporal logic for the specification and verification of reactive systems. LTL is called linear, because the qualitative notion of time is path-based and viewed to be linear: at each moment of time there is only one possible successor state and thus each time moment has a unique possible future. Technically speaking, this follows from the fact that the interpretation of LTL formulae is defined in terms of paths, i.e., sequences of states.

Paths themselves, though, are obtained from a transition system that might be branching: a state may have several, distinct direct successor states, and thus several computations may start in a state. The interpretation of LTL-formulae in a state requires that a formula φ holds in state s if *all* possible computations that start in s satisfy φ. The universal

quantification over all computations that is implicit in the LTL semantics can also be made explicit in the formula, e.g.:

$s \models \forall\varphi$ if and only if $\pi \models \varphi$ for all paths π starting in s

In LTL, we thus can state properties over all possible computations that start in a state, but not easily about *some* of such computations. To some extent this may be overcome by exploiting the duality between universal and existential quantification. For instance, to check whether there exists some computation starting in s that satisfies φ we may check whether $s \models \forall \neg \varphi$; if this formula is not satisfied by s, then there must be a computation that meets φ, otherwise they should all refute φ.

For more complicated properties, like "for every computation it is always possible to return to the initial state", this is, however, not possible. A naive attempt would be to require $\Box \Diamond \mathit{start}$ for every computation, i.e., $s \models \forall\Box \Diamond \mathit{start}$, where the proposition *start* uniquely identifies the initial state. This is, however, too strong as it requires a computation to *always* return to the initial state, not just *possibly*. Other attempts to specify the intended property also fail, and it even turns out to be the case that the property *cannot* be specified in LTL.

To overcome these problems, in the early eighties another strand of temporal logics for specification and verification purposes was introduced by Clarke and Emerson [86]. The semantics of this kind of temporal logic is not based on a linear notion of time—an infinite sequence of states—but on a *branching* notion of time—an infinite *tree* of states. Branching time refers to the fact that at each moment there may be several different possible futures. Each moment of time may thus split into several possible futures. Due to this branching notion of time, this class of temporal logic is known as *branching* temporal logic. The semantics of a branching temporal logic is defined in terms of an infinite, directed *tree* of states rather than an infinite sequence. Each traversal of the tree starting in its root represents a single path. The tree itself thus represents all possible paths, and is directly obtained from a transition system by "unfolding" at the state of interest. The tree rooted at state s thus represents all possible infinite computations in the transition system that start in s. Figure 6.1 depicts a transition system and its unfolding. (For convenience, each node in the tree consists of a pair indicating the state and the level of the node in the tree.)

The temporal operators in branching temporal logic allow the expression of properties of *some* or *all* computations that start in a state. To that end, it supports an existential path quantifier (denoted $\exists$) and a universal path quantifier (denoted $\forall$). For instance, the property $\exists\Diamond \Phi$ denotes that there exists a computation along which $\Diamond \Phi$ holds. That is, it states that there is at least one possible computation in which a state that satisfies Φ

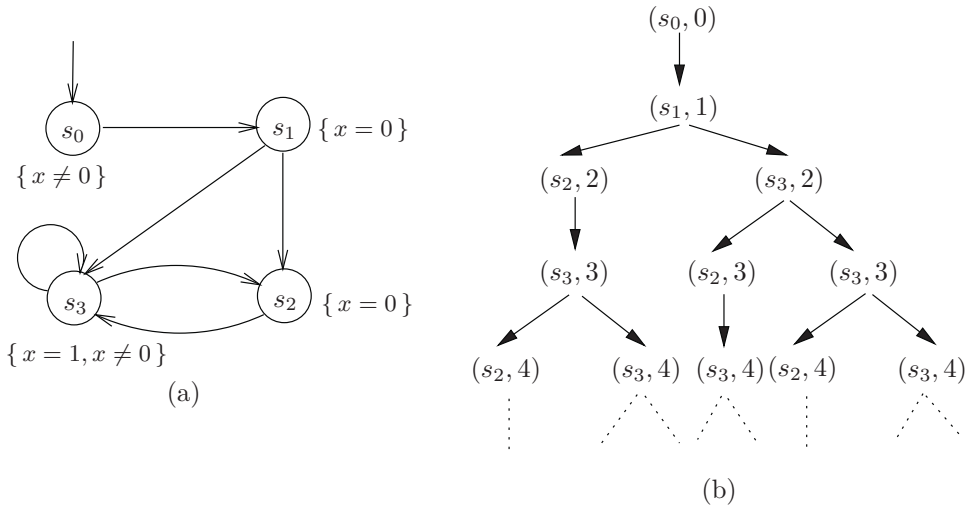


Figure 6.1: (a) A transition system and (b) a prefix of its infinite computation tree

is eventually reached. This does not, however, exclude the fact that there can also be computations for which this property does not hold, for instance, computations for which Φ is always refuted. The property $\forall\Diamond\,\Phi$, in contrast, states that all computations satisfy the property $\Diamond\,\Phi$. More complicated properties can be expressed by nesting universal and existential path quantifiers. For instance, the aforementioned property "for every computation it is always possible to return to the initial state" can be faithfully expressed by $\forall\Box\,\exists\Diamond\,\mathit{start}$: in any state ($\Box$) of any possible computation ($\forall$), there is a possibility ($\exists$) to eventually return to the start state ($\Diamond\,\mathit{start}$).

This chapter considers *Computation Tree Logic (CTL)*, a temporal logic based on propositional logic with a discrete notion of time, and only future modalities. CTL is an important branching temporal logic that is sufficiently expressive for the formulation of an important set of system properties. It was originally used by Clarke and Emerson [86] and (in a slightly different form) by Queille and Sifakis [347] for model checking. More importantly, it is a logic for which efficient and—as we will see—rather simple model-checking algorithms do exist.

Anticipatory to the results presented in this chapter, we summarize the major aspects of the linear-vs-branching-time debate and provide arguments that justify the treatment of model checking based on linear or branching time logics:

- The expressiveness of many linear and branching temporal logics is incomparable. This means that some properties that are expressible in a linear temporal logic cannot be expressed in certain branching temporal logics, and vice versa.

Aspect	*Linear time*	*Branching time*
"behavior" in a state s	path-based: $trace(s)$	state-based: computation tree of s
temporal logic	LTL: path formulae φ $s \models \varphi$ iff $\forall \pi \in Paths(s).\, \pi \models \varphi$	CTL: state formulae existential path quantification $\exists\varphi$ universal path quantification: $\forall\varphi$
complexity of the model checking problems	PSPACE–complete $\mathcal{O}(\lvert TS\rvert \cdot \exp(\lvert\varphi\rvert))$	*PTIME* $\mathcal{O}(\lvert TS\rvert \cdot \lvert\Phi\rvert)$
implementation-relation	trace inclusion and the like (proof is PSPACE-complete)	simulation and bisimulation (proof in polynomial time)
fairness	no special techniques needed	special techniques needed

Table 6.1: Linear-time vs. branching-time in a nutshell.

- The model-checking algorithms for linear and branching temporal logics are quite different. This results, for instance, in significantly different time and space complexity results.
- The notion of fairness can be treated in linear temporal logic without the need for any additional machinery since fairness assumptions can be expressed in the logic. For various branching temporal logics this is not the case.
- The equivalences and preorders between transition systems that "correspond" to linear temporal logic are based on traces, i.e., trace inclusion and equality, whereas for branching temporal logic such relations are based on simulation and bisimulation relations (see Chapter 7).

Table 6.1 summarizes the main differences between the linear-time and branching-time perspective in a succinct way.

6.2 Computation Tree Logic

This section presents the syntax and the semantics of CTL. The following sections will discuss the relation and differences between CTL and LTL, present a model-checking algorithm for CTL, and introduce some extensions of CTL.

6.2.1 Syntax

CTL has a two-stage syntax where formulae in CTL are classified into state and path formulae. The former are assertions about the atomic propositions in the states and their branching structure, while path formulae express temporal properties of paths. Compared to LTL formulae, path formulae in CTL are simpler: as in LTL they are built by the next-step and until operators, but they must not be combined with Boolean connectives and no nesting of temporal modalities is allowed.

Definition 6.1. Syntax of CTL

CTL *state formulae* over the set AP of atomic proposition are formed according to the following grammar:

$$\Phi ::= \text{true} \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi \mid \forall\varphi$$

where $a \in AP$ and φ is a path formula. CTL *path formulae* are formed according to the following grammar:

$$\varphi ::= \bigcirc \Phi \mid \Phi_1 \,\mathsf{U}\, \Phi_2$$

where Φ, Φ_1 and Φ_2 are state formulae. ■

Greek capital letters will denote CTL state formulae (CTL formulae, for short), whereas lowercase Greek letters will denote CTL path formulae.

CTL distinguishes between state formulae and path formulae. Intuitively, state formulae express a property of a state, while path formulae express a property of a path, i.e., an infinite sequence of states. The temporal operators $\bigcirc$ and U have the same meaning as in LTL and are path operators. Formula $\bigcirc\Phi$ holds for a path if Φ holds at the next state in the path, and $\Phi \,\mathsf{U}\, \Psi$ holds for a path if there is some state along the path for which Ψ holds, and Φ holds in all states prior to that state. Path formulae can be turned into state formulae by prefixing them with either the path quantifier $\exists$ (pronounced "for some path") or the path quantifier $\forall$ (pronounced "for all paths"). Note that the linear

temporal operators $\bigcirc$ and U are required to be immediately preceded by $\exists$ or $\forall$ to obtain a legal state formula. Formula $\exists\varphi$ holds in a state if there exists *some* path satisfying φ that starts in that state. Dually, $\forall\varphi$ holds in a state if *all* paths that start in that state satisfy φ.

Example 6.2. Legal CTL Formulae

Let $AP = \{\, x = 1, x < 2, x \geqslant 3 \,\}$ be a set of atomic propositions. Examples of syntactically correct CTL formulae are

$$\exists\bigcirc\,(x = 1), \forall\bigcirc\,(x = 1), \text{ and } x < 2 \ \vee \ x = 1$$

and $\exists((x < 2)\,\mathsf{U}\,(x \geqslant 3))$ and $\forall(\text{true}\,\mathsf{U}\,(x < 2))$. Some examples of formulae that are syntactically incorrect are

$$\exists(x = 1 \ \wedge \ \forall\bigcirc\,(x \geqslant 3)) \text{ and } \exists\bigcirc\,(\text{true}\,\mathsf{U}\,(x = 1)).$$

The first is not a CTL formula since $x = 1 \ \wedge \ \forall\bigcirc\,(x \geqslant 3)$ is not a path formula and thus must not be preceded by $\exists$ The second formula is not a CTL formula since $\text{true}\,\mathsf{U}\,(x = 1)$ is a path formula rather than a state formula, and thus cannot be preceded by $\bigcirc$. Note that

$$\exists\bigcirc\,(x = 1 \ \wedge \ \forall\bigcirc\,(x \geqslant 3)) \text{ and } \exists\bigcirc\,\forall(\text{true}\,\mathsf{U}\,(x = 1))$$

are, however, syntactically correct CTL formulae. ■

The Boolean operators true, false, $\wedge$, $\rightarrow$ and $\Leftrightarrow$ are defined in the usual way. The temporal modalities "eventually", "always", and "weak until" can be derived—similarly as for LTL—as follows:

$$\begin{array}{lrcl}
\text{eventually:} & \exists\Diamond\Phi & = & \exists(\text{true}\,\mathsf{U}\,\Phi) \\
& \forall\Diamond\Phi & = & \forall(\text{true}\,\mathsf{U}\,\Phi) \\
\text{always:} & \exists\Box\Phi & = & \neg\forall\Diamond\neg\Phi \\
& \forall\Box\Phi & = & \neg\exists\Diamond\neg\Phi
\end{array}$$

$\exists\Diamond\,\Phi$ is pronounced "Φ holds potentially" and $\forall\Diamond\,\Phi$ is pronounced "Φ is inevitable". $\exists\Box\,\Phi$ is pronounced "potentially always Φ", $\forall\Box\,\Phi$ is pronounced "invariantly Φ", and $\forall\bigcirc\,\Phi$ is pronounced "for all paths next Φ".

Note that "always" Φ cannot be obtained from the "equation" $\Box\Phi = \neg\Diamond\neg\Phi$ (as in LTL), since propositional logic operators cannot be applied to path formulae. In particular, $\exists\neg\Diamond\neg\Phi$ is *not* a CTL formula. Instead, we exploit the duality of existential and universal quantification:

- there exists a path with the property E if and only if the state property "not all paths violate property E" is satisfied, and
- all paths satisfy property E if and only if the state property "there is a path without property E" is violated.

Accordingly, $\exists\Box\,\Phi$ is not defined as $\exists\neg\Diamond\,\neg\Phi$, but rather as $\neg\forall\Diamond\,\neg\Phi$.

Example 6.3. CTL Formulae

To give a feeling about how simple properties can be formalized in CTL we treat some intuitive examples. The mutual exclusion property can be described in CTL by the formula

$$\forall\Box(\neg crit_1 \vee \neg crit_2).$$

CTL formulae of the form $\forall\Box\forall\Diamond\Phi$ express that Φ is infinitely often true on all paths. (This fact will be formally proven later; see Remark 6.8 on page 326.) The CTL formula

$$(\forall\Box\forall\Diamond crit_1) \;\wedge\; (\forall\Box\forall\Diamond crit_2)$$

thus requires each process to have access to the critical section infinitely often. In case of a traffic light, the safety property "each red light phase is preceded by a yellow light phase" can be formulated in CTL by

$$\forall\Box(yellow \;\vee\; \forall\bigcirc\neg red)$$

depending on the precise meaning of a "phase". The liveness property "the traffic light is infinitely often green" can be formulated as

$$\forall\Box\forall\Diamond green \quad .$$

Progress properties such as "every request will eventually be granted" can be described by

$$\forall\Box(request \;\longrightarrow\; \forall\Diamond response).$$

Finally, the CTL formula

$$\forall\Box\;\exists\Diamond\; start$$

expresses that in every reachable system state it is possible to return (via 0 or more transitions) to (one of) the starting state(s). ■

6.2.2 Semantics

CTL formulae are interpreted over the states and paths of a transition system TS. Formally, given a transition system TS, the semantics of CTL formulae is defined by two satisfaction relations (both denoted by $\models_{TS}$, or briefly $\models$): one for the state formulae and one for the path formulae. For the state formulae, $\models$ is a relation between the states in TS and state formulae. We write $s \models \Phi$ rather than $(s, \Phi) \in \models$. The intended interpretation is: $s \models \Phi$ if and only if state formula Φ holds in state s. For the path formulae, $\models$ is a relation between maximal path fragments in TS and path formulae. We write $\pi \models \Phi$ rather than $(\pi, \Phi) \in \models$. The intended interpretation is: $\pi \models \varphi$ if and only if path π satisfies path formula φ.

Definition 6.4. Satisfaction Relation for CTL

Let $a \in AP$ be an atomic proposition, $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system without terminal states, state $s \in S$, Φ, Ψ be CTL state formulae, and φ be a CTL path formula. The satisfaction relation $\models$ is defined for state formulae by

$$
\begin{array}{lll}
s \models a & \text{iff} & a \in L(s) \\
s \models \neg \Phi & \text{iff} & \text{not } s \models \Phi \\
s \models \Phi \wedge \Psi & \text{iff} & (s \models \Phi) \text{ and } (s \models \Psi) \\
s \models \exists \varphi & \text{iff} & \pi \models \varphi \text{ for some } \pi \in \mathit{Paths}(s) \\
s \models \forall \varphi & \text{iff} & \pi \models \varphi \text{ for all } \pi \in \mathit{Paths}(s)
\end{array}
$$

For path π, the satisfaction relation $\models$ for path formulae is defined by

$$
\begin{array}{lll}
\pi \models \bigcirc \Phi & \text{iff} & \pi[1] \models \Phi \\
\pi \models \Phi \,\mathsf{U}\, \Psi & \text{iff} & \exists\, j \geqslant 0.\ (\pi[j] \models \Psi \ \wedge\ (\forall\, 0 \leqslant k < j.\, \pi[k] \models \Phi)) \quad .
\end{array}
$$

where for path $\pi = s_0\, s_1\, s_2 \ldots$ and integer $i \geqslant 0$, $\pi[i]$ denotes the $(i{+}1)$th state of π, i.e., $\pi[i] = s_i$. ■

The interpretations for atomic propositions, negation, and conjunction are as usual, where it should be noted that in CTL they are interpreted over states, whereas in LTL they are interpreted over paths. state formula $\exists\varphi$ is valid in state s if and only if there exists some path starting in s that satisfies φ. In contrast, $\forall\varphi$ is valid in state s if and only if all paths starting in s satisfy φ. The semantics of the path formulae is identical (although formulated slightly more simply) to that for LTL.[1] For instance, $\exists \bigcirc \Phi$ is valid in state s if

[1] The semantics of the CTL path formulae is formulated more simply than for LTL, since in CTL each

and only if there exists some path π starting in s such that in the next state of this path, state $\pi[1]$, the property Φ holds. This is equivalent to the existence of a direct successor s' of s such that $s' \models \Phi$. $\forall(\Phi \,\mathsf{U}\, \Psi)$ is valid in state s if and only if every path starting in s has an initial finite prefix (possibly only containing s) such that Ψ holds in the last state of this prefix and Φ holds in all other states along the prefix. $\exists(\Phi \,\mathsf{U}\, \Psi)$ is valid in s if and only if there exists a path starting in s that satisfies $\Phi \,\mathsf{U}\, \Psi$. As for LTL, the semantics of CTL here is nonstrict in the sense that the path formula $\Phi \,\mathsf{U}\, \Psi$ is valid if the initial state of the path satisfies Ψ.

Definition 6.5. CTL Semantics for Transition Systems

Given a transition system *TS* as before, the *satisfaction set* $Sat_{TS}(\Phi)$, or briefly $Sat(\Phi)$, for CTL-state formula Φ is defined by:

$$Sat(\Phi) \;=\; \{\, s \in S \mid s \models \Phi \,\}.$$

The transition system *TS* satisfies CTL formula Φ if and only if Φ holds in all initial states of *TS*:

$$TS \models \Phi \quad \text{if and only if} \quad \forall s_0 \in I.\, s_0 \models \Phi \quad .$$

This is equivalent to $I \subseteq Sat(\Phi)$. ■

The semantics of the derived path operators "always" and "eventually" is similar to that in LTL. For path fragment $\pi = s_0\, s_1\, s_2 \ldots$:

$$\pi \models \Diamond\Phi \quad \text{if and only if} \quad s_j \models \Phi \text{ for some } j \geqslant 0.$$

From this it can be derived that:

$$s \models \exists\Box\Phi \quad \text{iff} \quad \exists\pi \in Paths(s).\, \pi[j] \models \Phi \text{ for all } j \geqslant 0,$$
$$s \models \forall\Box\Phi \quad \text{iff} \quad \forall\pi \in Paths(s).\, \pi[j] \models \Phi \text{ for all } j \geqslant 0.$$

Therefore, $\Box\Phi$ can be understood as CTL path formula with the semantics:

$$\pi = s_0\, s_1\, s_2 \ldots \models \Box\Phi \quad \text{if and only if} \quad s_j \models \Phi \text{ for all } j \geqslant 0.$$

In particular, $\forall\Box\Phi$ corresponds to the invariant over the invariant condition Φ.

In a similar way, one can derive that $\exists\Box\,\Phi$ is valid in state s if and only if there exists some path starting at s such that for each state on this path the formula Φ holds. The formula $\exists\Diamond\,\Phi$ is valid in state s if and only if Φ holds eventually along some path that starts in s, and $\forall\Diamond\,\Phi$ is valid if and only if this property holds for all paths that start in s. A schematic overview of the validity of $\exists\Box$, $\exists\Diamond$, $\forall\Diamond$, and $\forall\Box$ is given in Figure 6.2, where

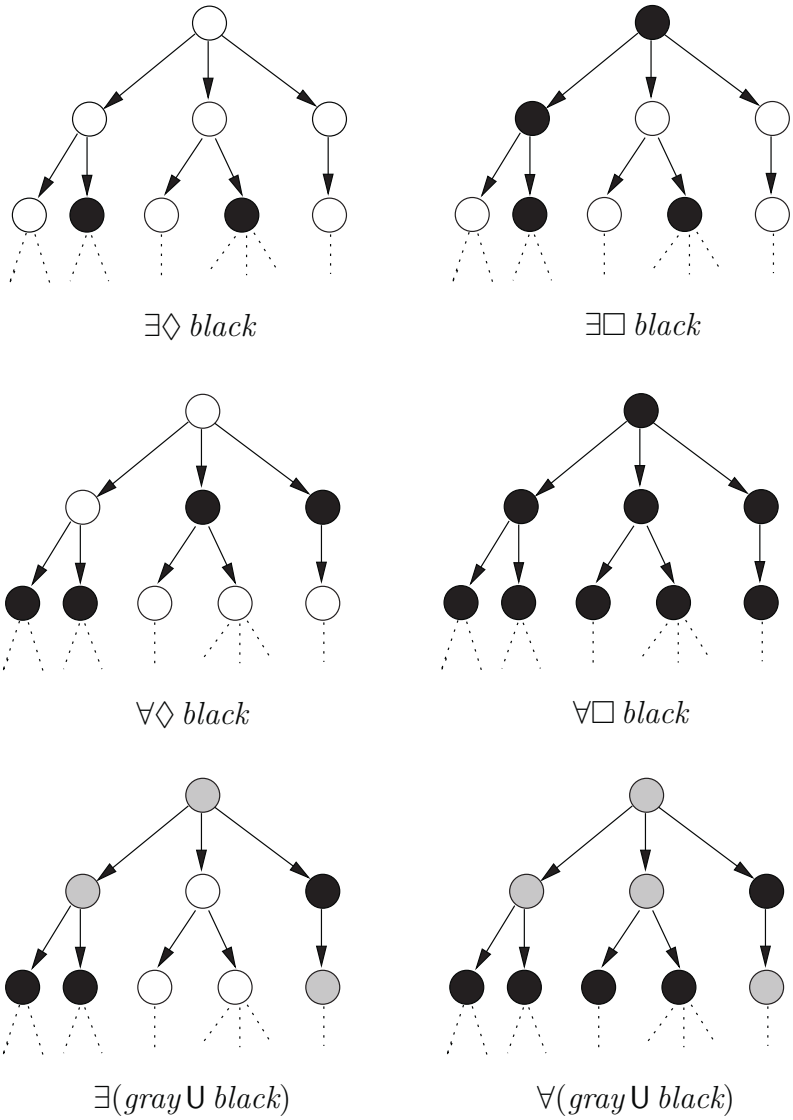


Figure 6.2: Visualization of semantics of some basic CTL formulae.

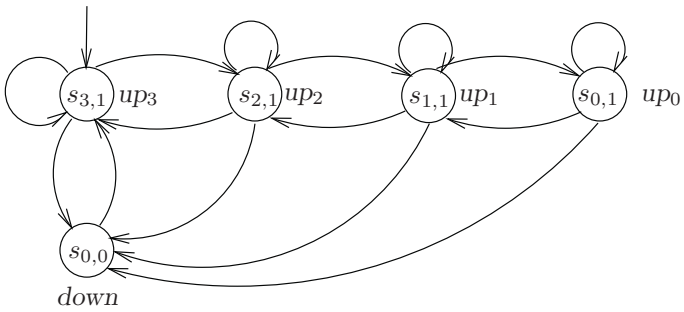


Figure 6.3: A transition system of the TMR system.

Property	*Formalization in CTL*
Possibly the system never goes down	$\exists\Box\, \neg\, down$
Invariantly the system never goes down	$\forall\Box\, \neg\, down$
It is always possible to start as new	$\forall\Box\, \exists\Diamond\, up_3$
The system always eventually goes down and is operational until going down	$\forall\, ((up_3\ \vee\ up_2)\, \mathsf{U}\, down)$

Table 6.2: Some properties for the TMR system and their formalization in CTL.

black-colored states satisfy the proposition *black*, gray states are labeled with *gray*, and all other states are labeled neither with *black* nor with *gray*.

Example 6.6. A Triple Modular Redundant System

Consider a triple modular redundant (TMR) system with three processors and a single voter. As each component of this system can fail, the reliability is increased by letting all processors execute the same program. The voter takes a majority vote of the outputs of the three processors. If a single processor fails, the system can still produce reliable outputs. Each component can be repaired. It is assumed that only one component at a time can fail and only one at a time can be repaired. On failure of the voter, the entire system fails. On repair of the voter, it is assumed that the system starts as being new, i.e., with three processors and a voter. The transition system of this TMR is depicted in Figure 6.3. States are of the form $s_{i,j}$ where i denotes the number of processors that is currently up ($0 < i \leqslant 3$) and j the number of operational voters ($j = 0, 1$). We consider the TMR system to be operational if at least two processors are functioning properly. Some interesting properties of this system and their formulation in CTL are listed in Table 6.2 on page 323. We consider each of the formulae in isolation:

temporal operator has to be immediately followed by a state formula.

- state formula $\exists\Box\,\neg\,down$ holds in state $s_{3,1}$, as there is a path, e.g., $(s_{3,1}\,s_{2,1})^\omega$, starting in that state and that never reaches the *down* state, i.e., $(s_{3,1}\,s_{2,1})^\omega \models \Box\,\neg\,down$.
- Formula $\forall\Box\,\neg\,down$, however, does not hold in state $s_{3,1}$, as there is a path starting from that state, such as $(s_{3,1})^+\,s_{0,0}\ldots$, that satisfies $\neg\,\Box\,\neg\,down$, or, equivalently, $\Diamond\,down$.
- Formula $\forall\Box\,\exists\Diamond\,up_3$ holds in state $s_{3,1}$, as in any state of any of its paths it is possible to return to the initial state, e.g., by first moving to state $s_{0,0}$ and then to $s_{3,1}$. For instance, the path $s_{3,1}\,(s_{2,1})^\omega \models \Box\,\exists\Diamond\,up_3$ since $s_{2,1} \models \exists\Diamond\,up_{3,1}$. This property should not be confused with the CTL formula $\forall\Diamond\,up_3$, which expresses that each path eventually will visit the initial state. (Note that this formula is trivially valid for state $s_{3,1}$ as it satisfies up_3.)
- The last property of Table 6.2 does not hold in state $s_{3,1}$ as there exists a path, such as $s_{3,1}\,s_{2,1}\,s_{1,1}\,s_{0,0}\ldots$, for which the path formula $(up_3 \vee up_2)\;\mathsf{U}\,down$ does not hold. The formula is refuted since the path visits state $s_{1,1}$, a state that satisfies neither *down* nor up_3 nor up_2.

■

Example 6.7. CTL Semantics

Consider the transition system depicted at the top (a) of Figure 6.4. Just below the transition system the validity of several CTL formulae is indicated for each state. (For simplicity, the initial states are not indicated.) A state is colored black if the formula is valid in that state; otherwise, it is white. The following formulae are considered:

- The formula $\exists\bigcirc a$ is valid for all states since all states have some direct successor state that satisfies a.
- $\forall\bigcirc a$ is not valid for state s_0, since a possible path starting at s_0 goes directly to state s_2 for which a does not hold. Since the other states have only direct successors for which a holds, $\forall\bigcirc a$ is valid for all other states.
- For all states except state s_2, it is possible to have a computation that leads to state s_3 (such as $s_0\,s_1\,s_3^\omega$ when starting in s_0) for which a is globally valid. Therefore, $\exists\Box\,a$ is valid in these states. Since $a \notin L(s_2)$ there is no path starting at s_2 for which a is globally valid.
- $\forall\Box\,a$ is only valid for s_3 since its only path, s_3^ω, always visits a state in which a holds. For all other states it is possible to have a path which contains s_2 that does not satisfy a. So for these states $\forall\Box\,a$ is not valid.

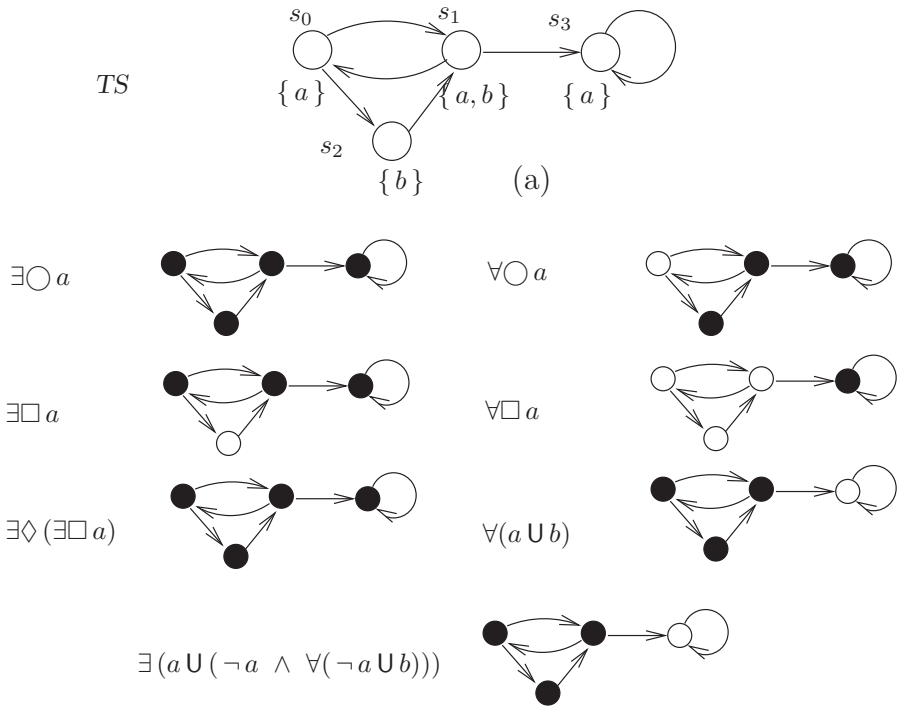


Figure 6.4: Interpretation of several CTL formulae.

- $\exists\Diamond\,(\exists\Box\,a)$ is valid for all states since from each state another state (either s_0, s_1, or s_3) can be eventually reached from which some computation can start along which a is globally valid.
- $\forall(a\,\mathsf{U}\,b)$ is not valid in s_3 since its only computation (s_3^ω) never reaches a state for which b holds. In state s_0 proposition a holds until b holds, and in states s_1 and s_2 proposition b holds immediately. So, for these states the formula is true.
- Finally, $\exists(a\,\mathsf{U}\,(\neg\,a\ \wedge\ \forall(\neg\,a\,\mathsf{U}\,b)))$ is not valid in s_3, since from s_3 a b-state can never be reached. For the states s_0 and s_1 the formula is valid, since state s_2 can be reached from these states via an a-path; $\neg\,a$ is valid in s_2, and from s_2 all possible paths satisfy $\neg\,a\,\mathsf{U}\,b$, since s_2 is a b-state. For instance, for state s_0 the path $(s_0\,s_2\,s_1)^\omega$ satisfies $a\,\mathsf{U}\,(\neg\,a\ \wedge\ \forall(\neg\,a\,\mathsf{U}\,b))$ since $a \in L(s_0)$, $a \notin L(s_2)$, and $b \in L(s_1)$. For state s_2 the property is valid since a is invalid in s_2 and for all paths starting at s_2 the first state is a b-state.

■

Remark 6.8. Infinitely Often

For a better comprehension of the semantics of CTL, let us prove that:

$$s \models \forall\Box\forall\Diamond a \quad \text{if and only if} \quad \forall\pi \in \mathit{Paths}(s).\,\pi[i] \models a \text{ for infinitely many } i\,.$$

$\Rightarrow$: Let s be a state, such that $s \models \forall\Box\,\forall\Diamond\,a$. The proof obligation is to show that every infinite path fragment π starting in s passes through an a-state infinitely often. Let $\pi = s_0\,s_1\,s_2\ldots \in \mathit{Paths}(s)$ and $j \geqslant 0$. We demonstrate that there exists an index $i \geqslant j$ with $s_i \models a$. Since $s \models \forall\Box\forall\Diamond a$, we have

$$\pi \models \Box\forall\Diamond a.$$

In particular, $s_j \models \forall\Diamond a$. From $\pi[j..] = s_j\,s_{j+1}\ldots \in \mathit{Paths}(s_j)$ it follows that

$$s_j\,s_{j+1}\,s_{j+2}\ldots \models \Diamond a.$$

Thus, there exists an index $i \geqslant j$ with $s_i \models a$. As this reasoning applies to any index j, path π visits an a-state infinitely often.

$\Leftarrow$: Let s be a state such that every infinite path fragment starting in s visits infinitely many a-states. Let $\pi = s_0\,s_1\,s_2\ldots \in \mathit{Paths}(s)$. To show that $s \models \forall\Box\forall\Diamond a$, it has to be proven that $\pi \models \Box\forall\Diamond a$, i.e.:

$$s_j \models \forall\Diamond a, \qquad \text{for any } j \geqslant 0.$$

Let $j \geqslant 0$ and $\pi' = s'_j\, s'_{j+1}\, s'_{j+2} \ldots \in \mathit{Paths}(s_j)$. To show that $s_j \models \forall\Diamond a$, it suffices to prove that π' visits at least one a-state. It is not difficult to infer that

$$\pi'' \;=\; \underbrace{s_0\, s_1\, s_2 \ldots\, s_j}_{\text{prefix of } \pi}\; \underbrace{s'_{j+1}\, s'_{j+2} \ldots}_{\pi' \in \mathit{Paths}(s_j)} \in \mathit{Paths}(s) \quad .$$

By assumption, π'' visits infinitely many a-states. In particular, there is an $i > j$ such that $s'_i \models a$. It now follows that

$$\pi' \;=\; s_j\, s'_{j+1} \ldots s'_{i-1}\, s'_i\, s'_{i+1}, \ldots \models \Diamond a$$

and, as this holds for any path $\pi' \in \mathit{Paths}(s_j)$, thus $s_j \models \forall\Diamond a$. This yields $\pi \models \Box\forall\Diamond a$ for all paths $\pi \in \mathit{Paths}(s)$. Thus, we have $s \models \forall\Box\,\forall\Diamond\, a$. ■

Remark 6.9. Weak Until

As for LTL (see Section 5.1.5), a slight variant of the until operator can be defined, viz. the weak-until operator, denoted W. The intuition behind this operator is that path π satisfies $\Phi\,\mathsf{W}\,\Psi$, for state formulae Φ and Ψ, if either $\Phi\,\mathsf{U}\,\Psi$ or $\Box\Phi$ holds. That is, the difference between until and weak until is that the latter does not require a Ψ-state to be reached eventually.

The weak-until operator in CTL cannot be defined directly starting from the LTL definition

$$\varphi\,\mathsf{W}\,\psi \;=\; \varphi\,\mathsf{U}\,\psi \;\vee\; \Box\varphi,$$

since $\exists(\varphi\,\mathsf{U}\,\psi \;\vee\; \Box\varphi)$ is not a syntactically correct CTL formula. However, using the LTL equivalence law $\varphi\,\mathsf{W}\,\psi \equiv \neg((\varphi \wedge \neg\psi)\,\mathsf{U}\,(\neg\varphi \wedge \neg\psi))$ and the duality between universal and existential quantification, the weak-until operator can be defined in CTL by

$$\begin{aligned} \exists(\Phi\,\mathsf{W}\,\Psi) &= \neg\forall((\Phi \wedge \neg\Psi)\,\mathsf{U}\,(\neg\Phi \wedge \neg\Psi)), \\ \forall(\Phi\,\mathsf{W}\,\Psi) &= \neg\exists((\Phi \wedge \neg\Psi)\,\mathsf{U}\,(\neg\Phi \wedge \neg\Psi)). \end{aligned}$$

Let us now check the semantics of $\exists\,\mathsf{W}$. From the above-defined duality, it follows that $s \models \exists(\Phi\,\mathsf{W}\,\Psi)$ if and only if there exists a path $\pi = s_0\, s_1\, s_2 \ldots$ that starts in s (i.e., $s_0 = s$) such that

$$\pi \not\models\; (\Phi \wedge \neg\Psi)\,\mathsf{U}\,(\neg\Phi \wedge \neg\Psi).$$

Such path exists if and only if

- either $s_j \models \Phi \wedge \neg\Psi$ for all $j \geqslant 0$, i.e., $\pi \models \Box(\Phi \wedge \neg\Psi)$, or
- there exists an index j such that

– $s_j \not\models \Phi \wedge \neg\Psi$ and $s_j \not\models \neg\Phi \wedge \neg\Psi$, i.e., $s_j \models \Psi$, and
– $s_i \models \Phi \wedge \neg\Psi$ for all $0 \leqslant i < j$.

This is equivalent to $\pi \models \Phi \,\mathsf{U}\, \Psi$.

Gathering these results yields

$$\begin{aligned} \pi \models \Phi \,\mathsf{W}\, \Psi \quad & \text{if and only if} \quad \pi \models \Phi \,\mathsf{U}\, \Psi \text{ or } \pi \models \Box(\Phi \wedge \neg\Psi), \\ & \text{if and only if} \quad \pi \models \Phi \,\mathsf{U}\, \Psi \text{ or } \pi \models \Box\Phi. \end{aligned}$$

Thus, the CTL formula $\exists(\Phi \,\mathsf{W}\, \Psi)$ is equivalent to $\exists(\Phi \,\mathsf{U}\, \Psi) \vee \exists\Box\Phi$. In the same way, one can check that the meaning of $\forall(\Phi \,\mathsf{W}\, \Psi)$ is as expected, i.e., $s \models \forall(\Phi \,\mathsf{W}\, \Psi)$ if and only if all paths starting in s fulfill $\Phi \,\mathsf{W}\, \Psi$ according to the LTL semantics of W. ■

Remark 6.10. The Semantics of Negation

For state s, we have $s \not\models \Phi$ if and only if $s \models \neg\Phi$. This, however, does not hold in general for transition systems. That is to say, it is possible that the statements $TS \not\models \Phi$ and $TS \not\models \neg\Phi$ *both hold*. This stems from the fact that there might be two initial states, s_0 and s_0', say, such that $s_0 \models \Phi$ and $s_0' \not\models \Phi$. Furthermore:

$$TS \not\models \neg\exists\varphi \;\text{ iff }\; \text{there exists a path } \pi \in \mathit{Paths}(TS) \text{ with } \pi \models \varphi.$$

This—at first glance surprising—equivalence is justified by the fact that the interpretation of CTL state formulae over transition systems is based on a universal quantification over the initial states. The statement $TS \not\models \neg\exists\varphi$ thus holds if and only if there exists an initial state $s_0 \in I$ with $s_0 \not\models \neg\exists\varphi$, i.e., $s_0 \models \exists\varphi$. On the other hand, $TS \models \exists\varphi$ requires that $s_0 \models \exists\varphi$ for *all* $s_0 \in I$. Consider the following transition system:

It follows that $s_0 \models \exists\Box\, a$, whereas $s_0' \not\models \exists\Box\, a$. Accordingly, $TS \not\models \neg\exists\Box\, a$ and $TS \not\models \exists\Box\, a$. ■

The semantics of CTL has been defined for a transition system without terminal states. This has the (technically) pleasant effect that all paths are infinite and simplifies the

definition of $\models$ for paths. In the following remark it is shown how to adapt the path semantics in case transition systems are considered with terminal states, i.e., when finite paths are possible.

Remark 6.11. CTL Semantics for Transition Systems with Terminal States

For finite maximal path fragment $\pi = s_0\, s_1\, s_2 \ldots s_n$ of length n, i.e., s_n is a terminal state, let

$$\begin{array}{lcll} \pi & \models & \bigcirc \Phi & \text{iff} \quad n > 0 \ \text{ and } \ s_1 \models \Phi, \\ \pi & \models & \Phi \,\mathsf{U}\, \Psi & \text{iff} \quad \text{there exists an index } j \in \mathbb{N} \text{ with } j \leqslant n, \text{ and} \\ & & & \qquad s_i \models \Phi, \ \text{ for } i = 0, 1, \ldots, j{-}1, \ \text{ and } s_j \models \Psi. \end{array}$$

Then, $s \models \forall\bigcirc \text{false}$ if and only if s is a terminal state. For the derived operators $\Diamond$ and $\Box$ we obtain

$$\begin{array}{lcll} \pi & \models & \Diamond\Phi & \text{iff} \quad \text{there exists an index } j \leqslant n \text{ with } s_j \models \Phi, \\ \pi & \models & \Box\Phi & \text{iff} \quad \text{for all } j \in \mathbb{N} \text{ with } j \leqslant n \text{ we have } s_j \models \Phi. \end{array}$$

■

6.2.3 Equivalence of CTL Formulae

CTL formulae Φ and Ψ are called equivalent whenever they are semantically identical, i.e., when for any state s it holds that[2] $s \models \Phi$ if and only if $s \models \Psi$.

Definition 6.12. Equivalence of CTL Formulae

CTL formulae Φ and Ψ (over AP) are called *equivalent*, denoted $\Phi \equiv \Psi$, if $Sat(\Phi) = Sat(\Psi)$ for all transition systems TS over AP. ■

Accordingly, $\Phi \equiv \Psi$ if and only if for any transition system TS we have:

$$TS \models \Phi \quad \text{if and only if} \quad TS \models \Psi \quad .$$

Besides the standard equivalence laws for the propositional logic fragment of CTL, there exist a number of equivalence rules for temporal modalities in CTL. An important set of equivalence laws is indicated in Figure 6.5. To understand the expansion laws, let us reconsider the expansion law for the until operator in LTL:

$$\varphi \,\mathsf{U}\, \psi \ \equiv \ \psi \ \vee \ (\varphi \ \wedge \ \bigcirc \ (\varphi \,\mathsf{U}\, \psi)).$$

[2] Recall that the notion CTL formula is used for a CTL state formula.

duality laws for path quantifiers

$$\forall\bigcirc\Phi \equiv \neg\exists\bigcirc\neg\Phi \qquad \exists\bigcirc\Phi \equiv \neg\forall\bigcirc\neg\Phi$$

$$\forall\Diamond\Phi \equiv \neg\exists\Box\neg\Phi \qquad \exists\Diamond\Phi \equiv \neg\forall\Box\neg\Phi$$

$$\begin{aligned}\forall(\Phi\,\mathsf{U}\,\Psi) &\equiv \neg\exists(\neg\Psi\,\mathsf{U}\,(\neg\Phi\wedge\neg\Psi)) \;\wedge\; \neg\exists\Box\neg\Psi\\ &\equiv \neg\exists((\Phi\wedge\neg\Psi)\,\mathsf{U}\,(\neg\Phi\wedge\neg\Psi)) \;\wedge\; \neg\exists\Box(\Phi\wedge\neg\Psi)\\ &\equiv \neg\exists((\Phi\wedge\neg\Psi)\,\mathsf{W}\,(\neg\Phi\wedge\neg\Psi))\end{aligned}$$

expansion laws

$$\begin{aligned}\forall(\Phi\,\mathsf{U}\,\Psi) &\equiv \Psi \vee (\Phi \wedge \forall\bigcirc\forall(\Phi\,\mathsf{U}\,\Psi))\\ \forall\Diamond\Phi &\equiv \Phi \vee \forall\bigcirc\forall\Diamond\Phi\\ \forall\Box\Phi &\equiv \Phi \wedge \forall\bigcirc\forall\Box\Phi\end{aligned}$$

$$\begin{aligned}\exists(\Phi\,\mathsf{U}\,\Psi) &\equiv \Psi \vee (\Phi \wedge \exists\bigcirc\exists(\Phi\,\mathsf{U}\,\Psi))\\ \exists\Diamond\Phi &\equiv \Phi \vee \exists\bigcirc\exists\Diamond\Phi\\ \exists\Box\Phi &\equiv \Phi \wedge \exists\bigcirc\exists\Box\Phi\end{aligned}$$

distributive laws

$$\forall\Box(\Phi\wedge\Psi) \equiv \forall\Box\Phi \wedge \forall\Box\Psi$$

$$\exists\Diamond(\Phi\vee\Psi) \equiv \exists\Diamond\Phi \vee \exists\Diamond\Psi$$

Figure 6.5: Some equivalence rules for CTL.

In CTL, similar expansion laws for $\exists(\Phi \,\mathsf{U}\, \Psi)$ and $\forall(\Phi \,\mathsf{U}\, \Psi)$ exist. For instance, we have that $\exists(\Phi \,\mathsf{U}\, \Psi)$ is equivalent to the fact that the current state either satisfies Ψ, or it satisfies Φ, and for *some* direct successor state, $\exists(\Phi \,\mathsf{U}\, \Psi)$ holds. The expansion laws for $\exists\Diamond\,\Phi$ and $\exists\Box\,\Phi$ can be simply derived from the expansion laws for $\exists\,\mathsf{U}$. The basic idea behind these laws—as for LTL—is to express the validity of a formula by a statement about the current state (without the need to use temporal operators) and a statement about the direct successors of this state (using either $\exists\bigcirc$ or $\forall\bigcirc$ depending on whether an existential or a universally quantified formula is treated). For instance, $\exists\Box\,\Phi$ is valid in state s if Φ is valid in s (a statement about the current state) and Φ holds for all states along some path starting at s (a statement about the successor states).

Not any law in LTL can be easily lifted to CTL. Consider, for example, the following statement:

$$\Diamond\,(\varphi \;\vee\; \psi) \;\equiv\; \Diamond\,\varphi \;\vee\; \Diamond\,\psi,$$

which is valid for any path. The same is true for:

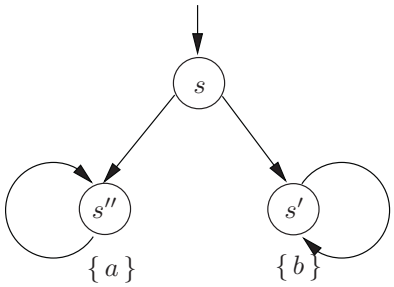
$$\exists\Diamond\,(\Phi \;\vee\; \Psi) \;\equiv\; \exists\Diamond\,\Phi \;\vee\; \exists\Diamond\,\Psi.$$

This can be seen as follows.

$\Leftarrow$: Assume that $s \models \exists\Diamond\,\Phi \;\vee\; \exists\Diamond\,\Psi$. Then, without loss of generality, we may assume that $s \models \exists\Diamond\,\Phi$. This means that there is some state s' (possibly $s = s'$), reachable from state s, such that $s' \models \Phi$. But then $s' \models \Phi \;\vee\; \Psi$. This means that there exists a reachable state from s which satisfies $\Phi \;\vee\; \Psi$. By the semantics of CTL it now follows that $s \models \exists\Diamond\,(\Phi \;\vee\; \Psi)$.

$\Rightarrow$: Let s be an arbitrary state such that $s \models \exists\Diamond\,(\Phi \;\vee\; \Psi)$. Then there exists a state s' (possibly $s = s'$) such that $s' \models \Phi \;\vee\; \Psi$. Without loss of generality we may assume that $s' \models \Phi$. But then we can conclude that $s \models \exists\Diamond\,\Phi$, as s' is reachable from s. Therefore we also have $s \models \exists\Diamond\,\Phi \;\vee\; \exists\Diamond\,\Psi$.

However, $\forall\Diamond\,(\Phi \;\vee\; \Psi) \not\equiv \forall\Diamond\,\Phi \;\vee\; \forall\Diamond\,\Psi$ since $\forall\Diamond\,(\Phi \;\vee\; \Psi) \;\Rightarrow\; \forall\Diamond\,\Phi \;\vee\; \forall\Diamond\,\Psi$ is invalid as shown by the following transition system:



For each path that starts in state s we have that $\Diamond\,(a \vee b)$ holds, so $s \models \forall\Diamond\,(a \vee b)$. This follows directly from the fact that each path visits either state s' or state s'' eventually, and $s' \models a \vee b$ and the same applies to s''. However, state s does not satisfy $\forall\Diamond\, a \vee \forall\Diamond\, b$. For instance, path $s\,(s'')^\omega \models \Diamond\, a$ but $s\,(s'')^\omega \not\models \Diamond\, b$. Thus, $s \not\models \forall\Diamond\, b$. By a similar reasoning applied to path $s\,(s')^\omega$ it follows that $s \not\models \forall\Diamond\, a$. Thus, $s \not\models \forall\Diamond\, a \vee \forall\Diamond\, b$. Stated in words, not all computations that start in state s eventually reach an a-state nor do they all eventually reach a b-state.

6.2.4 Normal Forms for CTL

The duality law for $\forall\bigcirc\,\Phi$ shows that $\forall\bigcirc$ can be treated as a derived operator of $\exists\bigcirc$. That is to say, the basic operators $\exists\bigcirc$, $\exists\,\mathsf{U}$, and $\forall\,\mathsf{U}$ would have been sufficient to define the syntax of CTL. The following theorem demonstrates that we can even omit the universal path quantifier and define all temporal modalities in CTL using the basic operators $\exists\bigcirc$, $\exists\,\mathsf{U}$, and $\exists\Box$.

Definition 6.13. Existential Normal Form (for CTL)

For $a \in AP$, the set of CTL state formulae in *existential normal form* (ENF, for short) is given by

$$\Phi \quad ::= \quad \text{true} \;\Big|\; a \;\Big|\; \Phi_1 \wedge \Phi_2 \;\Big|\; \neg\Phi \;\Big|\; \exists\bigcirc\,\Phi \;\Big|\; \exists(\Phi_1\,\mathsf{U}\,\Phi_2) \;\Big|\; \exists\Box\,\Phi.$$

■

Theorem 6.14. Existential Normal Form for CTL

For each CTL formula there exists an equivalent CTL formula in ENF.

Proof: The following duality laws allow elimination of the universal path quantifier and thus provide a translation of CTL formulae into equivalent ENF formulae:

$$\begin{aligned} \forall \bigcirc \Phi &\equiv \neg \exists \bigcirc \neg \Phi, \\ \forall (\Phi \,\mathsf{U}\, \Psi) &\equiv \neg \exists (\neg \Psi \,\mathsf{U}\, (\neg \Phi \wedge \neg \Psi)) \;\wedge\; \neg \exists \Box \neg \Psi. \end{aligned}$$

■

Recall that the basis syntax of CTL only uses $\exists \bigcirc$, $\exists \,\mathsf{U}$ and $\forall \bigcirc$ and $\forall \,\mathsf{U}$. Thus, the two rules used in the proof of Theorem 6.14 allow the removal of all universal quantifiers from a given CTL formula. However, when implementing the translation from CTL formulae to ENF formulae one might use analogous rules for the derived operators, such as

$$\begin{aligned} \forall \Diamond \Phi &\equiv \neg \exists \Box \neg \Phi, \\ \forall \Box \Phi &\equiv \neg \exists \Diamond \neg \Phi \;=\; \neg \exists (\mathrm{true} \,\mathsf{U}\, \Phi). \end{aligned}$$

Since the rewrite rule for $\forall \,\mathsf{U}$ triples the occurrences of the right formula Ψ, the translation from CTL to ENF can cause an exponential blowup.

Another normal form of importance is the *positive normal form*. A CTL formula is said to be in positive normal form (PNF, for short) whenever negations only occur adjacent to atomic propositions. That is, e.g., $\neg \forall (a \,\mathsf{U}\, \neg b)$ is not in PNF, whereas $\exists (\neg a \wedge \neg b \,\mathsf{U}\, a)$ is in PNF. To ensure that every CTL formula is equivalent to a formula in PNF, for each operator a dual operator is necessary. We have that conjunction and disjunction are dual, and that $\bigcirc$ is dual to itself. As for LTL, we adopt the weak until operator W as a dual operator of U.

Definition 6.15. Positive Normal Form (for CTL)

The set of CTL state formulae in *positive normal form* (PNF, for short) is given by

$$\Phi \;::=\; \mathrm{true} \;\big|\; \mathrm{false} \;\big|\; a \;\big|\; \neg a \;\big|\; \Phi_1 \wedge \Phi_2 \;\big|\; \Phi_1 \;\vee\; \Phi_2 \;\big|\; \exists \varphi \;\big|\; \forall \varphi$$

where $a \in AP$ and the path formulae are given by

$$\varphi \;::=\; \bigcirc \Phi \;\big|\; \Phi_1 \,\mathsf{U}\, \Phi_2 \;\big|\; \Phi_1 \,\mathsf{W}\, \Phi_2.$$

■

Theorem 6.16. Existence of Equivalent PNF Formulae

For each CTL formula there exists an equivalent CTL formula in PNF.

Proof: Any CTL formula can be transformed into PNF by successively "pushing" negations "inside" the formula at hand. This is facilitated by the following equivalence laws:

$$\begin{array}{lcl}
\neg\text{true} & \equiv & \text{false} \\
\neg\neg\Phi & \equiv & \Phi \\
\neg(\Phi \wedge \Psi) & \equiv & \neg\Phi \ \vee \ \neg\Psi \\
\neg\forall\bigcirc\Phi & \equiv & \exists\bigcirc\neg\Phi \\
\neg\exists\bigcirc\Phi & \equiv & \forall\bigcirc\neg\Phi \\
\neg\forall(\Phi \,\mathsf{U}\, \Psi) & \equiv & \exists((\Phi \wedge \neg\Psi) \,\mathsf{W}\, (\neg\Phi \wedge \neg\Psi)) \\
\neg\exists(\Phi \,\mathsf{U}\, \Psi) & \equiv & \forall((\Phi \wedge \neg\Psi) \,\mathsf{W}\, (\neg\Phi \wedge \neg\Psi)).
\end{array}$$

■

Due to the fact that in the rules for $\forall\,\mathsf{U}$ and $\exists\,\mathsf{U}$ the number of occurrences of Ψ (and Φ) is doubled, the length of an equivalent CTL formula may be exponentially longer than the original CTL formula. [3] The same phenomenon appeared in defining the PNF for LTL when using the weak-until operator. As for LTL, this exponential blowup can be avoided by using the release operator, which in CTL can be defined by: $\exists(\Phi \,\mathsf{R}\, \Psi) = \neg\forall((\neg\Phi) \,\mathsf{U}\, (\neg\Psi))$ and $\forall(\Phi \,\mathsf{R}\, \Psi) = \neg\exists((\neg\Phi) \,\mathsf{U}\, (\neg\Psi))$.

6.3 Expressiveness of CTL vs. LTL

Although many relevant properties of reactive systems can be specified in LTL and CTL, the logics CTL and LTL are incomparable according to their expressiveness. More precisely, there are properties that one can express in CTL, but that cannot be expressed in LTL, and vice versa.

Let us first define what it means for CTL and LTL formulae to be equivalent. Intuitively speaking, equivalent means "express the same thing". More precisely:

Definition 6.17. Equivalence between CTL- and LTL Formulae

CTL formula Φ and LTL formula φ (both over AP) are *equivalent*, denoted $\Phi \equiv \varphi$, if for any transition system TS over AP:

$$TS \models \Phi \quad \text{if and only if} \quad TS \models \varphi.$$

[3] Although the rewrite rules for $\neg\forall\,\mathsf{U}$ and $\neg\exists\,\mathsf{U}$ could be simplified by $\neg\forall(\Phi \,\mathsf{U}\, \Psi) \equiv \exists\Big((\neg\Psi) \,\mathsf{W}\, (\neg\Phi \wedge \neg\Psi)\Big)$ and $\neg\exists(\Phi \,\mathsf{U}\, \Psi) \equiv \forall\Big((\neg\Psi) \,\mathsf{W}\, (\neg\Phi \wedge \neg\Psi)\Big)$, there is still a duplication of formula Ψ.

■

The LTL formula φ holds in state s of a transition system whenever all paths starting in s satisfy φ. Given this (semantical) universal quantification over paths, it seems natural that, e.g., the LTL formula $\Diamond a$ is equivalent to the CTL formula $\forall\Diamond a$. This seems to suggest that for a given CTL formula, an equivalent LTL formula is obtained by simply omitting all universal path quantifiers (as these are implicit in LTL). The following result by Clarke and Draghicescu [85] (for which the proof is omitted) shows that dropping all (universal and existential) quantifiers is a safe way to generate an equivalent LTL formula, provided there are equivalent LTL formulae:

Theorem 6.18. Criterion for Transforming CTL Formulae into Equivalent LTL Formulae

Let Φ be a CTL formula, and φ the LTL formula that is obtained by eliminating all path quantifiers in Φ. Then:

$$\Phi \equiv \varphi \text{ or there does not exist any LTL formula that is equivalent to } \Phi.$$

For the following CTL formulae, an equivalent LTL formula is obtained by simply omitting all path quantifiers: a, $\forall\bigcirc a$, $\forall(a \,\mathsf{U}\, b)$, $\forall\Diamond a$, $\forall\Box a$, and $\forall\Box\forall\Diamond a$. The fact that the CTL formula $\forall\Box\forall\Diamond a$ is equivalent to the LTL formula $\Box\Diamond a$ has been established earlier in Remark 6.8 (326). However, $\forall\Diamond\forall\Box a$ and $\Diamond\Box a$ are not equivalent. The LTL formula $\Diamond\Box a$ ensures that a will eventually forever (i.e., continuously from some point on) hold. The semantics of $\forall\Diamond\forall\Box a$ is different, however. The CTL formula $\forall\Diamond\forall\Box a$ asserts that on any computation, eventually some state, s say, is reached such that $s \models \forall\Box a$. Note that

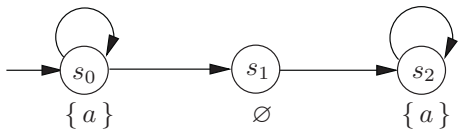
$$s \models \forall\Diamond \underbrace{\forall\Box a}_{\Phi}$$

if and only if for any path $\pi = s_0\, s_1\, s_2 \ldots \in \mathit{Paths}(s)$, $s_j \models \Phi$ for some j. For $\Phi = \forall\Box a$, this entails that for any such path π there is some state s_j such that all reachable states from s_j satisfy the atomic proposition a.

Lemma 6.19. Persistence

The CTL formula $\forall\Diamond\forall\Box a$ and the LTL formula $\Diamond\Box a$ are not *equivalent.*

Proof: Consider the following transition system TS over $AP = \{ a \}$:



The initial state s_0 satisfies the LTL formula $\Diamond\Box a$, since each path starting in s_0 eventually remains forever in one of the two states s_0 or s_2, which are both labeled with a. The CTL formula $\forall\Diamond\forall\Box a$, however, does not hold in s_0, since we have $s_0^\omega \not\models \Diamond\forall\Box a$ (as $s_0 \not\models \forall\Box a$). This is due to the fact that the path $s_0^* s_1 s_2^\omega$ passes through the $\neg a$-state s_1. Thus, s_0^ω is a path starting in s_0 which will never reach a state satisfying $\forall\Box a$, i.e., $s_0^\omega \not\models \Diamond\forall\Box a$. Accordingly, it follows that

$$s_0 \not\models \forall\Diamond\forall\Box a.$$

■

Given that the CTL formulae $\forall\Diamond\forall\Box a$ and the LTL formula $\Diamond\Box a$ are not equivalent and the fact that $\Diamond\Box a$ is obtained from $\forall\Diamond\forall\Box a$ by eliminating the universal path quantifiers, it follows from Theorem 6.18 that there does not exist an LTL formula that is equivalent to $\forall\Diamond\forall\Box a$. In a similar way, it can be shown that the CTL formulae $\forall\Diamond(a \wedge \forall\bigcirc a)$ and $\Diamond(a \wedge \bigcirc a)$ are not equivalent, and thus, the requirement $\forall\Diamond(a \wedge \forall\bigcirc a)$ cannot be expressed in LTL.

Lemma 6.20. Eventually an a-State with only direct a-Successors

The CTL formula $\forall\Diamond(a \wedge \forall\bigcirc a)$ and the LTL formula $\Diamond(a \wedge \bigcirc a)$ are not *equivalent.*

Proof: Consider the transition system depicted in Figure 6.6. All paths that start in the initial state s_0 have either as prefix the path fragment $s_0 s_1$ or $s_0 s_3 s_4$. Clearly, all such paths satisfy the LTL formula $\Diamond(a \wedge \bigcirc a)$, and so, $s_0 \models \Diamond(a \wedge \bigcirc a)$. On the other hand, however, $s_0 \not\models \forall\Diamond(a \wedge \forall\bigcirc a)$ as the path $s_0 s_1 (s_2)^\omega$ does not satisfy $\Diamond(a \wedge \forall\bigcirc a)$. This follows from the fact that state s_0 has the non-a-state s_3 as direct successor, i.e., $s_0 \not\models a \wedge \forall\bigcirc a$. ■

These examples show that certain requirements that can be expressed in CTL, cannot be expressed in LTL. The following theorem provides, in addition, some examples of LTL formulae for which no equivalent CTL formula exists. This establishes that the expressiveness of the temporal logics LTL and CTL are incomparable.

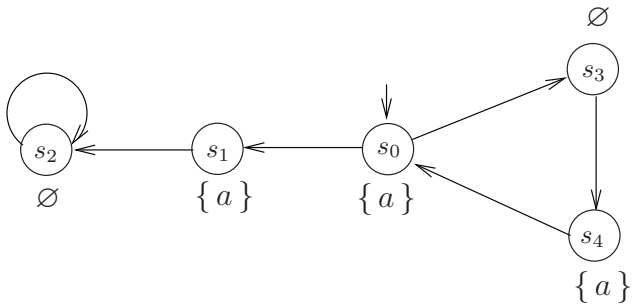


Figure 6.6: Transition system for $\forall\Diamond\,(a \wedge \forall\bigcirc\, a)$.

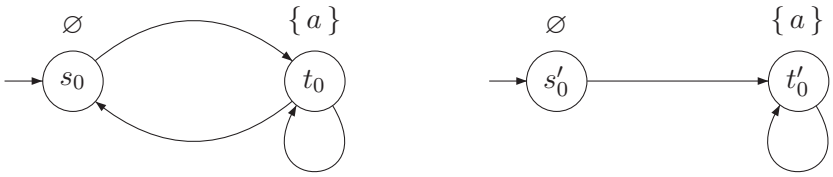


Figure 6.7: The base transition systems: TS_0 (left) and TS'_0 (right).

Theorem 6.21. Incomparable Expressiveness of CTL and LTL

(a) There exist LTL formulae for which no equivalent CTL formula exists. This holds for, for instance

$$\Diamond\Box a \qquad \textit{or} \qquad \Diamond(a \wedge \bigcirc\, a).$$

(b) There exist CTL formulae for which no equivalent LTL formula exists. This holds for, for instance

$$\forall\Diamond\forall\Box\, a \quad \textit{and} \quad \forall\Diamond\,(a \wedge \forall\bigcirc\, a) \quad \textit{and} \quad \forall\Box\,\exists\Diamond\, a.$$

Proof:

(a) Consider the formula $\Diamond\,\Box\, a$. The proof for $\Diamond\,(a \wedge \bigcirc\, a)$ goes along similar lines and is omitted here. Consider the two series of transition systems $TS_0, TS_1, TS_2, \ldots$ and $TS'_0, TS'_1, TS'_2, \ldots$ that are inductively defined as follows (see Figures 6.7 and 6.8).

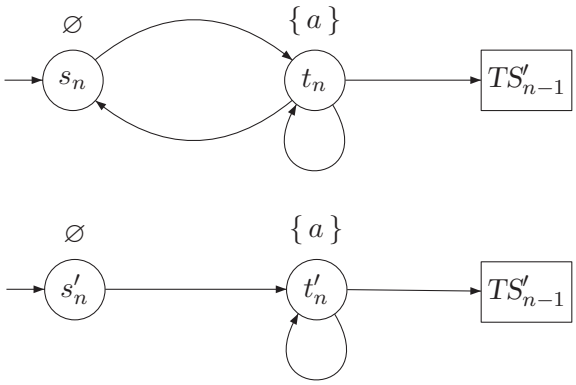


Figure 6.8: Inductive construction of TS_n (upper) and TS'_n (lower).

For all transition systems, $AP = \{ a \}$, and the action labels are not important. Let, for $n \geqslant 0$,

$$TS_n = (S_n, \{ \tau \}, \rightarrow_n, \{ s_n \}, \{ a \}, L_n)$$

and

$$TS_n = (S'_n, \{ \tau \}, \rightarrow'_n, \{ s'_n \}, \{ a \}, L'_n)$$

where $S_0 = \{ s_0, t_0 \}$, $S'_0 = \{ s'_0, t'_0 \}$, and for $n > 0$:

$$S_n \;=\; S'_{n-1} \cup \{ s_n, t_n \} \quad \text{and} \quad S'_n \;=\; S'_{n-1} \cup \{ s'_n, t'_n \}.$$

The labeling functions are defined such that all states t_i are labeled with $\{ a \}$ and all states s_i are labeled with $\varnothing$. Thus, $L_0(s_0) = \varnothing$ and $L_0(t_0) = \{ a \}$, and for $n > 0$, the labels of all states in TS'_{n-1} remain the same and are extended with

$$L_n(s_n) = L'_n(s'_n) = \varnothing \quad \text{and} \quad L_n(t_n) = L'_n(t'_n) = \{ a \}.$$

Finally, the transition relations $\rightarrow_n$ and $\rightarrow'_n$ contain $\rightarrow'_{n-1}$ (where $\rightarrow_{-1} = \varnothing$), as well as the transitions

$$\begin{aligned} TS_n : &\quad s_n \rightarrow_n t_n, \quad t_n \rightarrow_n t_n, \quad t_n \rightarrow_n s'_{n-1}, \quad t_n \rightarrow_n s_n \\ TS'_n : &\quad s'_n \rightarrow'_n t'_n, \quad t'_n \rightarrow'_n t'_n, \quad t'_n \rightarrow'_n s'_{n-1} \end{aligned}$$

where the action labels are omitted for simplicity.

Thus, the only difference between TS_n and TS'_n is the fact that TS_n includes the edge $t_n \rightarrow s_n$, whereas this edge is absent in TS'_n. Some example instantiations of TS_n and TS'_n are indicated in Figure 6.9.

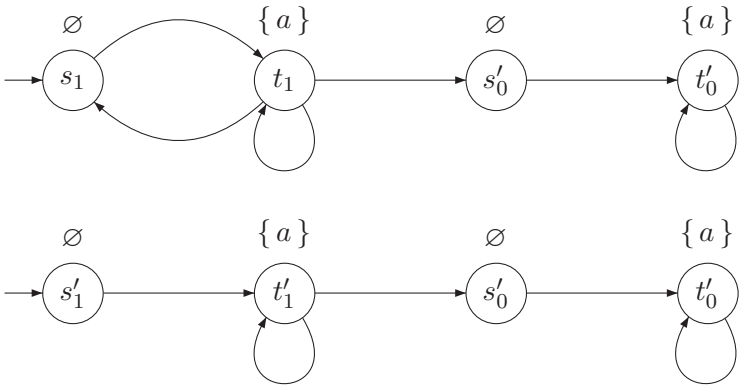


Figure 6.9: The transition systems TS_1 (upper) and TS'_1 (lower).

It follows from the construction of TS_n and TS'_n that

$$TS_n \not\models \Diamond\Box a \quad \text{and} \quad TS'_n \models \Diamond\Box a \quad \text{for all } n \geqslant 0.$$

This can be proven as follows. TS_n contains the initial path $(s_n\, t_n)^\omega$ that visits s_n and t_n in an alternating fashion. We have that

$$\mathit{trace}((s_n\, t_n)^\omega) \;=\; \varnothing\,\{\,a\,\}\,\varnothing\,\{\,a\,\}\,\varnothing\ldots \quad \text{thus} \quad \mathit{trace}((s_n\, t_n)^\omega) \not\models \Diamond\,\Box\, a.$$

As the considered path is initial, it follows that $TS_n \not\models \Diamond\,\Box\, a$. On the other hand, as TS'_n has no opportunity to infinitely often return to an $\neg a$-state, each initial path in TS'_n is of the following form:

$$\pi'_i \;=\; s'_n\, t'_n \ldots s'_i\, (t'_i)^\omega$$

for some i. Due to the fact that

$$\mathit{trace}(\pi'_i) \;=\; \varnothing\,\{\,a\,\}\,\varnothing\ldots\varnothing\,(\{\,a\,\})^\omega$$

it follows that $\mathit{trace}(\pi'_i) \models \Diamond\,\Box\, a$. As this applies to any initial path of TS'_n, we have $TS'_n \models \Diamond\,\Box\, a$.

By means of induction on n, it can be proven that TS_n and TS'_n cannot be distinguished by any CTL formula of length at most n. That is to say, for all $n \geqslant 0$,

$$\forall \text{CTL formula } \Phi \text{ with } |\Phi| \leqslant n:\; TS_n \models \Phi \quad \text{if and only if} \quad TS'_n \models \Phi.$$

(The proof of this fact is left to the interested reader.)

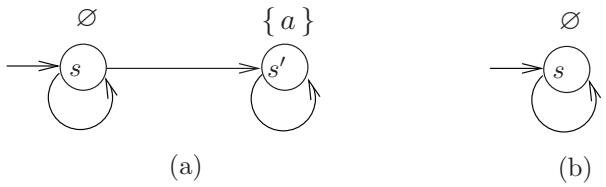


Figure 6.10: Two transition systems for $\forall\Box\,\exists\Diamond\, a$.

The final step of the proof is now as follows. Assume there is a CTL formula Φ that is equivalent to $\Diamond\Box\, a$. Let $n = |\Phi|$ be the length of the formula Φ and

$$TS \;=\; TS_n, \quad TS' \;=\; TS'_n.$$

On the one hand, it follows from $TS \not\models \Diamond\Box\, a$ and $TS' \models \Diamond\Box\, a$ that

$$TS \not\models \Phi \text{ and } TS' \models \Phi.$$

On the other hand, it results from the fact that TS_n and TS'_n cannot be distinguished by a CTL formula (of length at most n) that Φ has the same truth-value under TS and under TS'. This yields a contradiction.

(b) We concentrate on the proof that there does not exist an equivalent formulation of the CTL formula $\forall\Box\,\exists\Diamond\, a$ in LTL. The fact that there do not exist equivalent LTL formulations of the CTL formulae $\forall\Diamond\,\forall\Box\, a$ and $\forall\Diamond\,(a \wedge \forall\bigcirc\, a)$ follows directly from Lemmas 6.19 and 6.20 respectively, and Theorem 6.18. The proof for $\forall\Box\,\exists\Diamond\, a$ is by contraposition. Let φ be an LTL formula that is equivalent to $\forall\Box\,\exists\Diamond\, a$. Consider the transition system TS depicted in Figure 6.10(a). As $TS \models \forall\Box\,\exists\Diamond\, a$, and $\varphi \equiv \forall\Diamond\,\exists\Diamond\, a$, it follows that $TS \models \varphi$, and therefore

$$\mathit{Traces}(TS) \;\subseteq\; \mathit{Words}(\varphi).$$

Since $\pi = s^\omega$ is a path in TS, it follows that

$$\mathit{trace}(\pi) \;=\; \varnothing\,\varnothing\,\varnothing\,\varnothing\ldots \in \mathit{Traces}(TS) \;\subseteq\; \mathit{Words}(\varphi).$$

Now consider the transition system TS' depicted in Figure 6.10(b). Note that TS' is part of transition system TS. The paths starting in s in TS' are also paths starting from s in TS, so we have $s \models \varphi$ in TS' and thus $TS' \models \varphi$. However, $s \not\models \forall\Box\,\exists\Diamond\, a$, since $\exists\Diamond\, a$ is never valid along the only path s^ω, and thus $TS' \not\models \forall\Box\,\exists\Diamond\, a$. This is a contradiction.

■

Note that the CTL formula $\forall\Box\,\exists\Diamond\, a$ is of significant practical use, since it expresses the fact that it is possible to reach a state for which a holds irrespective of the current state. If a characterizes a state where a certain error is repaired, the formula expresses the fact that it is always possible to recover from that error.

6.4 CTL Model Checking

This section is concerned with CTL model checking, which means a decision algorithm that checks whether $TS \models \Phi$ for a given transition system TS and CTL formula Φ. Throughout this section, it is assumed that TS is finite, and has no terminal states. We will see that CTL-model checking can be performed by a recursive procedure that calculates the satisfaction set for all subformulae of Φ and finally checks whether all initial states belong to the satisfaction set of Φ.

Throughout this section, we consider CTL formulae in ENF, i.e., CTL formulae built by the basic modalities $\exists\bigcirc$, $\exists\,\mathsf{U}$, and $\exists\Box$. This requires algorithms that generate $Sat(\exists\bigcirc\,\Phi)$, $Sat(\exists(\Phi\,\mathsf{U}\,\Psi))$, and $Sat(\exists\Box\,\Phi)$ when $Sat(\Phi)$ and $Sat(\Psi)$ are given. Although each CTL formula can be transformed algorithmically into an equivalent ENF formula, for an implementation of the CTL model-checking algorithm it is recommended to use similar techniques to handle universal quantification, i.e., to design algorithms that generate $Sat(\forall\bigcirc\,\Phi)$, $Sat(\forall(\Phi\,\mathsf{U}\,\Psi))$, and $Sat(\forall\Box\,\Phi)$ directly from $Sat(\Phi)$ and $Sat(\Psi)$. (The same holds for other derived operators such as W or R.)

6.4.1 Basic Algorithm

The model-checking problem for CTL is to verify for a given transition system TS and CTL formula Φ whether $TS \models \Phi$. That is, we need to establish whether the formula Φ is valid in each initial state s of TS. The basic procedure for CTL model checking is rather straightforward:

- the set $Sat(\Phi)$ of all states satisfying Φ is computed recursively, and
- it follows that $TS \models \Phi$ if and only if $I \subseteq Sat(\Phi)$

where I is the set of initial states of TS. Note that by computing $Sat(\Phi)$ a more general problem than just checking whether $TS \models \Phi$ is solved. In fact, it checks for *any* state s in S whether $s \models \Phi$, and not just for the initial states. This is sometimes referred

to as a *global* model-checking procedure. The basic idea of the algorithm is sketched in Algorithm 13 where $Sub(\Phi)$ is the set of subformulae of Φ. In the sequel of this section it is assumed that TS is finite and has no terminal states.

Algorithm 13 CTL model checking (basic idea)

Input: finite transition system TS and CTL formula Φ (both over AP)
Output: $TS \models \Phi$

```
                                   (* compute the sets Sat(Φ) = { s ∈ S | s ⊨ Φ } *)
for all i ⩽ | Φ | do
  for all Ψ ∈ Sub(Φ) with | Ψ | = i do
    compute Sat(Ψ) from Sat(Ψ')    (* for maximal genuine Ψ' ∈ Sub(Ψ) *)
  od
od
return I ⊆ Sat(Φ)
```

The recursive computation of $Sat(\Phi)$ basically boils down to a *bottom-up traversal* of the *parse tree* of the CTL state formula Φ. The nodes of the parse tree represent the subformulae of Φ. The leaves stand for the constant true or an atomic proposition $a \in AP$. All inner nodes are labeled with an operator. For ENF formulae the labels of the inner nodes are $\neg$, $\wedge$, $\exists\bigcirc$, $\exists\,\mathsf{U}$, or $\exists\Box$.

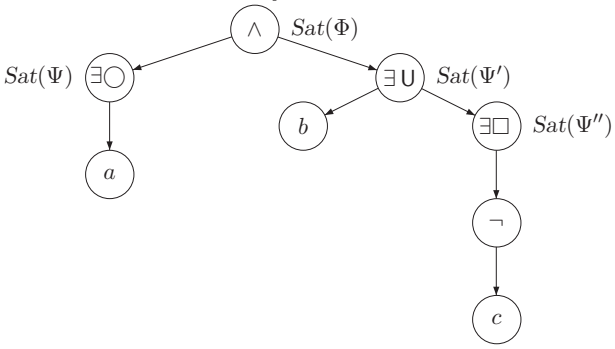
For each node of the parse tree, i.e., for each subformula Ψ of Φ, the set $Sat(\Psi)$ of states is computed for which Ψ holds. This computation is carried out in a bottom-up fashion, starting from the leaves of the parse tree and finishing at the root of the tree, the (unique) node in the parse tree that corresponds to Φ. At an intermediate node, the results of the computations of its children are used and combined in an appropriate way to establish the states of its associated subformula. The type of computation at such node, v say, depends on the operator (e.g., $\wedge$, $\exists\bigcirc$, or $\exists\,\mathsf{U}$) that is at the "top level" of the subformula treated. The children of node v stand for the *maximal genuine subformulae* of the formula Ψ_v that is represented by v. As soon as $Sat(\Psi)$ is computed, subformula Ψ is (theoretically) replaced by a new atomic proposition a_Ψ, and the labeling function L is adjusted as follows: a_Ψ is added to $L(s)$ if and only if $s \in Sat(\Psi)$. Once the bottom-up computations continue with the father, w say, of node v, $\Psi = \Psi_v$ is a maximal genuine subformula of Ψ_w, and all states that are labeled with a_Ψ are known to satisfy Ψ. In fact, one might say that Ψ is replaced in the formula by the atomic proposition a_Ψ. This technique will be of importance when treating the model checking of CTL with fairness.

Example 6.22.

Consider the following state formula over $AP = \{ a, b, c \}$:

$$\Phi \;=\; \underbrace{\exists \bigcirc a}_{\Psi} \;\wedge\; \underbrace{\exists (b \,\mathsf{U}\, \underbrace{\exists \Box \neg c}_{\Psi''})}_{\Psi'} \quad .$$

The indicated formulae Ψ and Ψ' are the maximal proper subformulae of Φ, while Ψ'' is a maximal proper subformula of Ψ. The syntax tree for Φ is of the following form:



The satisfaction sets for the leaves result directly from the labeling function L. The treatment of subformula $\neg c$ only needs the satisfaction set for $Sat(c)$ to be complemented. Using $Sat(\neg c)$, the set $Sat(\exists \Box \neg c)$ can be computed. The subformula Ψ'' can now be replaced by the fresh atomic proposition a_3 where $a_3 \in L(s)$ if and only if $s \in Sat(\exists \Box \neg c)$. The computation now continues with determining $Sat(\exists (b \,\mathsf{U}\, a_3))$. In a similar way, $Sat(\exists \bigcirc a)$ can be computed by means of $Sat(a)$.

Once the subformulae Ψ and Ψ' are treated, they can be replaced by the atomic propositions a_1, a_2, respectively, such that

$$a_1 \in L(s) \quad \text{iff} \quad s \models \exists \bigcirc a \quad \text{and} \quad a_2 \in L(s) \quad \text{iff} \quad s \models \exists (b \,\mathsf{U}\, a_3).$$

The formula that is to be treated for the root node simply thus is: $\Phi' \;=\; a_1 \wedge a_2$. $Sat(\Phi')$ results from intersecting $Sat(a_1) = Sat(\Psi)$ and $Sat(a_2) = Sat(\Psi')$. Note that a_1, a_2, and a_3 are fresh atomic propositions, i.e., $\{ a_1, a_2, a_3 \} \cap AP = \varnothing$. The above procedure thus is considered over $AP' = AP \cup \{ a_1, a_2, a_3 \}$. ■

Theorem 6.23. Characterization of $\mathrm{Sat}(\cdot)$ ***for CTL formulae in ENF***

Let $TS = (S, Act, \to, I, AP, L)$ be a transition system without terminal states. For all CTL formulae Φ, Ψ over AP it holds that

(a) $Sat(true) = S$,

(b) $Sat(a) = \{ s \in S \mid a \in L(s) \}$, *for any* $a \in AP$,

(c) $Sat(\Phi \wedge \Psi) = Sat(\Phi) \cap Sat(\Psi)$,

(d) $Sat(\neg\Phi) = S \setminus Sat(\Phi)$,

(e) $Sat(\exists\bigcirc \Phi) = \{ s \in S \mid Post(s) \cap Sat(\Phi) \neq \varnothing \}$,

(f) $Sat(\exists(\Phi \,\mathsf{U}\, \Psi))$ *is the smallest subset* T *of* S, *such that*

(1) $Sat(\Psi) \subseteq T$ *and (2)* $s \in Sat(\Phi)$ *and* $Post(s) \cap T \neq \varnothing$ *implies* $s \in T$,

(g) $Sat(\exists\Box \Phi)$ *is the largest subset* T *of* S, *such that*

(3) $T \subseteq Sat(\Phi)$ *and (4)* $s \in T$ *implies* $Post(s) \cap T \neq \varnothing$.

In the clauses (f) and (g), the terms "smallest" and "largest" should be interpreted with respect to the partial order induced by set inclusion.

Proof: The validity of the claims indicated in (a) through (e) is straightforward. We only prove the propositions (f) and (g):

Proof of (f): The proof of this claim consists of two parts:

(i) Show that $T = Sat(\exists(\Phi \,\mathsf{U}\, \Psi))$ satisfies (1) and (2). From the expansion law

$$\exists(\Phi \,\mathsf{U}\, \Psi) \equiv \Psi \vee (\Phi \wedge \exists\bigcirc \exists(\Phi \,\mathsf{U}\, \Psi)),$$

it directly follows that T satisfies the properties (1) and (2).

(ii) Show that for any T satisfying properties (1) and (2) we have

$$Sat(\exists(\Phi \,\mathsf{U}\, \Psi)) \subseteq T.$$

This is proven as follows. Let $s \in Sat(\exists(\Phi \,\mathsf{U}\, \Psi))$. Distinguish between $s \in Sat(\Psi)$ and $s \notin Sat(\Psi)$. If $s \in Sat(\Psi)$, it follows from (1) that $s \in T$. In case $s \notin Sat(\Psi)$, there exists a path $\pi = s_0\, s_1\, s_2 \ldots$ starting in $s{=}s_0$, such that $\pi \models \Phi \,\mathsf{U}\, \Psi$. Let $n > 0$, such that $s_i \models \Phi$, $0 \leqslant i < n$, and $s_n \models \Psi$. Then:

- $s_n \in Sat(\Psi) \subseteq T$,
- $s_{n-1} \in T$, since $s_n \in Post(s_{n-1}) \cap T$ and $s_{n-1} \in Sat(\Phi)$,
- $s_{n-2} \in T$, since $s_{n-1} \in Post(s_{n-2}) \cap T$ and $s_{n-2} \in Sat(\Phi)$,

- ……,
- $s_1 \in T$, since $s_2 \in \mathit{Post}(s_1) \cap T$ and $s_1 \in \mathit{Sat}(\Phi)$, and finally
- $s_0 \in T$, since $s_1 \in \mathit{Post}(s_0) \cap T$ and $s_0 \in \mathit{Sat}(\Phi)$.

It thus follows that $s = s_0 \in T$.

Proof of (g): The proof of this claim consists of two parts:

(i) Show that $T = \mathit{Sat}(\exists\Box\Phi)$ satisfies (3) and (4). From the expansion law

$$\exists\Box\,\Phi \quad \equiv \quad \Phi \wedge \exists\bigcirc\exists\Box\,\Phi,$$

it directly follows that T satisfies the properties (3) and (4).

(ii) Show that for any T satisfying properties (3) and (4)

$$T \subseteq \mathit{Sat}(\exists\Box\Phi).$$

This proof goes as follows. Let $T \subseteq S$ satisfy (3) and (4) and $s \in T$. Let $\pi = s_0\, s_1\, s_2 \ldots$ be a path starting in $s{=}s_0$. (As TS has no terminal states, such a path exists.) Then we derive

- $s_0 = s$.
- Since $s_0 \in T$, there exists a state $s_1 \in \mathit{Post}(s_0) \cap T$.
- Since $s_1 \in T$, there exists a state $s_2 \in \mathit{Post}(s_1) \cap T$.
- ……

Here, property (4) is exploited in every step. From property (3), it follows that

$$s_i \in T \ \subseteq \mathit{Sat}(\Phi), \quad i \geqslant 0.$$

Thus, $\pi = s_0\, s_1\, s_2 \ldots$ satisfies $\Box\Phi$. It follows that

$$s \in \mathit{Sat}(\exists\Box\Phi).$$

As this reasoning applies to any $s \in T$, it follows that $T \subseteq \mathit{Sat}(\exists\Box\,\Phi)$.

■

Remark 6.24. Alternative Formulation of $\mathit{Sat}(\exists(\Phi\,\mathsf{U}\,\Psi)$ *and* $\mathit{Sat}(\exists\Box\,\Phi)$

The characterizations of the sets $\mathit{Sat}(\exists(\Phi\,\mathsf{U}\,\Psi))$ and $\mathit{Sat}(\exists\Box\,\Phi)$ indicated in Theorem 6.23

are based upon the fixed-point equation induced by the expansion laws for $\exists(\Phi \,\mathsf{U}\, \Psi)$ and $\exists\Box\, \Phi$, respectively. Consider, for instance, the expansion law

$$\exists(\Phi \,\mathsf{U}\, \Psi) \;\equiv\; \Psi \;\vee\; (\Phi \,\wedge\, \exists\bigcirc \exists(\Phi \,\mathsf{U}\, \Psi)).$$

The recursive nature of this law suggests to considering the CTL formula $\exists(\Phi \,\mathsf{U}\, \Psi)$ as a *fixed point* of the logical equation

$$F \;\equiv\; \Psi \,\vee\, (\Phi \,\wedge\, \exists\bigcirc F).$$

By the expansion law $F = \exists(\Phi \,\mathsf{U}\, \Psi)$ is a solution, but there are also other solutions that are not equivalent to $\exists(\Phi \,\mathsf{U}\, \Psi)$, such as $F = \exists(\Phi \,\mathsf{W}\, \Psi)$ (see Remark 6.25). However, a unique characterization of $\exists(\Phi \,\mathsf{U}\, \Psi)$ is obtained by the fact that $\exists(\Phi \,\mathsf{U}\, \Psi)$ is the *least* solution of $F \;\equiv\; \Psi \,\vee\, (\Phi \,\wedge\, \exists\bigcirc F)$. Using a set-theoretical counterpart by means of $Sat(\cdot)$, we obtain the following equivalent formulation of constraint (f) in Theorem 6.23:

(f′) $Sat(\exists(\Phi \,\mathsf{U}\, \Psi))$ is the smallest set $T \subseteq S$ satisfying

$$Sat(\Psi) \;\cup\; \{\, s \in Sat(\Phi) \mid Post(s) \cap T \neq \varnothing \,\} \;\subseteq\; T.$$

In fact, "$\subseteq$" may be replaced by "$=$".

In a similar way, $\exists\Box\, \Phi$ can be considered as the *greatest* fixed point of the logical equation

$$F \;=\; \Phi \,\wedge\, \exists\bigcirc F.$$

Using a set-theoretical counterpart of this equation we obtain the following equivalent formulation of constraint (g) in Theorem 6.23:

(g′) $Sat(\exists\Box\Phi)$ is the largest set $T \subseteq S$ satisfying

$$T \;\subseteq\; \{\, s \in Sat(\Phi) \mid Post(s) \cap T \neq \varnothing \,\}.$$

Also in this characterization "$\subseteq$" may be replaced by "$=$". ■

Characterizations of the satisfaction sets for universally quantified CTL formulae can be obtained using the result in Theorem 6.23. This yields

(h) $Sat(\forall\bigcirc \Phi) \;=\; \{\, s \in S \mid Post(s) \subseteq Sat(\Phi) \,\}.$

(i) $Sat(\forall(\Phi \,\mathsf{U}\, \Psi))$ is the smallest set $T \subseteq S$ satisfying

$$Sat(\Psi) \;\cup\; \{\, s \in Sat(\Phi) \mid Post(s) \subseteq T \,\} \;\subseteq\; T.$$

(j) $Sat(\forall\Box\, \Phi)$ is the largest set $T \subseteq S$ satisfying

$$T \;\subseteq\; \{\, s \in Sat(\Phi) \mid Post(s) \subseteq T \,\}.$$

Remark 6.25. Weak Until

The weak-until operator satisfies the same expansion laws as the until operator.

$$\begin{aligned}
\exists(\Phi \,\mathsf{W}\, \Psi) &\;\equiv\; \Psi \;\vee\; (\Phi \;\wedge\; \exists\bigcirc \exists(\Phi \,\mathsf{W}\, \Psi)),\\
\forall(\Phi \,\mathsf{W}\, \Psi) &\;\equiv\; \Psi \;\vee\; (\Psi \;\wedge\; \forall\bigcirc \forall(\Phi \,\mathsf{W}\, \Psi)).
\end{aligned}$$

The difference, however, is that the weak-until operator represents the largest solution (i.e., fixed point) of the expansion law, whereas the until operator denotes the smallest solution. The satisfaction sets for weak until are characterized as follows:

(k) $Sat(\exists(\Phi \,\mathsf{W}\, \Psi))$ is the largest set $T \subseteq S$ satisfying

$$T \;\subseteq\; Sat(\Psi) \;\cup\; \{\, s \in Sat(\Phi) \mid Post(s) \cap T \neq \varnothing \,\}.$$

(l) $Sat(\forall(\Phi \,\mathsf{W}\, \Psi))$ is the largest set $T \subseteq S$ satisfying

$$T \;\subseteq\; Sat(\Psi) \;\cup\; \{\, s \in Sat(\Phi) \mid Post(s) \subseteq T \,\}.$$

■

Without loss of generality, we may assume that the CTL formula Φ to be verified is in ENF (see Theorem 6.14, page 332). That is, the model-checking algorithm is supposed to be preceded by transforming the CTL formula at hand into ENF. The characterizations of the satisfaction sets indicated in Theorem 6.23 are exploited to compute the sets $Sat(\cdot)$. The essential steps for computing the satisfaction sets are summarized in Algorithm 14 on page 348.

6.4.2 The Until and Existential Always Operator

To treat constrained reachability properties, given by CTL formulae of the form $\exists(\Phi \,\mathsf{U}\, \Psi)$, the characterization in Theorem 6.23 is exploited. Recall that $Sat(\exists(\Phi \,\mathsf{U}\, \Psi))$ is character-

Algorithm 14 Computation of the satisfaction sets

Input: finite transition system TS with state set S and CTL formula Φ in ENF
Output: $Sat(\Phi) = \{ s \in S \mid s \models \Phi \}$

(* recursive computation of the sets $Sat(\Psi)$ for all subformulae Ψ of Φ *)

switch(Φ):

true	:	**return** S;
a	:	**return** $\{ s \in S \mid a \in L(s) \}$;
$\Phi_1 \wedge \Phi_2$	:	**return** $Sat(\Phi_1) \cap Sat(\Phi_2)$;
$\neg\Psi$	:	**return** $S \setminus Sat(\Psi)$;
$\exists \bigcirc \Psi$	:	**return** $\{ s \in S \mid Post(s) \cap Sat(\Psi) \neq \varnothing \}$;
$\exists(\Phi_1 \,\mathsf{U}\, \Phi_2)$	:	$T := Sat(\Phi_2)$; (* compute the smallest fixed point *) **while** $\{ s \in Sat(\Phi_1) \setminus T \mid Post(s) \cap T \neq \varnothing \} \neq \varnothing$ **do** **let** $s \in \{ s \in Sat(\Phi_1) \setminus T \mid Post(s) \cap T \neq \varnothing \}$; $T := T \cup \{ s \}$; **od**; **return** T;
$\exists \Box \Phi$	:	$T := Sat(\Phi)$; (* compute the greatest fixed point *) **while** $\{ s \in T \mid Post(s) \cap T = \varnothing \} \neq \varnothing$ **do** **let** $s \in \{ s \in T \mid Post(s) \cap T = \varnothing \}$; $T := T \setminus \{ s \}$; **od**; **return** T;

end switch

ized as the smallest set $T \subseteq S$, where S is the set of states in the transition system under consideration, such that

$$(1)\ Sat(\Psi) \subseteq T \text{ and } (2)\ (s \in Sat(\Phi) \text{ and } Post(s) \cap T \neq \varnothing) \ \Rightarrow\ s \in T. \tag{6.1}$$

This characterization suggests adopting the following iterative procedure to compute $Sat(\exists(\Phi \,\mathsf{U}\, \Psi))$:

$$T_0 \ =\ Sat(\Psi) \quad \text{and} \quad T_{i+1} \ =\ T_i \ \cup\ \{ s \in Sat(\Phi) \mid Post(s) \cap T_i \neq \varnothing \}.$$

Intuitively speaking, the set T_i contains all states that can reach a Ψ-state in at most i steps via a Φ-path. This is to be understood as follows. From the fact that $Sat(\exists(\Phi \,\mathsf{U}\, \Psi))$ satisfies the conditions (1) and (2) in (6.1), it can be demonstrated by induction on j that

$$T_0 \subseteq T_1 \subseteq T_2 \subseteq \ldots \subseteq T_j \subseteq T_{j+1} \subseteq \ \ldots \ \subseteq\ Sat(\exists(\Phi \,\mathsf{U}\, \Psi)).$$

Since we assume a finite transition system TS, there exists a $j \geqslant 0$ such that

$$T_j \ =\ T_{j+1} \ =\ T_{j+2} \ =\ \ldots \quad .$$

Therefore

$$T_j \;=\; T_j \;\cup\; \{\, s \in \mathit{Sat}(\Phi) \mid \mathit{Post}(s) \cap T_j \neq \varnothing \,\}$$

and, hence:

$$\{\, s \in \mathit{Sat}(\Phi) \mid \mathit{Post}(s) \cap T_j \neq \varnothing \,\} \;\subseteq\; T_j.$$

Hence, T_j satisfies property (2). Further,

$$\mathit{Sat}(\Psi) \;=\; T_0 \;\subseteq\; T_j.$$

These considerations show that T_j possesses the properties (1) and (2). Since $\mathit{Sat}(\exists(\Phi \,\mathsf{U}\, \Psi))$ is the *smallest* set of states satisfying the properties (1) and (2), it follows that

$$\mathit{Sat}(\exists(\Phi \,\mathsf{U}\, \Psi)) \;\subseteq\; T_j$$

and thus $\mathit{Sat}(\exists(\Phi \,\mathsf{U}\, \Psi)) = T_j$. Hence, for any $j \geqslant 0$ we have

$$T_0 \subsetneqq T_1 \subsetneqq T_2 \subsetneqq \ldots \subsetneqq T_j = T_{j+1} = \ldots = \mathit{Sat}(\exists(\Phi \,\mathsf{U}\, \Psi)).$$

Algorithm 15 (see page 351) shows a more detailed version of the backward search indicated earlier in Algorithm 14 (see page 348). As each Ψ-state obviously satisfies $\exists(\Phi \,\mathsf{U}\, \Psi)$, all states in $\mathit{Sat}(\Psi)$ are initially considered to satisfy $\exists(\Phi \,\mathsf{U}\, \Psi)$). This conforms to the initialization to the variable E. An iterative procedure is subsequently started that can be considered to systematically check the state space in a "backward" manner. In each iteration, all Φ-states are determined that can move by a single transition to (one of) the states of which we already know to satisfy $\exists(\Phi \,\mathsf{U}\, \Psi)$). Thus, in the ith iteration of the procedure, all Φ-states are considered that can move to a Ψ-state in at most i steps. This corresponds to the set T_i. Termination of the algorithm intuitively follows, as the number of states in the transition system is finite. Note that the algorithm assumes a transition system representation (or its state graph) by means of "inverse" adjacency lists, i.e., list representations for the sets of predecessors $\mathit{Pre}(s') \;=\; \{\, s \in S \mid s' \in \mathit{Post}(s) \,\}$.

Example 6.26.

Consider the transition system depicted in Figure 6.11, and suppose we are interested in checking the formula $\exists\Diamond\, \Phi$ with $\Phi = ((a = c) \wedge (a \neq b))$. Recall that $\exists\Diamond\, \Phi = \exists(\mathrm{true}\,\mathsf{U}\,\Phi)$. To check $\exists\Diamond\, \Phi$ we invoke Algorithm 14 (see page 348). This algorithm recursively computes $\mathit{Sat}(\mathrm{true})$ and $\mathit{Sat}((a = c) \wedge (a \neq b))$. This corresponds to the situation depicted in Figure 6.12(a), where all states in the set T are colored black, and white otherwise. In the first iteration, we select and delete s_5 from E, but as $\mathit{Pre}(s_5) = \varnothing$, T remains unaffected. On considering $s_4 \in E$, $\mathit{Pre}(s_4) = \{\, s_6 \,\}$ is added to T (and E), see Figure 6.12(b). During the next iteration, the only predecessor of s_6 is added, yielding the snapshot in Figure 6.12(c). After the fourth iteration, the algorithm terminates as there are no new predecessors of Φ-states encountered, i.e., $E = \varnothing$ (see Figure 6.12(d)). ■

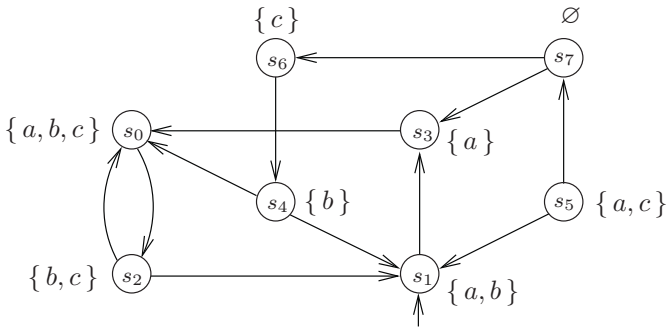


Figure 6.11: An example of a transition system.

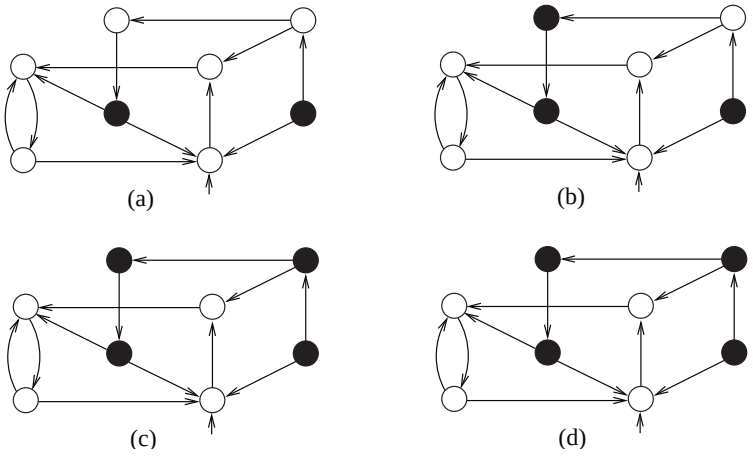


Figure 6.12: Example of backward search for $\exists(\text{true}\,\mathsf{U}\,(a{=}c) \wedge (a{\neq}b))$.

Algorithm 15 Enumerative backward search for computing $Sat(\exists(\Phi \,\mathsf{U}\, \Psi))$

Input: finite transition system TS with state set S and CTL formula $\exists(\Phi \,\mathsf{U}\, \Psi)$
Output: $Sat(\exists(\Phi \,\mathsf{U}\, \Psi)) = \{\, s \in S \mid s \models \exists(\Phi \,\mathsf{U}\, \Psi) \,\}$

```
E := Sat(Ψ);                          (* E administers the states s with s ⊨ ∃(Φ U Ψ) *)
T := E;                               (* T contains the already visited states s with s ⊨ ∃(Φ U Ψ) *)
while E ≠ ∅ do
  let s' ∈ E;
  E := E \ { s' };
  for all s ∈ Pre(s') do
    if s ∈ Sat(Φ) \ T then E := E ∪ { s }; T := T ∪ { s } fi
  od
od
return T
```

Let us now consider the computation of $Sat(\exists\Box\,\Phi)$ for the transition system TS. As for the until-operator, the algorithm for $\exists\Box\,\Phi$ is based on the characterization in Theorem 6.23, i.e., $Sat(\exists\Box\,\Phi)$ is the largest set $T \subseteq S$ satisfying

$$T \subseteq Sat(\Phi) \quad \text{and} \quad (s \in T \text{ implies } T \cap Post(s) \neq \varnothing).$$

The basic idea is to compute $Sat(\exists\Box\,\Phi)$ by means of the iteration

$$T_0 \;=\; Sat(\Phi) \quad \text{and} \quad T_{i+1} \;=\; T_i \;\cap\; \{\, s \in Sat(\Phi) \mid Post(s) \cap T_i \neq \varnothing \,\}.$$

Then, for all $j \geqslant 0$, it holds that

$$T_0 \supsetneqq T_1 \supsetneqq T_2 \supsetneqq \ldots \supsetneqq T_j = T_{j+1} = \ldots = T = Sat(\exists\Box\,\Phi).$$

The above iteration can be realized by means of a *backward search* starting with

$$T = Sat(\Phi) \quad \text{and} \quad E = S \setminus Sat(\Phi).$$

Here T equals T_0 and E contains all states that refute $\exists\Box\,\Phi$. During the backward search, states are iteratively removed from T, for which it has been established that they refute $\exists\Box\,\Phi$. This applies to any $s \in T$ satisfying

$$Post(s) \cap T \;=\; \varnothing.$$

Although $s \models \Phi$ (as it is in T), all its successors refute $\exists\Box\,\Phi$ (as they are not in T), and therefore s refutes $\exists\Box\,\Phi$. Once such states are encountered, they are inserted in E to enable the possible removal of other states in T.

Algorithm 16 Enumerative backward search to compute $Sat(\exists\Box\,\Phi)$

Input: finite transition system TS with state set S and CTL formula $\exists\Box\,\Phi$
Output: $Sat(\exists\Box\,\Phi) = \{\, s \in S \mid s \models \exists\Box\,\Phi \,\}$

```
E := S \ Sat(Φ);                          (* E contains any not visited s' with s' ⊭ ∃□Φ *)
T := Sat(Φ);          (* T contains any s for which s ⊨ ∃□Φ has not yet been disproven *)
for all s ∈ Sat(Φ) do count[s] := | Post(s) |; od          (* initialize array count *)
while E ≠ ∅ do
                                   (* loop invariant: count[s] = | Post(s) ∩ (T ∪ E) | *)
  let s' ∈ E;                                                          (* s' ⊭ Φ *)
  E := E \ { s' };                                   (* s' has been considered *)
  for all s ∈ Pre(s') do
                          (* update counters count[s] for all predecessors s of s' *)
    if s ∈ T then
      count[s] := count[s] − 1;
      if count[s] = 0 then
        T := T \ { s };                          (* s does not have any successor in T *)
        E := E ∪ { s };
      fi
    fi
  od
od
return T
```

The resulting computational procedure is detailed in Algorithm 16 on page 352. In order to support the test whether $Post(s) \cap T = \varnothing$, a counter $count[s]$ is exploited that keeps track of the number of direct successors of s in $T \cup E$:

$$count[s] \;=\; |\; Post(s) \;\cap\; (T \cup E)\;|.$$

Once $count[s] = 0$, we have that $Post(s) \cap (T \cup E) = \varnothing$, and hence $Post(s) \cap T = \varnothing$. Thus, state s is not in $Sat(\exists\Box\,\Phi)$ and therefore can be safely removed from T. On termination, $E = \varnothing$, and thus $count[s] = |\, Post(s) \cap T\,|$. It follows that any state $s \in Sat(\Phi)$ for which $count[s] > 0$ satisfies the CTL formula $\exists\Box\,\Phi$.

It is left to the interested reader to consider how the outlined approach can be modified to compute $Sat(\exists(\Phi\,\mathsf{W}\,\Psi))$.

Example 6.27.

Consider the transition system depicted in Figure 6.11 and the formula $\exists\Box\, b$. Initially,

$$T_0 = \{\, s_0, s_1, s_2, s_4 \,\} \quad \text{and} \quad E = \{\, s_3, s_5, s_6, s_7 \,\} \quad \text{and} \quad \mathit{count} = [1,1,2,1,2,2,1,2].$$

Suppose state $s_3 \in E$ is selected in the first iteration. As $s_1 \in \mathit{Pre}(s_3)$ and $s_1 \in T$, $\mathit{count}[s_1] := 0$. Accordingly, s_1 is deleted from T and added to E. This yields

$$T_1 = \{\, s_0, s_2, s_4 \,\} \quad \text{and} \quad E = \{\, s_1, s_5, s_6, s_7 \,\} \quad \text{and} \quad \mathit{count} = [1,0,2,1,2,2,1,2].$$

We also have $s_7 \in \mathit{Pre}(s_3)$, but as $s_7 \notin T$, this affects neither $\mathit{count}[s_7]$, T nor E.

Suppose s_6 and s_7 are selected in the second and third iteration, respectively. None of these states has a predecessor in T_1, so we obtain

$$T_3 = \{\, s_0, s_2, s_4 \,\} \quad \text{and} \quad E = \{\, s_1, s_5 \,\} \quad \text{and} \quad \mathit{count} = [1,0,2,1,2,2,1,2].$$

Now select $s_1 \in E$ in the next iteration. As $\mathit{Pre}(s_1) \cap T_3 = \{\, s_2, s_4 \,\}$, the counters for s_2 and s_4 are decremented. This yields

$$T_4 = \{\, s_0, s_2, s_4 \,\} \quad \text{and} \quad E = \{\, s_5 \,\} \quad \text{and} \quad \mathit{count} = [1,0,1,1,1,2,1,2].$$

As $\mathit{Pre}(s_5) = \varnothing$, T and *count* are unaffected in the subsequent iteration. As $E = \varnothing$, the algorithm terminates and returns $T = \{\, s_0, s_2, s_4 \,\}$ as the final outcome. ■

This section is concluded by outlining an alternative algorithm for computing $\mathit{Sat}(\exists\Box\,\Phi)$. Since the computation of the satisfaction sets takes place by means of a bottom-up traversal through the parse tree of the formula at hand, it is assumed that $\mathit{Sat}(\Phi)$ is at our disposal. A possibility to compute $\mathit{Sat}(\exists\Box\,\Phi)$ is to only consider the Φ-states of transition system TS and ignore all $\neg\Phi$-states. The justification of this modification to TS is that all removed states will not satisfy $\exists\Box\,\Phi$ (as they violate Φ) and therefore can be safely removed. For $TS = (S, \mathit{Act}, \rightarrow, I, \mathit{AP}, L)$ let $TS[\Phi] = (S', \mathit{Act}, \rightarrow', I', \mathit{AP}, L')$ with $S' = \mathit{Sat}(\Phi)$, $\rightarrow' \;=\; \rightarrow \,\cap\, (S' \times \mathit{Act} \times S')$, $I' = I \cap S'$ and $L'(s) = L(s)$ for all $s \in S'$. Then, all nontrivial strongly connected components (SCCs) [4] in the state graph induced by $TS[\Phi]$ are computed. All states in each such SCC C satisfy $\exists\Box\,\Phi$, as any state in C is reachable from any other state in C, and—by construction—all states in C satisfy Φ. Finally, all states in $TS[\Phi]$ are computed that can reach such SCC. If state $s \in S'$ and there exists such a path, then—by construction of $TS[\Phi]$—the property $\exists\Box\,\Phi$ is satisfied by s; otherwise, it is not. This can be done by a backward search. The worst-case time complexity of this

[4] A strongly connected component (SCC) of a digraph G is a maximal, connected subgraph of G. Stated differently, the SCCs of a graph are the equivalence classes of vertices under the "are mutually reachable" relation. A *nontrivial* SCC is an SCC that contains at least one transition.

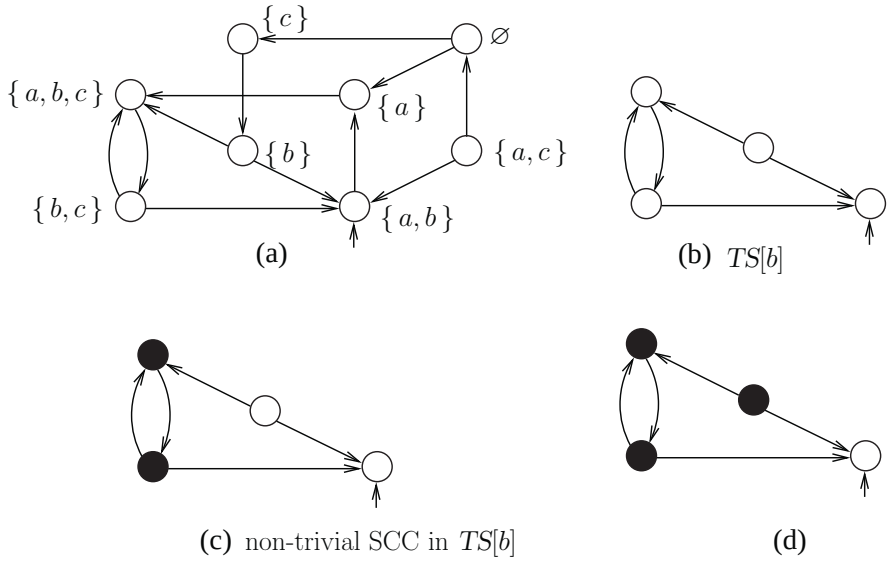


Figure 6.13: Computing $Sat(\exists\Box\, b)$ using strongly connected components in $TS[\Phi]$.

alternative algorithm is the same as for Algorithm 16. This approach is illustrated by the following example.

Example 6.28. Alternative Algorithm for $\exists\Box\,\Phi$

Consider the transition system of Figure 6.11 and CTL formula $\exists\Box\, b$. The modified transition system $TS[b]$ consists of the four states that are labeled with b, see the gray states in Figure 6.13(a). The only nontrivial SCC of this structure is indicated by the black states, see Figure 6.13(b). As there is only a single b-state (that is not in the SCC) that can reach the nontrivial SCC, this state satisfies $\exists\Box\, b$, and the computation finishes, see Figure 6.13(c). ■

Theorem 6.29.

For state s in transition system TS and CTL formula Φ:

$$s \models \exists\Box\,\Phi \text{ iff } s \models \Phi \text{ and there is a nontrivial SCC in } TS[\Phi] \text{ reachable from } s.$$

Proof: $\Rightarrow$: Suppose $s \models \exists\Box\,\Phi$. Clearly, s is a state in $TS[\Phi]$. Let π be a path in TS starting in s such that $\pi \models \Box\,\Phi$. As TS is finite, π has a suffix $\rho = s_1\, s_2 \ldots s_k$ for $k > 1$, representing a cycle that is traversed infinitely often. As π is also a path in $TS[\Phi]$, the

states s_1 through s_k are all in $TS[\Phi]$. Since π is traversed infinitely often, it represents a cycle, and thus any pair of states s_i and s_j is mutually reachable. Stated differently, $\{ s_1, \ldots, s_k \}$ is either an SCC or contained in some SCC in $TS[\Phi]$. As π is a path starting in s, these states are reachable from s.

$\Leftarrow$: Suppose s is a state in $TS[\Phi]$ and there exists an SCC in $TS[\Phi]$ reachable from s. Let s' be a state in such SCC. As the SCC is nontrivial, s' is reachable from itself by a path of length at least one. Repeating this cycle infinitely often yields an infinite path π with $\pi \models \Box\, \Phi$. The path from s to s' followed by $\pi[1..]$ now satisfies $\Box\, \Phi$ and starts in s. Thus, $s \models \exists \Box\, \Phi$. ■

6.4.3 Time and Space Complexity

The time complexity of the CTL model-checking algorithm is determined as follows. Let TS be a finite transition system with N states and K transitions. Under the assumption that the sets of predecessors $Pre(\cdot)$ are represented as linked lists, the time complexity of Algorithms 15 and 16 lies in $\mathcal{O}(N+K)$. Given that the computation of the satisfaction sets $Sat(\Phi)$ is a bottom-up traversal over the parse tree of Φ and thus linear in $|\, \Phi \,|$, the time complexity of Algorithm 14 (see page 348) is in

$$\mathcal{O}((N+K) \cdot |\, \Phi \,|).$$

When the initial states of TS are administrated in, e.g., a linked list, checking whether $I \subseteq Sat(\Phi)$ can be done in $\mathcal{O}(N_0)$ where N_0 is the cardinality of the set I. Recall that the CTL model-checking algorithm requires the CTL formula to be checked to be in existential normal form. As the transformation of any CTL formula into an equivalent CTL formula in ENF may yield an exponential blowup of the formula, it is recommended to treat the modalities such as $\forall \mathsf{U}$, $\forall \Diamond$, $\forall \Box$, $\exists \Diamond$, $\forall \mathsf{W}$, and $\exists \mathsf{W}$, analogous to the introduced approaches for $\exists \mathsf{U}$ and $\exists \Box$, by exploiting the characterizations of $Sat(\forall \Box\, \Phi)$, $Sat(\exists \Diamond\, \Phi)$, and so on. The resulting algorithms are all linear in N and K. Thus, we obtain the following theorem:

Theorem 6.30. Time Complexity of CTL Model Checking

For transition system TS with N states and K transitions, and CTL formula Φ, the CTL model-checking problem $TS \models \Phi$ can be determined in time $\mathcal{O}\,((N+K) \cdot |\, \Phi \,|)$.

Let us compare this complexity bound with that for LTL model checking. Recall that LTL model checking is exponential in the size of the formula. Although the difference in time complexity with respect to the length of the formula seems drastic (exponential for LTL vs.

linear for CTL), this should *not* be interpreted as "CTL model checking is more efficient than LTL model checking". From Theorem 6.18 it follows that whenever $\varphi \equiv \Phi$ for LTL formula φ and CTL formula Φ, then φ is obtained by removing all path quantifiers in Φ. Thus, CTL formulae are at least as long as their equivalent counterparts in LTL (if these exist). In fact, if P $\neq$ NP, then there exist LTL formulae φ_n with $|\,\varphi_n\,| \in \mathcal{O}(poly(n))$ for which CTL-equivalent formulae do exist, but not of polynomial length. LTL formulae may be exponentially shorter than any of their equivalent formulation in CTL. The latter effect is illustrated in the following example. From Lemma 5.45, it follows that the Hamiltonian path problem for digraphs with n nodes can be encoded in LTL formula φ_n whose length is polynomial in n. More precisely, given a digraph G with nodeset $\{\,1, \ldots, n\,\}$ there is an LTL formula φ_n such that (1) G has a Hamiltonian path if and only if $\neg\varphi_n$ does not hold for the transition system associated with G and (2) $\neg\varphi_n$ has an equivalent CTL formula.

To express the existence of a Hamiltonian path in CTL requires a CTL formula of exponential length, unless P = NP: if it is assumed that $\neg\varphi_n \equiv \neg\Phi_n$ for CTL formulae Φ_n of polynomial length, then the Hamiltonian path problem could be solved by CTL model checking, and thus in polynomial time. Since, however, the Hamiltonian path problem is NP-complete, this is only possible for the case of PTIME = NP.

Example 6.31. The Hamiltonian Path Problem

Consider the NP-complete problem of finding a Hamiltonian path in an arbitrary, connected, directed graph $G = (V, E)$ where V denotes the set of vertices and $E \subseteq V \times V$, the set of edges. Let $V = \{\,v_1, \ldots, v_n\,\}$. A Hamiltonian path is a (finite) path through the graph G which visits each state exactly once. Starting from graph G, a transition system $TS(G)$ is derived as well as a CTL formula Φ_n, such that

$$G \text{ contains a Hamiltonian path if and only if } TS \not\models \neg\Phi_n.$$

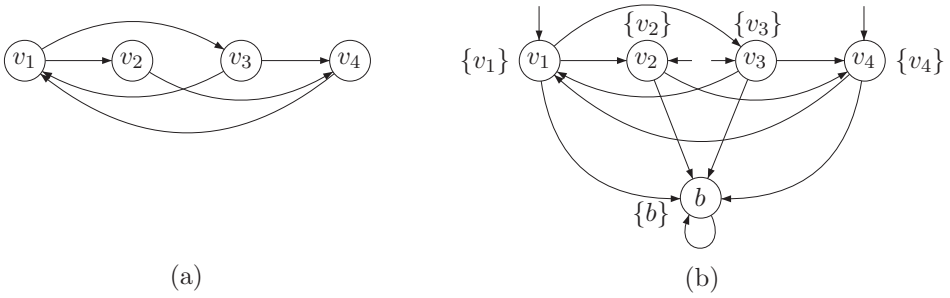
The transition system TS is defined as follows

$$TS \;=\; (V \cup \{\,b\,\}, \{\,\tau\,\}, \rightarrow, V, V, L)$$

with $L(v_i) = \{\,v_i\,\}$, $L(b) = \varnothing$. The transition relation $\rightarrow$ is defined by

$$\frac{(v_i, v_j) \in E}{v_i \xrightarrow{\tau} v_j} \quad \text{and} \quad \frac{v_i \in V \cup \{\,b\,\}}{v_i \xrightarrow{\tau} b}.$$

All vertices of G are states in the transition system TS. A new state b is introduced that is a direct successor of any state (including b). Figure 6.14 shows (a) an example of a directed graph G and (b) its transition system $TS(G)$. The sole purpose of the new state b is to ensure that the transition system has no terminal states. Note that $TS(G)$ is defined as in the proof for the existence of a polynomial reduction of the Hamiltonian



(a) (b)

Figure 6.14: Example of encoding the Hamiltonian path problem as a transition system.

path problem onto the complement of the LTL model-checking problem; see Lemma 5.45 on page 288.

It remains to give a recipe for the construction of Φ_n, a CTL formula that captures the existence of a Hamiltonian path. Let Φ_n be defined as follows:

$$\Phi_n \;=\; \bigvee_{\substack{(i_1,\ldots,i_n)\\ \text{permutation of } (1,\ldots,n)}} \Psi(v_{i_1}, v_{i_2}, \ldots, v_{i_n})$$

such that $\Psi(v_{i_1}, \ldots, v_{i_n})$ is a CTL formula that is satisfied if and only if $v_{i_1}, v_{i_2}, \ldots, v_{i_n}$ is a Hamiltonian path in G. The formulae $\Psi(v_{i_1}, \ldots, v_{i_n})$ are inductively defined as follows:

$$\begin{aligned} \Psi(v_i) &= v_i \\ \Psi(v_{i_1}, v_{i_2}, \ldots, v_{i_n}) &= v_{i_1} \wedge \exists \bigcirc \Psi(v_{i_2}, \ldots, v_{i_n}) \quad \text{if } n > 1. \end{aligned}$$

Stated in words, $\Psi(v_{i_1}, \ldots, v_{i_n})$ holds if there exists a path on which v_{i_1}, v_{i_2}, v_{i_3} successively hold. Since each state has a transition to the b state, the trace $\{ v_{i_1} \}\{ v_{i_2} \}\ldots\{ v_{i_n} \}$ can be extended with $\{ b \}^\omega$. An example of an instantiation of Ψ_n is

$$\Phi_2 \;=\; (v_1 \wedge \exists\bigcirc v_2) \;\vee\; (v_2 \wedge \exists\bigcirc v_1)$$

and

$$\begin{aligned} \Phi_3 \;=\; & (v_1 \wedge \exists \bigcirc (v_2 \wedge \exists\bigcirc v_3)) \;\vee\; (v_1 \wedge \exists \bigcirc (v_3 \wedge \exists\bigcirc v_2)) \\ \vee\; & (v_2 \wedge \exists \bigcirc (v_1 \wedge \exists\bigcirc v_3)) \;\vee\; (v_2 \wedge \exists \bigcirc (v_3 \wedge \exists\bigcirc v_1)) \\ \vee\; & (v_3 \wedge \exists \bigcirc (v_1 \wedge \exists\bigcirc v_2)) \;\vee\; (v_3 \wedge \exists \bigcirc (v_2 \wedge \exists\bigcirc v_1)). \end{aligned}$$

It is not difficult to infer that

$$\mathit{Sat}(\; \Psi(v_{i_1}, \ldots, v_{i_n}) \;) \;=\; \begin{cases} \{ v_{i_1} \} & \text{if } v_{i_1}, \ldots, v_{i_n} \text{ is a Hamiltonian path in } G \\ \varnothing & \text{otherwise.} \end{cases}$$

Thus:

$TS \not\models \neg\Phi_n$

iff there is an initial state s of TS for which $s \not\models \neg\Phi_n$

iff there is an initial state s of TS for which $s \models \Phi_n$

iff $\exists v$ in G and a permutation $i_1, \ldots, i_n$ of $1, \ldots, n$ with $v \in Sat(\Psi(v_{i_1}, \ldots, v_{i_n}))$,

iff $\exists v$ in G and a permutation $i_1, \ldots, i_n$ of $1, \ldots, n$, such that $v = v_{i_1}$ and $v_{i_1}, \ldots, v_{i_n}$ is a Hamiltonian path in G

iff G has a Hamiltonian path.

Thus, G contains a Hamiltonian path if and only if $TS \not\models \neg\Phi_n$.

By the explicit enumeration of all possible permutations we obtain a formula with a length that is exponential in the number of vertices in the graph. This does not prove that there does not exist an equivalent, but shorter, CTL formula which describes the Hamiltonian path problem. Actually, shorter formalizations in CTL cannot be expected, since the CTL model-checking problem is polynomially solvable whereas the Hamiltonian path problem is NP-complete. ■

6.5 Fairness in CTL

Recall that fairness assumptions (see Section 3.5) are used to rule out certain computations that are considered to be unrealistic for the system under consideration. These unreasonable computations that ignore certain transition alternatives forever are the "unfair" ones; the remaining computations are thus the "fair" ones. As there are different notions of fairness, various distinct forms of fairness can be imposed: unconditional, strong, and weak fairness constraints.

An important distinction between LTL and CTL is that fairness assumptions can be incorporated into LTL without any specific changes while a special treatment of fairness is required for CTL. That is to say, fairness assumptions can be added as premise to the LTL formula to be verified. The LTL model-checking problem where only fair paths are considered (i.e., considering the fair satisfaction relation $\models_{\mathit{fair}}$) can thus be reduced to the traditional LTL model-checking problem, i.e., with respect to the common satisfaction relation $\models$. In this case, the LTL formula φ to be verified is replaced by $\mathit{fair} \rightarrow \varphi$:

$$TS \models_{\mathit{fair}} \varphi \quad \text{if and only if} \quad TS \models (\mathit{fair} \rightarrow \varphi).^5$$

[5] This observation is mainly of theoretical interest since it is more efficient to design special LTL model-

For more details we refer to Section 5.1.6 (see page 257).

An analogous approach is *not* possible for CTL. This stems from the fact that most fairness constraints cannot be encoded in a CTL formula. An indication of this is the fact that persistence properties $\Diamond\Box a$ are inherent in, e.g., strong fairness conditions $\Box\Diamond b \rightarrow \Box\Diamond c \equiv \Diamond\Box\neg b \vee \Box\Diamond c$ and cannot be expressed in CTL. To be a bit more precise: fairness constraints operate on the path level and replace the standard meaning "for all paths" of universal quantification with "for all fair paths" and existential quantification "there exists a path" with "there exists a fair path". Thus, if *fair* expresses the fairness condition on the path level, then CTL formulae that encode the intuitive meaning of $\forall(\mathit{fair} \rightarrow \varphi)$ and $\exists(\mathit{fair} \wedge \varphi)$ would be needed. However, these are not legal CTL formulae since (1) the Boolean connectives $\rightarrow$ and $\wedge$ are not allowed on the level of CTL path formulae and (2) fairness conditions cannot be described by CTL path formulae.

Therefore, an alternative approach is taken to treat fairness in CTL. In order to deal with fairness constraints in CTL, the semantics of CTL is slightly modified such that the state formulae $\forall\varphi$ and $\exists\varphi$ are interpreted over all fair paths rather than over all possible paths. A fair path is a path that satisfies a set of fairness constraints. It is assumed that the given fixed fairness constraint is described by a formula as in LTL.

Definition 6.32. CTL Fairness Assumptions

A *strong CTL fairness constraint* (over AP) is a term of the form

$$\mathit{sfair} = \bigwedge_{1 \leqslant i \leqslant k} (\Box \Diamond \Phi_i \rightarrow \Box \Diamond \Psi_i)$$

where Φ_i and Ψ_i (for $1 \leqslant i \leqslant k$) are CTL formulae over AP. Weak and unconditional CTL fairness constraints are defined analogously by conjunctions of terms of the form $(\Diamond \Box \Phi_i \rightarrow \Box \Diamond \Psi_i)$ and $\Box \Diamond \Psi_i$, respectively. A *CTL fairness assumption* is a conjunction of strong, weak, and unconditional CTL fairness constraints. ■

Note that CTL fairness assumptions are *not* CTL path formulae, but they can be viewed as LTL formulae using CTL state formulae instead of atomic propositions. For instance, imposing the strong fairness constraint $\bigwedge_{1 \leqslant i \leqslant k} (\Box \Diamond \Phi_i \rightarrow \Box \Diamond \Psi_i)$ on paths means that a path must either have only finitely many states satisfying Φ_i or infinitely many states satisfying Ψ_i, for any $1 \leqslant i \leqslant k$ (or both).

Let $TS = (S, \mathit{Act}, \rightarrow, I, AP, L)$ be a transition system without terminal states, π be an infinite path fragment in TS, and *fair* a fixed CTL fairness assumption. $\pi \models \mathit{fair}$ denotes

checking algorithms when dealing with fairness assumptions than to apply the reduction to the standard semantics $\models$.

that π satisfies the formula *fair*, where $\models$ should be read as the LTL semantics. Consider, for example, strong fairness. For an infinite path $\pi = s_0\, s_1\, s_2 \ldots$, we have

$$\pi \models \bigwedge_{1 \leqslant i \leqslant k} (\Box \Diamond \Phi_i \rightarrow \Box \Diamond \Psi_i)$$

if and only if for every $i \in \{1, \ldots, k\}$ either $s_j \models \Phi_i$ for finitely many indices j, or $s_j \models \Psi_i$ for infinitely many j (or both). Here, the statement $s_j \models \Phi_i$ should be interpreted according to the CTL semantics, that is, the semantics without taking any fairness into account.

The semantics of CTL under fairness assumption *fair* is identical to the semantics given earlier (see Definition 6.4), except that the path quantifications range over all fair paths rather than over all paths. The fair paths starting in state s are defined as

$$\mathit{FairPaths}(s) \;=\; \{\pi \in \mathit{Paths}(s) \mid \pi \models \mathit{fair}\}.$$

Let $\mathit{FairPaths}(TS)$ denote the set of all fair paths in TS, i.e.:

$$\mathit{FairPaths}(TS) \;=\; \bigcup_{s_0 \in I} \mathit{FairPaths}(s_0).$$

The fair interpretation of CTL is defined in terms of the satisfaction relation $\models_{\mathit{fair}}$. We have $s \models_{\mathit{fair}} \Phi$ if and only if Φ is valid in state s under the fairness assumption *fair*.

Definition 6.33. Satisfaction Relation for CTL with Fairness

Let $TS = (S, \mathit{Act}, \rightarrow, I, \mathit{AP}, L)$ be a transition system without terminal states and $s \in S$. The satisfaction relation $\models_{\mathit{fair}}$ for CTL fairness assumption *fair* is defined for state formulae by

$$\begin{array}{lll}
s \models_{\mathit{fair}} a & \text{iff} & a \in L(s) \\
s \models_{\mathit{fair}} \neg \Phi & \text{iff} & \text{not } s \models_{\mathit{fair}} \Phi \\
s \models_{\mathit{fair}} \Phi \wedge \Psi & \text{iff} & (s \models_{\mathit{fair}} \Phi) \text{ and } (s \models_{\mathit{fair}} \Psi) \\
s \models_{\mathit{fair}} \exists \varphi & \text{iff} & \pi \models_{\mathit{fair}} \varphi \text{ for some } \pi \in \mathit{FairPaths}(s) \\
s \models_{\mathit{fair}} \forall \varphi & \text{iff} & \pi \models_{\mathit{fair}} \varphi \text{ for all } \pi \in \mathit{FairPaths}(s)
\end{array}$$

Here, $a \in \mathit{AP}$, Φ, Ψ are CTL state formulae, and φ a CTL path formula. For path π, the satisfaction relation $\models_{\mathit{fair}}$ for path formulae is defined as in Definition 6.4:

$$\begin{array}{lll}
\pi \models_{\mathit{fair}} \bigcirc \Phi & \text{iff} & \pi[1] \models_{\mathit{fair}} \Phi \\
\pi \models_{\mathit{fair}} \Phi \,\mathsf{U}\, \Psi & \text{iff} & \exists\, j \geqslant 0.\ (\pi[j] \models_{\mathit{fair}} \Psi \ \wedge\ (\forall\, 0 \leqslant k < j.\, \pi[k] \models_{\mathit{fair}} \Phi))
\end{array}$$

where for path $\pi = s_0\, s_1\, s_2 \ldots$ and integer $i \geqslant 0$, $\pi[i]$ denotes the $(i{+}1)$-th state of π, i.e., $\pi[i] = s_i$. ■

Whereas for LTL, fairness constraints can be specified as part of the formula to be checked, for CTL similar constraints are imposed on the underlying model of the system under consideration, i.e., the transition system.

Definition 6.34. CTL Semantics with Fairness for Transition Systems

For CTL-state formula Φ, and CTL fairness assumption *fair*, the *satisfaction set* $Sat_{fair}(\Phi)$ is defined by

$$Sat_{fair}(\Phi) \;=\; \{\, s \in S \mid s \models_{fair} \Phi \,\}.$$

The transition system TS satisfies CTL formula Φ under fairness assumption *fair* if and only if Φ holds in all initial states of TS:

$$TS \models_{fair} \Phi \quad \text{if and only if} \quad \forall s_0 \in I.\, s_0 \models_{fair} \Phi.$$

This is equivalent to $I \subseteq Sat_{fair}(\Phi)$. ■

Example 6.35. CTL Fairness Assumption

Consider the transition system TS depicted in Figure 6.15 and suppose we are interested in establishing whether or not $TS \models \forall\Box\,(a \;\Rightarrow\; \forall\Diamond\, b)$. This formula is invalid since the path $s_0\, s_1\, (s_2\, s_4)^\omega$ never goes through a b-state. The reason that this property is not valid is as follows. At state s_2 there is a nondeterministic choice between moving either to state s_3 or to s_4. By continuously ignoring the possibility of going to s_3 we obtain a computation for which $\forall\Box\,(a \;\Rightarrow\; \forall\Diamond\, b)$ is invalid, hence:

$$TS \not\models \forall\Box\,(a \;\Rightarrow\; \forall\Diamond\, b).$$

Usually, though, the intuition is that if there is infinitely often a choice of moving to s_3, then s_3 should be visited in some fair way.

Let us impose the unconditional fairness assumption:

$$\mathit{fair} \;\equiv\; \Box\Diamond\, a \;\wedge\; \Box\Diamond\, b.$$

We now check $\forall\Box\,(a \;\Rightarrow\; \forall\Diamond\, b)$ under *fair*, i.e., consider the verification problem $TS \models_{fair} \forall\Box\,(a \;\Rightarrow\; \forall\Diamond\, b)$. Due to the fairness assumption, paths like $s_0\, s_1\, (s_2\, s_4)^\omega$ are excluded, since s_3 is never visited along this path. It can now easily be established that

$$TS \models_{fair} \forall\Box\,(a \;\Rightarrow\; \forall\Diamond\, b) \quad .$$

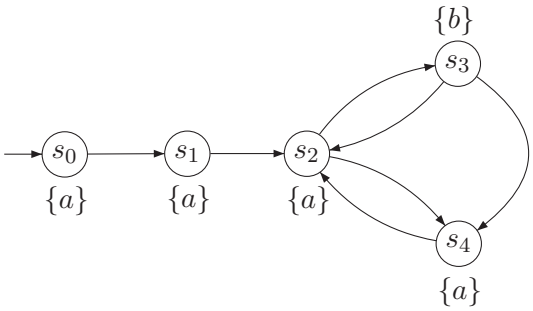


Figure 6.15: An example of a transition system.

■

Example 6.36. Mutual Exclusion

Consider the semaphore-based solution to the two-process mutual exclusion problem. The transition system of this concurrent program is denoted TS_{sem}. The CTL formula

$$\Phi \;=\; (\forall\Box\forall\Diamond\, crit_1) \;\wedge\; (\forall\Box\forall\Diamond\, crit_2)$$

describes the liveness property that both processes infinitely often have access to the critical section. It follows that $TS_{sem} \not\models \Phi$. We first impose the weak fairness assumption

$$wfair \;=\; (\Diamond\Box\, noncrit_1 \rightarrow \Box\Diamond\, wait_1) \wedge (\Diamond\Box\, noncrit_2 \rightarrow \Box\Diamond\, wait_2)$$

and the strong fairness assumption

$$sfair \;=\; (\Box\Diamond\, wait_1 \rightarrow \Box\Diamond\, crit_1) \wedge (\Box\Diamond\, wait_1 \rightarrow \Box\Diamond\, crit_1)\,.$$

It then follows that $TS_{sem} \models_{fair} \Phi$ where $fair = wfair \wedge sfair$.

As a second example, consider the arbiter-based solution to the two-process mutual exclusion problem, see Example 5.27 on page 259. The decision as to which process will acquire access to the critical section is determined by coin flipping by the arbiter. Consider the unconditional fairness assumption

$$ufair \;=\; \Box\Diamond\, head \;\wedge\; \Box\Diamond\, tail.$$

This fairness assumption can be considered to require a fair coin such that the events "heads" and "tails" occur infinitely often with probability 1. Apparently, it follows that

$$TS_1 \parallel Arbiter \parallel TS_2 \;\not\models\; \Phi, \quad \text{and} \quad TS_1 \parallel Arbiter \parallel TS_2 \;\models_{ufair}\; \Phi.$$

■

As explained before, fairness is treated in CTL by considering a fair satisfaction relation, denoted $\models_{fair}$ where *fair* is the fairness assumption considered. In the remainder of this section, algorithms will be provided in order to check whether

$$TS \models_{fair} \Phi$$

for CTL formula Φ and CTL-fairness assumption *fair*. As before, *TS* is supposed to be finite and have no terminal states and Φ is a CTL formula in ENF. Note that the assumption that Φ is in ENF does not impose any restriction as any CTL formula can be transformed into an equivalent (with respect to $\models_{fair}$) CTL formula in ENF. This can be established in the same way as Theorem 6.14 (page 332).

The basic idea is to exploit the CTL model-checking algorithms (without fairness) to compute $Sat_{fair}(\Phi) = \{ s \in S \mid s \models_{fair} \Phi \}$. Suppose *fair* is the strong CTL fairness constraint:

$$fair \;=\; \bigwedge_{0<i\leqslant k} (\Box \Diamond \Phi_i \rightarrow \Box \Diamond \Psi_i)$$

where Φ_i and Ψ_i are CTL formulae over *AP*. Recall that Φ_i and Ψ_i are interpreted according to the standard CTL semantics, i.e., without taking any fairness assumptions into account. By applying the CTL model-checking algorithm, first the sets $Sat(\Phi_i)$ and $Sat(\Psi_i)$ are determined. The formulae Φ_i and Ψ_i can thus be replaced by (fresh) atomic propositions a_i and b_i, say. It thus suffices to consider strong fairness assumptions of the form

$$fair \;=\; \bigwedge_{0<i\leqslant k} (\Box \Diamond a_i \rightarrow \Box \Diamond b_i)\,.$$

Once the fairness assumption is simplified, the sets $Sat_{fair}(\Psi)$ are determined for all subformulae Ψ of Φ using the standard CTL model-checking algorithm (i.e., without fairness), together with an algorithm to compute $Sat_{fair}(\exists\Box\, a)$ for $a \in AP$. The outcome of the model-checking procedure is "yes" if $I \subseteq Sat_{fair}(\Phi)$, and "no" otherwise.

The essential ideas are outlined in Algorithm 17 (page 364).

The subformulae of Φ are treated as in the CTL model-checking routine. Based on the syntax tree of Φ, a bottom-up computation is initiated. It is essential that during the computation of $Sat_{fair}(\Psi)$, the maximal genuine subformulae of Ψ have been already processed and replaced by atomic propositions. For the propositional logic fragment,

Algorithm 17 CTL model checking with fairness (basic idea)

Input: finite transition system TS, CTL formula Φ in ENF, and CTL fairness assumption *fair* over k CTL state formulae Φ_i and Ψ_i
Output: $TS \models_{fair} \Phi$

---

for all $0 < i \leqslant k$ **do**
  determine $Sat(\Phi_i)$ and $Sat(\Psi_i)$
  if $s \in Sat(\Phi_i)$ **then** $L(s) := L(s) \cup \{ a_i \}$; **fi**
  if $s \in Sat(\Psi_i)$ **then** $L(s) := L(s) \cup \{ b_i \}$; **fi**
od
compute $Sat_{fair}(\exists\Box\,\text{true}) = \{ s \in S \mid FairPaths(s) \neq \varnothing \}$;
forall $s \in Sat_{fair}(\exists\Box\,\text{true})$ **do** $L(s) := L(s) \cup \{ a_{fair}; \}$ **od**
(* compute $Sat_{fair}(\Phi)$ *)
for all $i \leqslant |\,\Phi\,|$ **do**
  for all $\Psi \in Sub(\Phi)$ with $|\,\Psi\,| = i$ **do**
    switch(Ψ):

true	:	$Sat_{fair}(\Psi) := S$;
a	:	$Sat_{fair}(\Psi) := \{ s \in S \mid a \in L(s) \}$;
$a \wedge a'$	:	$Sat_{fair}(\Psi) := \{ s \in S \mid a, a' \in L(s) \}$;
$\neg a$	:	$Sat_{fair}(\Psi) := \{ s \in S \mid a \notin L(s) \}$;
$\exists\bigcirc a$	:	$Sat_{fair}(\Psi) := Sat(\exists\bigcirc (a \wedge a_{fair}))$;
$\exists(a \,\mathsf{U}\, a')$	:	$Sat_{fair}(\Psi) := Sat(\exists(a \,\mathsf{U}\, (a' \wedge a_{fair})))$;
$\exists\Box\, a$	:	compute $Sat_{fair}(\exists\Box\, a)$

    end switch
    replace all occurrences of Ψ in Φ by the atomic proposition a_Ψ;
    forall $s \in Sat_{fair}(\Psi)$ **do** $L(s) := L(s) \cup \{ a_\Psi \}$ **od**
  od
od
return $I \subseteq Sat_{fair}(\Phi)$

---

the approach is straightforward:

$$\begin{array}{lcl} Sat_{fair}(\text{true}) & = & S \\ Sat_{fair}(a) & = & \{\, s \in S \mid a \in L(s) \,\} \\ Sat_{fair}(\neg a) & = & S \setminus Sat_{fair}(a) \\ Sat_{fair}(a \wedge a') & = & Sat_{fair}(a) \cap Sat_{fair}(a'). \end{array}$$

For all nodes of the syntax tree that are labeled with either $\exists\bigcirc$ or $\exists\,\mathsf{U}$ (i.e, nodes that represent a subformula of the form $\Psi = \exists\bigcirc a$ or of the form $\Psi = \exists(a\,\mathsf{U}\,a')$), the following observation is used. For any infinite path fragment π in TS, π is fair if and only if one (or all) suffix(es) of π is (are) fair:

$$\pi \models \mathit{fair} \quad \text{iff} \quad \pi[j..] \models \mathit{fair} \text{ for some } j \geqslant 0 \quad \text{iff} \quad \pi[j..] \models \mathit{fair} \text{ for all } j \geqslant 0.$$

The following two lemmas provide the ingredients for checking subformulae of the "type" $\exists\bigcirc$ and $\exists\,\mathsf{U}$.

Lemma 6.37. Next Step for Fair Satisfaction Relation

$s \models_{fair} \exists\bigcirc a$ if and only if $\exists s' \in Post(s)$ with $s' \models a$ and $FairPaths(s') \neq \varnothing$.

Proof: $\Rightarrow$: Assume $s \models_{fair} \exists\bigcirc a$. Then there exists a fair path $\pi = s_0\, s_1\, s_2\, s_3 \ldots \in Paths(s)$ with $s_1 \models a$. Since π is fair, the path $\pi[1..] = s_1\, s_2\, s_3 \ldots$ is fair too. Thus, $s' = s_1 \in Post(s)$ satisfies the indicated property.

$\Leftarrow$: Assume $s' \models a$ and $FairPaths(s') \neq \varnothing$ for some $s' \in Post(s)$. Thus there exists a fair path

$$\pi' \;=\; s'\, s'_1\, s'_2\, s'_3 \ldots$$

starting in s'. Therefore, $\pi \;=\; s\, s'\, s'_1\, s'_2\, s'_3 \ldots$ is a fair path starting s such that $\pi \models \bigcirc a$. Thus, $s \models_{fair} \exists\bigcirc a$. ∎

Using analogous arguments we obtain:

Lemma 6.38. Until for Fair Satisfaction Relation

$s \models_{fair} \exists(a_1\,\mathsf{U}\,a_2)$ if and only if there exists a finite path fragment

$$s_0\, s_1\, s_2\, s_3 \ldots s_n \in Paths_{fin}(s) \quad \text{with } n \geqslant 0$$

such that $s_i \models a_1$ for $0 \leqslant i < n$, $s_n \models a_2$, and $FairPaths(s_n) \neq \varnothing$.

The results of the previous two lemmas lead to the following approach. As a first step, the set

$$Sat_{fair}(\exists\Box\,\text{true}) \;=\; \{\, s \in S \mid FairPaths(s) \neq \varnothing \,\}$$

is computed. (The algorithmic way to do so is explained later in this chapter.) That is, all states are determined for which it is guaranteed that at least one fair path emanates. Once such a state is visited, it is thus guaranteed that a fair continuation is possible. The labels of the thus computed states are extended with the new atomic proposition a_{fair}, i.e.:

$$a_{fair} \in L(s) \quad \text{if and only if} \quad s \in Sat_{fair}(\exists\Box\,\text{true}).$$

The sets $Sat_{fair}(\exists\bigcirc a)$ and $Sat_{fair}(\exists(a\,\mathsf{U}\,a'))$ result from

$$\begin{aligned} Sat_{fair}(\exists\bigcirc a) &= Sat\left(\exists\bigcirc (a \wedge a_{fair})\right), \\ Sat_{fair}(\exists(a\,\mathsf{U}\,a')) &= Sat\left(\exists(a\,\mathsf{U}\,(a' \wedge a_{fair}))\right). \end{aligned}$$

As a result, these satisfaction sets can be computed using an ordinary CTL model checker. These considerations lead to Algorithm 17 on page 364 and provide the basis for the following result:

Theorem 6.39. Model-Checking CTL with Fairness

The model-checking problem for CTL with fairness can be reduced to

- *the model-checking problem for CTL (without fairness), and*
- *the problem of computing $Sat_{fair}(\exists\Box a)$ for the atomic proposition a.*

Note that the set $Sat_{fair}(\exists\Box\,\text{true})$ corresponds to $Sat_{fair}(\exists\Box\,a)$ whenever every state is labeled with a. Thus, an algorithm to compute $Sat_{fair}(\exists\Box\,a)$ can also be used to compute $Sat_{fair}(\exists\Box\,\text{true})$.

In the following, we explain how to compute the satisfaction set $Sat_{fair}(\exists\Box\,a)$ for $a \in AP$ in case *fair* is a *strong* CTL fairness assumption. Weak fairness assumptions can be treated in an analogous way. As we will see, unconditional fairness assumptions are a special case. Arbitrary fairness assumptions with unconditional, strong, and weak fairness constraints can be treated by using algorithms in which the corresponding techniques are appropriately combined.

Consider the strong CTL fairness assumption over the atomic propositions a_i and b_i ($0 < i \leqslant k$):

$$sfair \;=\; \bigwedge_{0<i\leqslant k} (\Box\Diamond a_i \rightarrow \Box\Diamond b_i).$$

The following lemma provides a graph-theoretical characterization of the fair satisfaction set $Sat_{sfair}(\exists\Box\,a)$ where a is an atomic proposition.

Lemma 6.40. Characterization of $Sat_{sfair}(\exists\Box a)$

$s \models_{sfair} \exists\Box\, a$ if and only if there exists a finite path fragment $s_0\, s_1 \ldots s_n$ and a cycle $s'_0\, s'_1 \ldots s'_r$ such that

(1) $s_0 = s \quad$ *and* $\quad s_n = s'_0 = s'_r$,

(2) $s_i \models a$, *for any* $0 \leqslant i \leqslant n$, *and* $s'_j \models a$, *for any* $0 < j \leqslant r$, *and*

(3) $Sat(a_i) \cap \{\, s'_1, \ldots, s'_r \,\} = \varnothing \;$ *or* $\; Sat(b_i) \cap \{\, s'_1, \ldots, s'_r \,\} \neq \varnothing$ *for all* $1 \leqslant i \leqslant k$.

Proof: $\Leftarrow$: Assume there exists a finite path fragment $s_0\, s_1 \ldots s_n$ and a cycle $s'_0\, s'_1 \ldots s'_r$ such that the conditions (1) through (3) hold for state s. Consider the infinite path fragment $\pi = s_0\, s_1 \ldots s_n\, s'_1 \ldots s'_r\, s'_1 \ldots s'_r \ldots \in Paths(s)$ which is obtained by traversing the cycle $s'_0\, s'_1 \ldots s'_r$ infinitely often. From constraints (1) and (3), it follows that π is fair, i.e., $\pi \models sfair$. From (2), it follows that $\pi \models \Box a$. Thus, $s \models_{sfair} \exists\Box\, a$.

$\Rightarrow$: Assume $s \models_{sfair} \exists\Box\, a$. Since $s \models_{sfair} \exists\Box\, a$, there exists an infinite path fragment $\pi = s_0\, s_1\, s_2 \ldots$ with $\pi \models \Box a$ and $\pi \models sfair$. Let $i \in \{\, 1, \ldots, k \,\}$. Distinguish between two cases:

1. $\pi \models \Box\Diamond a_i$. Since $\pi \models sfair$, there is a state $s' \in Sat(b_i)$ which is infinitely often visited in π . Let $n \in \mathbb{N}$ such that $s_n = s'$ and $s' \notin \{\, s_0, s_1, \ldots, s_{n-1} \,\}$. That is, s_n is the first occurrence of s' in π. Further, let $r > n$ such that $s_n = s_r$. Then $s_n\, s_{n+1} \ldots s_{r-1}\, s_r = s_n$ is a cycle with
$$s_n \in Sat(b_i) \cap \{\, s_n, s_{n+1}, \ldots, s_{r-1}, s_r \,\}.$$
2. $\pi \not\models \Box\Diamond a_i$. Then there exists an index n such that $s_n, s_{n+1}, s_{n+2} \ldots \notin Sat(a_i)$. Assume without loss of generality that s_n occurs infinitely often in π. Since there are finitely many states, there exists an $r > n$ such that $s_n = s_r$. Thus, $s_n\, s_{n+1} \ldots s_{r-1}\, s_r = s_n$ is a cycle with $Sat(a_i) \cap \{\, s_n, s_{n+1}, \ldots, s_{r-1}, s_r \,\} = \varnothing$.

From both cases it follows that there exists a finite path fragment $s_0\, s_1 \ldots s_n$ and a cycle $s'_0 \ldots s'_r$ satisfying the constraints (1) through (3). ■

This characterization is used to compute $Sat_{sfair}(\exists\Box\, a)$ in the following way. Consider the directed graph $G[a] = (S, E_a)$ whose set of edges E_a is defined as

$$(s, s') \in E_a \quad \text{if and only if} \quad s' \in Post(s) \;\wedge\; s \models a \;\wedge\; s' \models a.$$

Stated in words, $G[a]$ is obtained from the state graph G_{TS} by eliminating all edges (s, s') for which either $s \not\models a$ or $s' \not\models a$. (Thus, $G[a]$ is the state graph of the transition system $TS[a]$.) Apparently, each infinite path in $G[a]$ (that starts in $s \in I$) corresponds to an infinite path in the transition system TS satisfying $\Box a$. Conversely, each infinite path fragment π of TS with $\pi \models \Box a$ is a path in $G[a]$. In particular,

$$s \models_{sfair} \exists \Box a$$

if and only if there exists a nontrivial, strongly connected set of nodes D in $G[a]$ that is reachable from s, such that for all $1 \leqslant i \leqslant k$:

$$D \cap Sat(a_i) = \varnothing \qquad \text{or} \qquad D \cap Sat(b_i) \neq \varnothing.$$

Let T be the union of all nontrivial strongly connected components C in the graph $G[a]$ such that *sfair* is realizable in C, i.e., there exists a nontrivial strongly connected subset D of C satisfying $D \cap Sat(a_i) = \varnothing$ or $D \cap Sat(b_i) \neq \varnothing$, for all $1 \leqslant i \leqslant k$. It then follows that

$$Sat_{sfair}(\exists \Box\, a) \;=\; \{\, s \in S \mid Reach_{G[a]}(s) \cap T \neq \varnothing \,\}.$$

Here, $Reach_{G[a]}(s)$ is the set of states that is reachable from s in $G[a]$. Note that as SCC D is contained in SCC C, D can be reached from any state in C; reachability of C is thus of relevance.

The key part of the model-checking algorithm is now the computation of the set T. This amounts to investigating the SCCs C of $G[a]$ and checking for which SCC *sfair* is realizable. As the computation of T for the general case is slightly involved, we first consider two special (and simpler) cases: a_i = true for all i, i.e., unconditional fairness, and the case for which k=1, i.e., a single strong fairness constraint.

In the sequel, each set C of nodes in $G[a]$ is identified with the subgraph (C, E_C) where E_C results from the edge relation in $G[a]$ by removing all edges with starting or target vertex not in C.

Unconditional Fairness Let a_i = true for all $1 \leqslant i \leqslant k$. In this case

$$sfair \;\equiv \bigwedge_{1 \leqslant i \leqslant k} \Box \Diamond\, b_i.$$

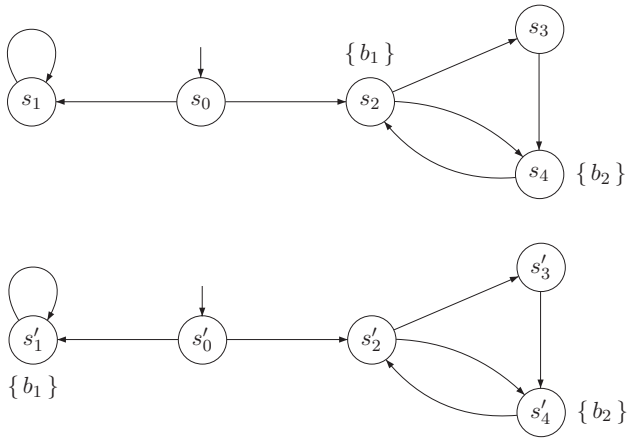
Obviously, $s \models_{sfair} \exists \Box\, a$ if and only if there exists a nontrivial SCC C in $G[a]$ that is reachable from s, such that C contains at least one b_i-state for any i. In that case, $\Box \Diamond\, b_i$ is realizable in SCC C, for any i. Let T be the set union of all nontrivial SCCs C of $G[a]$ satisfying $C \cap Sat(b_i) \;\neq\; \varnothing$ for all $1 \leqslant i \leqslant k$. It now follows that

$$s \models_{sfair} \exists \Box\, a \quad \text{if and only if} \quad Reach_{G[a]}(s) \cap T \neq \varnothing.$$

The following example illustrates this.

Example 6.41.

Consider the transition systems TS (upper) and TS' (lower) in which it is implicitly assumed that all states are labeled with the atomic proposition a:



(Due to this assumption, $TS[a] = TS$ and $TS'[a] = TS'$.) Consider the unconditional fairness assumption:

$$sfair \;=\; \Box\Diamond b_1 \;\wedge\; \Box\Diamond b_2.$$

The transition system TS contains the reachable nontrivial SCC $C = \{ s_2, s_3, s_4 \}$ such that $C \cap Sat(b_1) \neq \varnothing$ and $C \cap Sat(b_2) \neq \varnothing$. We thus have $s_0 \models_{sfair} \exists\Box\, a$, and hence $TS \models_{sfair} \exists\Box\, a$. On the other hand, TS' contains two non-trivial SCCs, but none that contains a b_1-state *and* a b_2-state. Therefore $s'_0 \not\models_{sfair} \exists\Box\, a$ and hence $TS' \not\models_{sfair} \exists\Box\, a$. ■

Strong Fairness with a Single Fairness Constraint Assume k=1. i.e., *sfair* is assumed to be of the form

$$sfair \;=\; \Box\Diamond a_1 \rightarrow \Box\Diamond b_1.$$

We have that $s \models_{sfair} \exists\Box a$ if and only if there exists a nontrivial strongly connected component C of $G[a]$ with $C \subseteq Reach_{G[a]}(s)$ such that at least one of the following two conditions (1) or (2) holds:

(1) $C \cap Sat(b_1) \neq \varnothing$, or

(2) $D \cap Sat(a_1) = \varnothing$ for some nontrivial SCC D of C.

Algorithm 18 Computation of $Sat_{sfair}(\exists\Box a)$

Input: A finite TS without terminal states, $a \in AP$ and $fair = \bigwedge_{0<i\leqslant k} sfair_i$ with $sfair_i = \Box\Diamond a_i \rightarrow \Box\Diamond b_i$
Output: $\{ s \in S \mid s \models_{fair} \exists\Box a \}$

```
compute the SCCs of the state graph G[a] of TS[a];
T := ∅;
for all nontrivial SCCs C in G[a] do
                        (* check whether the fairness assumption sfair can be realized in C *)
  if CheckFair(C, k, sfair_1, ..., sfair_k) then
    T := T ∪ C;
  fi
od
return { s ∈ S | Reach_{G[a]}(s) ∩ T ≠ ∅ }          (* e.g., backwards reachability *)
```

Intuitively, in case (1) C stands for a cyclic set of states that realizes $\Box\Diamond b_1$, while D in (2) represents a cyclic set of states for which $\neg a_1$ continuously holds. The SCCs D of C that do not contain any a_1-state can easily be computed by determining the nontrivial SCCs in the graph that is obtained from C by eliminating all a_1-states. (Stated differently, C realizes the unconditional fairness constraint $\Box\Diamond b_1$, while D realizes the unconditional fairness constraint $\Box\Diamond \mathrm{true}$ in the transition system induced by C after eliminating all a_1-states. This characterization is helpful to understand the general algorithm below.) Every path that has a suffix consisting of the infinite repetition of a cycle that runs through all states of C (case (1)) or that eventually reaches D and stays there forever satisfies the fairness assumption *sfair*. Accordingly, we may define T as the union of all nontrivial SCCs C of $G[a]$ satisfying the above constraints (1) and (2). Then, $s \models_{sfair} \exists\Box a$ if and only if $Reach_{G[a]}(s) \cap T \neq \varnothing$.

Strong Fairness with $k > 1$ Fairness Constraints In this case, *sfair* is defined for $k > 1$ by

$$sfair \;=\; \bigwedge_{1\leqslant i\leqslant k} sfair_i \quad \text{with} \quad sfair_i \;= \Box\Diamond a_i \rightarrow \Box\Diamond b_i.$$

As for the other cases, the initial step is to determine $G[a]$ and its set of SCCs. We have to compute all nontrivial SCCs C such that for some cyclic subset D of C all fairness constraints $\Box\Diamond a_i \rightarrow \Box\Diamond b_i$ are realizable in D, i.e., for $1 \leqslant i \leqslant k$:

$$D \cap Sat(a_i) = \varnothing \quad \text{or} \quad D \cap Sat(b_i) \neq \varnothing.$$

Algorithm 18 on page 370 provides the schema for the computation of $Sat_{sfair}(\exists\Box a)$. It uses the recursive procedure *CheckFair* (see Algorithm 19, page 372) to check the real-

izability of *sfair* in SCC C of $G[a]$. The inputs of *CheckFair* are a cyclic strongly connected subgraph D of $G[a]$, index j, and j strong fairness assumptions $\mathit{sfair}_{i_1}, \ldots, \mathit{sfair}_{i_j}$. $\mathit{CheckFair}(D, j, \mathit{sfair}_{i_1}, \ldots, \mathit{sfair}_{i_j})$ returns true if $\bigwedge_{1 \leqslant \ell \leqslant j} \mathit{sfair}_{i_\ell}$ is realizable in D. Thus, the invocation $\mathit{CheckFair}(C, k, \mathit{sfair}_1, \ldots, \mathit{sfair}_k)$ yields an affirmative answer whether $C \subseteq T$ by returning true if and only if the fairness assumption $\mathit{sfair} = \bigwedge_{1 \leqslant i \leqslant k} \mathit{sfair}_i$ is realizable in C.

The idea of Algorithm 19, $\mathit{CheckFair}(C, k, \mathit{sfair}_1, \ldots, \mathit{sfair}_k)$, is as follows. First, it is checked whether $C \cap \mathit{Sat}(b_i) \neq \varnothing$ for each fairness constraint $\mathit{sfair}_i = \Box\Diamond a_i \rightarrow \Box\Diamond b_i$.

- If yes, then $\mathit{sfair} = \bigwedge_{0 < i \leqslant k} \mathit{sfair}_i$ is realizable in C.
- Otherwise, there exists an index $j \in \{\, 1, \ldots, k \,\}$ such that $C \cap \mathit{Sat}(b_j) = \varnothing$. The goal is then to realize the condition

$$\bigwedge_{\substack{0 < i \leqslant k \\ i \neq j}} \mathit{sfair}_i \;\wedge\; \Box\neg a_j$$

  in C. For this, we investigate the subgraph $C[\neg a_j]$ of C that arises by removing all states where a_j holds and their incoming and outgoing edges. The goal is to check the existence of a cyclic subgraph of $C[\neg a_j]$ such that the remaining $k-1$ fairness constraints $\mathit{sfair}_1, \ldots, \mathit{sfair}_{j-1}, \mathit{sfair}_{j+1}, \ldots, \mathit{sfair}_k$ are realizable in this subgraph. This is done by analyzing the nontrivial SCCs D of $C[\neg a_j]$.
  - If there is no nontrivial SCC D of $C[\neg a_j]$, then *sfair* is not realizable in C.
  - Otherwise, invoke $\mathit{CheckFair}(D, k{-}1, \ldots)$ for each of these nontrivial SCCs D of $C[\neg a_j]$ to check whether the remaining $k{-}1$ fairness constraints are realizable in D.

The main steps of this procedure are summarized in Algorithm 19 on page 372. Note that in each recursive call one of the fairness constraints is removed, and thus the recursion depth is at most k, in which case $k = 0$ and the condition "$\forall i \in \{1, \ldots, k\}. C \cap \mathit{Sat}(b_j) \neq \varnothing$" of the first ifstatement is obviously fulfilled.

The cost function for $\mathit{CheckFair}(C, k, \mathit{sfair}_1, \ldots, \mathit{sfair}_k)$, is given by the recurrence equation:

$$T(n,k) = \mathcal{O}(n) + \max\{ \sum_{1 \leqslant \ell \leqslant r} T(n_\ell, k{-}1) \mid n_1, \ldots, n_r \geqslant 1, n_1 + \ldots + n_r \leqslant n \}$$

where n is the size (number of vertices and edges) of the subgraph C of $G[a]$. The values $n_1, \ldots, n_r$ denote the sizes of the nontrivial strongly connected components $D_1, \ldots, D_r$ of $C[\neg a_j]$. It is easy to see that the solution of this recurrence is $\mathcal{O}(n{\cdot}k)$. This yields:

Algorithm 19 Recursive algorithm $CheckFair(C, k, sfair_1, \ldots, sfair_k)$

Input: nontrivial SCC C in $G[a]$, and strong fairness constraints $sfair_i = \Box\Diamond a_i \rightarrow \Box\Diamond b_i$, $i = 1, \ldots, k$.
Output: true if $\bigwedge_{1 \leqslant i \leqslant k} sfair_i$ is realizable in C. Otherwise false.

---

if $\forall i \in \{1, \ldots, k\}. C \cap Sat(b_j) \neq \varnothing$ **then**
  return true (* $\bigwedge_{1 \leqslant i \leqslant k} \Box\Diamond b_i$ is realizable in C *)
else
  choose an index $j \in \{1, \ldots, k\}$ with $C \cap Sat(b_j) = \varnothing$;
  if $C[\neg a_j]$ is acyclic (or empty) **then**
    return false
  else
    compute the nontrivial SCCs of $C[\neg a_j]$;
    for all nontrivial SCC D of $C[\neg a_j]$ **do**
      if $CheckFair(D, k-1, sfair_1, \ldots, sfair_{j-1}, sfair_{j+1}, \ldots, sfair_k)$ **then**
        return true
      fi
    od
  fi

fi
return false

Theorem 6.42. Time Complexity of Verifying $\exists\Box a$ under Fairness

For transition system TS with N states and K transitions, and CTL strong fairness assumption fair with k conjuncts, the set $Sat_{fair}(\exists\Box\, a)$ can be computed in $\mathcal{O}\,((N{+}K)\cdot k)$.

The time complexity is thus linear in the size of the transition system and the number of imposed fairness constraints. The set T resulting from all SCCs C of $G[a]$, for which $\mathit{CheckFair}(C,1)$ returns the value true, can be computed (with appropriate implementation) in time $\mathcal{O}(\mathit{size}(G[a])\cdot k)$. A reachability analysis (by, e.g., depth-first search) results in the set

$$Sat_{fair}(\exists\Box a) \quad = \quad \{\, s \in S \mid Reach_{G[a]}(s) \cap T \neq \varnothing \,\}.$$

Gathering these results yields that CTL model checking under strong fairness constraints and for CTL formulae in ENF can be done in time linear in the size of the transition system, the length of the formula, and the number of (strong) fairness constraints. This, in fact, also holds for arbitrary CTL formulae (with universal quantification which can be treated by techniques similar as the ones we presented for existential quantification) and weak fairness constraints, or any mixture of unconditional, strong, and weak fairness constraints. We thus obtain:

Theorem 6.43. Time Complexity of CTL Model Checking with Fairness

For transition system TS with N states and K transitions, CTL formula Φ, and CTL fairness assumption fair with k conjuncts, the CTL model-checking problem $TS \models_{fair} \Phi$ can be determined in time $\mathcal{O}((N{+}K)\cdot|\Phi|\cdot k)$.

6.6 Counterexamples and Witnesses

A major strength of model checking is the possibility of generating a counterexample in case a formula is refuted. Let us first explain what is meant by a counterexample. In the case of LTL, a counterexample for $TS \not\models \varphi$ is a sufficiently long prefix of a path π that indicates why π refutes φ. For instance, a counterexample for the LTL formula $\Diamond\, a$ is a finite prefix of just $\neg a$-states that ends with a single cycle traversal. Such counterexample suggests that there is a $\Box\,\neg a$-path. Similarly, a counterexample for $\bigcirc\, a$ consists of a path π for which $\pi[1]$ violates a.

For CTL the situation is somewhat more involved due to the existential path quantification. For CTL formulae of the form $\forall\varphi$ a sufficiently long prefix of π with $\pi \not\models \varphi$ provides—as in LTL—sufficient information about the source of the refutation. In the

case of a path formula of the form $\exists\varphi$, it is unclear what a counterexample will be: if $TS \not\models \exists\varphi$, then all paths violate φ and no path satisfies φ. However, if one checks the formula $\exists\varphi$ for a transition system TS, then it is quite natural that for the answer "yes, $TS \models \exists\varphi$" one aims at an initial path where φ holds, while the answer "no" might be sufficient.[6] Therefore, CTL model checking supports the system diagnosis by providing *counterxamples* or *witnesses*. Intuitively, counterexamples indicate the refutation of universally quantified path formulae, while witnesses indicate the satisfaction of existentially quantified path formulae. From a path-based view, the concepts are counterexamples and witnesses can be explained as follows:

- a sufficiently long prefix of a path π with $\pi \not\models \varphi$ is a *counterexample* for the CTL path formula $\forall\varphi$, and
- a sufficiently long prefix of a path π with $\pi \models \varphi$ is a *witness* of the CTL path formula $\exists\varphi$.

To exemplify the idea of generating a witness, consider the following well-known combinatorial problem.

Example 6.44. The Wolf-Goat-Cabbage Problem

Consider the problem of bringing a ferryman (f), a goat (g), a cabbage (c) and a wolf (w) from one side of a river to the other side. The only available means to go from one riverside to another is a small boat that is capable of carrying at most two occupants. In order for the boat to be steered, one of the occupants needs to be the ferryman. Of course, the boat does not need to be fully occupied, and the ferryman can cross the river alone. For obvious reasons, neither the goat and the cabbage nor the goat and the wolf should be left unobserved by the ferryman. The question is whether there exists a series of boat movements such that all three items can be carried by the ferryman to the other side of the river.

This problem can be represented as a CTL model-checking problem in a rather natural way. The behavior of the goat, wolf, and cabbage is provided by a simple two-state transition system depicted in Figure 6.16. The state identifiers indicate the position of each of the items: 0 stands for the current (i.e., starting) riverside, and 1 for the other side

[6] Since the standard CTL model-checking procedure calculates the set of states where the given (state) formula Φ holds, also some information extracted from $Sat(\Phi)$ could be returned to the user. For instance, if $TS \not\models \exists\varphi$, then the model checker might return the set of initial states s_0 where $s_0 \not\models \exists\varphi$. This information could be understood as a counterexample and used for debugging purposes. This issue will not be addressed here. Instead we will discuss the concept of path-based counterexamples (also often called "error traces") and their duals, i.e., computations that provide a proof for the existence of computations with certain properties.

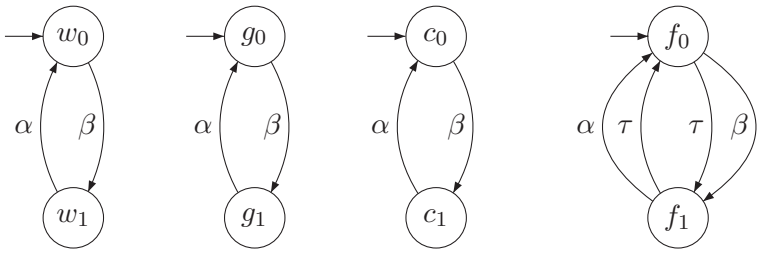


Figure 6.16: Transition systems for the wolf, goat, cabbage, and ferryman.

(i.e., the goal). The synchronization actions α and β are used to describe the boat trips that are taking place together with the ferryman. The ferryman behaves very similarly, but has in addition the possibility to cross the river alone. This corresponds to the two τ-labeled transitions. The resulting transition system representing the entire "system"

$$TS \;=\; (\mathit{wolf} \,|||\, \mathit{goat} \,|||\, \mathit{cabbage}) \parallel \mathit{ferryman},$$

has $2^4 = 16$ states. The resulting transition system is depicted in Figure 6.17. Note that the transitions are all bidirectional as each boat movement can be reversed. The existence of a sequence of boat movements to bring the two animals and the cabbage to the other riverbank can be expressed as the CTL state formula $\exists\varphi$ where

$$\varphi \;=\; \Big(\bigwedge_{i=0,1} (w_i \wedge g_i \rightarrow f_i) \;\wedge\; (c_i \wedge g_i \rightarrow f_i) \Big) \;\mathsf{U}\; (c_1 \wedge f_1 \wedge g_1 \wedge w_1).$$

Here, the left part of the until formula forbids the scenarios in which the wolf and the goat, or the cabbage and the goat are left unobserved. A witness of the CTL path formula φ is an initial finite path fragment which leads from

$$\text{initial state } \langle c_0, f_0, g_0, w_0\rangle \text{ to target state } \langle c_1, f_1, g_1, w_1\rangle$$

and which does not pass through any of the (six) states in which the wolf and the goat or the goat and the cabbage are left alone on one riverbank. That is, e.g., the states $\langle c_0, f_0, g_1, w_1\rangle$ and $\langle c_1, f_0, g_1, w_0\rangle$ should be avoided. An example witness for φ is:

$\langle c_0, f_0, g_0, w_0\rangle$	goat to riverbank 1
$\langle c_0, f_1, g_1, w_0\rangle$	ferryman comes back to riverbank 0
$\langle c_0, f_0, g_1, w_0\rangle$	cabbage to riverbank 1
$\langle c_1, f_1, g_1, w_0\rangle$	goat back to riverbank 0
$\langle c_1, f_0, g_0, w_0\rangle$	wolf to riverbank 1
$\langle c_1, f_1, g_0, w_1\rangle$	ferryman comes back to riverbank 0
$\langle c_1, f_0, g_0, w_1\rangle$	goat to riverbank 1
$\langle c_1, f_1, g_1, w_1\rangle$	

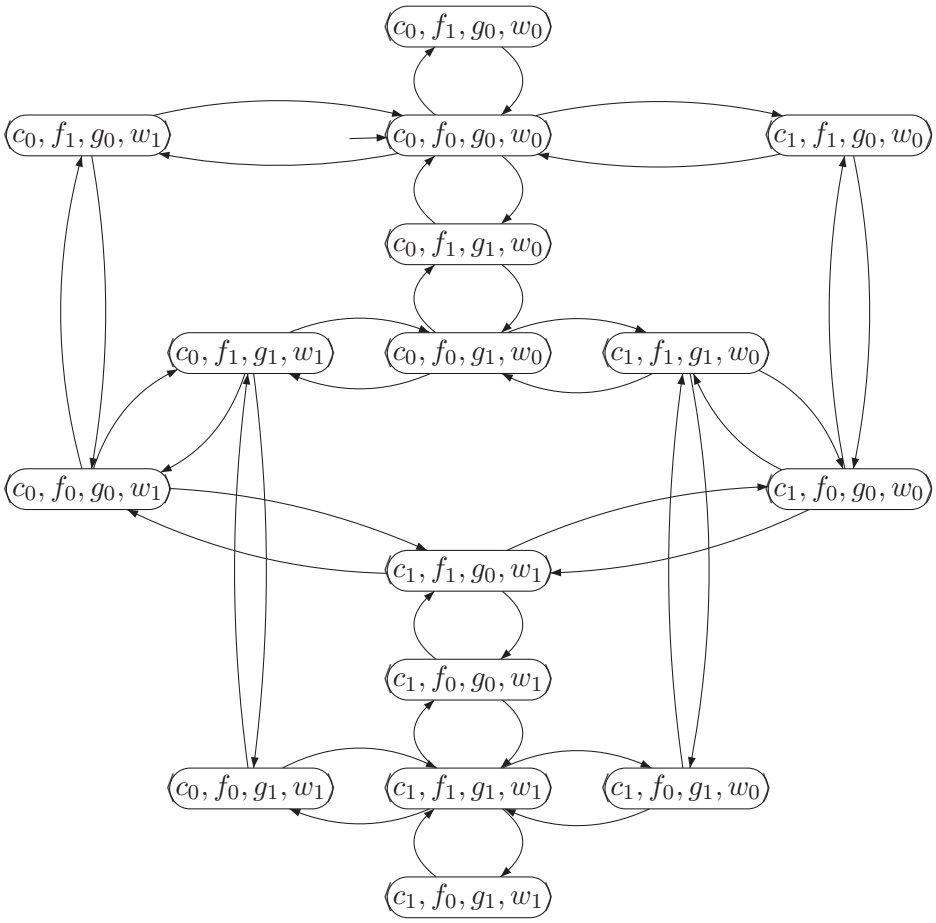


Figure 6.17: Transition system of the wolf-goat-cabbage problem.

■

6.6.1 Counterexamples in CTL

Now we explain how to generate counterexamples or witnesses for CTL (path) formulae. We consider here path formulae of the form $\bigcirc \Phi$, $\Phi \,\mathsf{U}\, \Psi$, and $\Box \Phi$. (Techniques for other operators, such as W or R, can be derived. Alternatively, corresponding considerations can be made for them.)

In the sequel, let $TS = (S, Act, \rightarrow, I, AP, L)$ be a finite transition system without terminal states.

The Next Operator A counterexample of $\varphi = \bigcirc \Phi$ is a pair of states (s, s') with $s \in I$ and $s' \in \mathit{Post}(s)$ such that $s' \not\models \Phi$. A witness of $\varphi = \bigcirc \Phi$ is a pair of states (s, s') with $s \in I$ and $s' \in \mathit{Post}(s)$ with $s' \models \Phi$. Thus, counterexamples and witnesses for the next-step operator result from inspecting the immediate successors of the initial states of *TS*.

The Until Operator A witness of $\varphi = \Phi \,\mathsf{U}\, \Psi$ is an initial path fragment $s_0\, s_1 \ldots s_n$ for which

$$s_n \models \Psi \quad \text{and} \quad s_i \models \Phi \text{ for } 0 \leqslant i < n.$$

Witnesses can be determined by a backward search starting in the set of Ψ-states.

A counterexample is an initial path fragment that indicates a path π for which either:

$$\pi \models \Box(\Phi \wedge \neg\Psi) \quad \text{or} \quad \pi \models (\Phi \wedge \neg\Psi) \,\mathsf{U}\, (\neg\Phi \wedge \neg\Psi).$$

For the first case, a counterexample is an initial path fragment of the form

$$\underbrace{s_0\, s_1 \ldots s_{n-1}\, \underbrace{s_n\, s'_1 \ldots s'_r}_{\text{cycle}}}_{\text{satisfy } \Phi \wedge \neg\Psi} \quad \text{with } s_n = s'_r.$$

For the second case, an initial path fragment of the form

$$\underbrace{s_0\, s_1 \ldots s_{n-1}}_{\text{satisfy } \Phi \wedge \neg\Psi}\; s_n \quad \text{with } s_n \models \neg\Phi \wedge \neg\Psi$$

does suffice as counterexample. Counterexamples can be determined by an analysis of the digraph $G = (S, E)$ where

$$E \;=\; \{\, (s, s') \in S \times S \mid s' \in \mathit{Post}(s) \;\wedge\; s \models \Phi \wedge \neg\Psi \,\}.$$

Then the SCCs of G are determined. Each path in G that starts in an initial state $s_0 \in S$ and leads to a nontrivial SCC C in G provides a counterexample of the form

$$s_0\, s_1 \ldots s_n \underbrace{s'_1 \ldots s'_r}_{\in C} \quad \text{with} \quad s_n = s'_r.$$

Each path in G that leads from an initial state s_0 to a trivial terminal SCC

$$C = \{\, s' \,\} \quad \text{with} \quad s' \not\models \Psi$$

provides a counterexample of the form $s_0\, s_1 \ldots s_n$ with $s_n \models \neg\Phi \wedge \neg\Psi$.

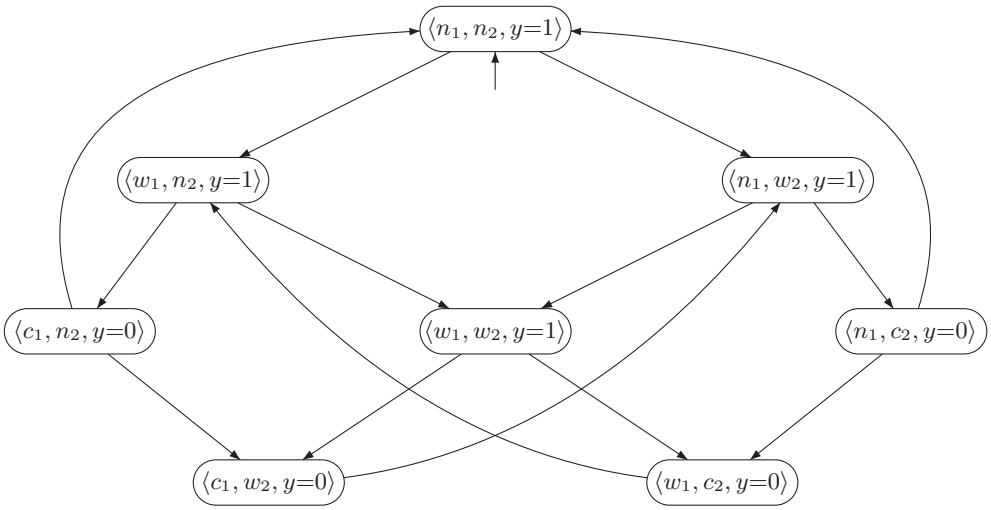


Figure 6.18: Transition system of semaphore-based mutual exclusion algorithm.

The Always Operator A counterexample for the formula $\varphi = \Box\Phi$ is an initial path fragment $s_0\, s_1 \ldots s_n$ such that $s_i \models \Phi$ for $0 \leqslant i < n$ and $s_n \not\models \Phi$. Counterexamples may be determined by a backward search starting in $\neg\Phi$-states.

A witness of $\varphi = \Box\, \Phi$ consists of an initial path fragment of the form

$$\underbrace{s_0\, s_1 \ldots s_n\, s'_1 \ldots s'_r}_{\text{satisfy } \Phi} \quad \text{with} \quad s_n = s'_r.$$

Witnesses can be determined by a simple cycle search in the digraph $G = (S, E)$ where the set of edges E is obtained from the transitions emanating from Φ-states, i.e., $E = \{\, (s, s') \mid s' \in \mathit{Post}(s) \ \wedge \ s \models \Phi \,\}$.

Example 6.45. Counterexamples and Semaphore-Based Mutual Exclusion

Recall the two-process mutual exclusion algorithm that exploits a binary semaphore y to resolve contention (see Eaxmple 3.9 on page 98). For convenience, the transition system TS_{Sem} of this algorithm is depicted in Figure 6.18. Consider the CTL formula over $AP = \{\, c_1, c_2, n_1, n_2, w_1, w_2 \,\}$:

$$\forall(\underbrace{((n_1 \wedge n_2) \ \vee \ w_2)}_{\Phi} \ \mathsf{U} \ \underbrace{c_2}_{\Psi}).$$

It expresses that process P_2 acquires access to the critical section once it starts waiting to enter it.

Note that the state labeling of TS_{Sem} can be directly obtained from the information in

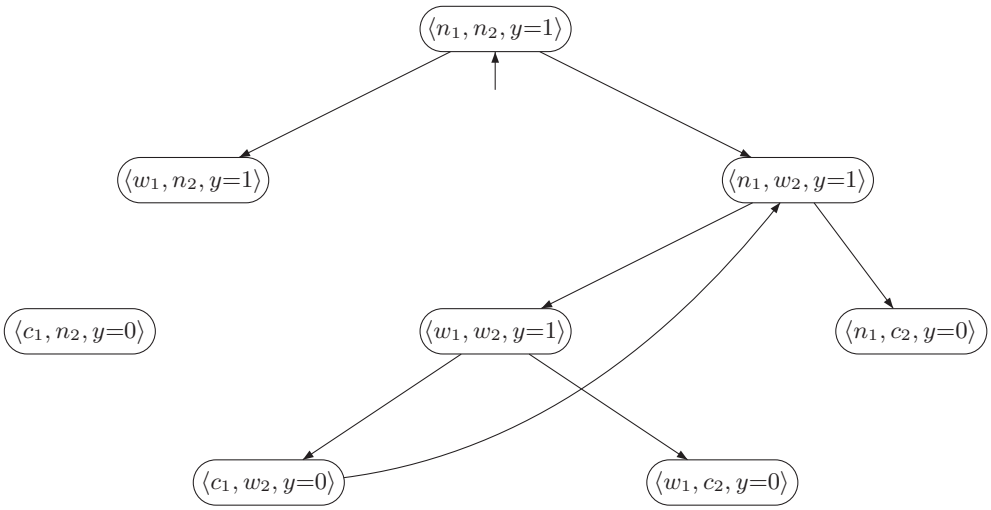


Figure 6.19: Graph to detect counterexamples for $\forall(((n_1 \wedge n_2) \ \vee \ w_2) \, \mathsf{U} \, c_2)$.

the states. Evidently, the CTL formula is refuted by TS_{Sem}. Counterexamples can be established as follows. First, the graph G is determined by only allowing edges that emanate from states satisfying $\Phi \wedge \neg\Psi$. This entails that all edges emanating from s with $s \models \neg((n_1 \wedge n_2) \ \vee \ w_2) \ \vee \ c_2$ are eliminated. This yields the graph depicted in Figure 6.19. The graph G contains a single nontrivial SCC C that is reachable from the initial state in TS_{Sem}. The initial path fragment:

$$\langle n_1, n_2, y{=}1 \rangle \underbrace{\langle n_1, w_2, y{=}1 \rangle \, \langle w_1, w_2, y{=}1 \rangle \, \langle c_1, w_2, y{=}0 \rangle}_{\in C}$$

is a counterexample as it shows that there is a path π in TS_{Sem} satisfying $\Box \, (((n_1 \wedge n_2) \ \vee \ w_2) \ \wedge \ \neg c_2)$, i.e., a path for which c_2 is never established.

Alternatively, any path from the initial state to the trivial terminal SCC satisfying $\neg c_2$ is a counterexample as well. This only holds for the terminal SCC $\{ \langle w_1, n_2, y{=}1 \rangle \}$. ■

Theorem 6.46. Time Complexity of Counterexample Generation

Let TS be a transition system TS with N states and K transitions and φ a CTL- path formula. If $TS \not\models \forall\varphi$, then a counterexample for φ in TS can be determined in time $\mathcal{O}(N{+}K)$. The same holds for a witness for φ, provided that $TS \models \exists\varphi$.

6.6.2 Counterexamples and Witnesses in CTL with Fairness

In the case CTL fairness assumptions are imposed, witnesses and counterexamples can be provided in a similar way as for CTL. Suppose *fair* is the fairness assumption of interest.

The Next Operator A witness for $\pi \models_{fair} \bigcirc a$ originates from a witness for $\pi \models \bigcirc (a \wedge a_{fair})$. Note that the latter is obtained as for CTL, as no fairness is involved. A counterexample to $\bigcirc a$ is a prefix of a fair path $\pi = s_0\, s_1\, s_2 \ldots$ in TS with $\pi \not\models \bigcirc a$. As π is fair and $\pi \not\models \bigcirc a$, it follows that $s_1 \models a_{fair}$ and $s_1 \not\models a$. So, $s_1 \not\models a_{fair} \rightarrow a$. A counterexample to $\bigcirc a$ with respect to $\models_{fair}$ thus results from a counterexample to the formula $\bigcirc (a_{fair} \rightarrow a)$ for CTL without fairness.

The Until Operator A witness of $a \,\mathsf{U}\, a'$ with respect to the fair semantics is a witness of $a \,\mathsf{U}\, (a' \wedge a_{fair})$ with respect to the standard CTL semantics. A counterexample for $a \,\mathsf{U}\, a'$ with respect to the fair semantics is either a witness of $(a \wedge a') \,\mathsf{U}\, (\neg a \wedge \neg a' \wedge a_{fair})$ under the common semantics $\models$ or a witness of $\Box(a \wedge \neg a')$ with respect to the fair semantics (explained below).

The Always Operator For a formula of the form $\Box a$, a counterexample with respect to the fair semantics is an initial path fragment $s_0\, s_1 \ldots s_n$ such that:

$$s_n \models \neg a \wedge a_{fair} \quad \text{and} \quad s_i \models a \;\; \text{for } 0 \leqslant i < n.$$

Consider the strong fairness assumption of the form:

$$\mathit{sfair} \;=\; \bigwedge_{0<i\leqslant k} (\Box \Diamond a_i \rightarrow \Box \Diamond b_i).$$

A witness of $\Box a$ under *sfair* is an initial path fragment

$$\underbrace{s_0\, s_1 \ldots s_n\, s'_1\, s'_2 \ldots s'_r}_{\models a} \qquad \text{with } s_n = s'_r$$

such that for all $0 < i \leqslant k$ it holds that

$$\mathit{Sat}(a_i) \cap \{\, s'_1, \ldots, s'_r \,\} = \varnothing \text{ or } \mathit{Sat}(b_i) \cap \{\, s'_1, \ldots, s'_r \,\} \neq \varnothing.$$

A witness can be computed by means of an analysis of the SCCs of a digraph that originates from the state graph of TS after some slight modifications. The costs are (multiplicatively) linear in the number of fairness conditions and in the size of the state graph.

Theorem 6.47. Time Complexity of Fair Counterexample Generation

For transition system TS with N states and K transitions, CTL path formula φ and CTL fairness assumption fair with k conjuncts such that $TS \not\models_{fair} \forall\varphi$, a counterexample for φ in TS can be determined in time $\mathcal{O}((N+K)\cdot k)$. The same holds for a witness for φ if $TS \models \exists\varphi$.

Example 6.48.

Consider the transition system depicted in Figure 6.11 (page 350) and assume the formula of interest is:

$$\exists\Box\,(a \;\vee\; (b = c)) \quad \text{under fairness constraint} \quad \mathit{sfair} = \Box\Diamond\,(q \wedge r) \rightarrow \Box\Diamond\,\neg(q \;\vee\; r).$$

The strong CTL fairness constraint asserts that in case either state s_0 or s_2 is visited infinitely often, then also state s_3 or s_7 needs to be visited infinitely often. The path $s_1\,s_3\,s_0\,s_2\,s_0$ is a witness of $\exists\Box\,(a \;\vee\; (b = c))$ in the absence of any fairness constraint. It is, however, no witness under *sfair* as it visits s_0 (and s_2) infinitely often, but not s_3 or s_7. On the other hand, the path $s_1\,s_3\,s_0\,s_2\,s_1$ is a witness under *sfair*. ■

6.7 Symbolic CTL Model Checking

The CTL model-checking procedure described so far relies on the assumption that the transition system has an explicit representation by the predecessor and successor lists per state. Such an *enumerative* representation is not adequate for very large transition systems. To attack the state explosion problem, the CTL model-checking procedure can be reformulated in a *symbolic* way where sets of states and sets of transitions are represented rather than single states and transitions. This set-based approach is very natural for CTL, since its semantics and model-checking algorithm rely on satisfaction sets for subformulae. There are several possibilities to realize the CTL model checking algorithm in a purely set-based stetting. The most prominent ones rely on a binary encoding of the states, which permits identifying subsets of the state space and the transition relation with switching functions. To obtain compact representations of switching functions, special data structures have been developed, such as ordered *binary decision diagrams*. Other forms for the representation of switching functions, such as conjunctive normal forms, could be used as well. They are widely used in the context of so-called *SAT-based model checking* where the model-checking problem is reduced to the satisfiability problem for propositional formulae (SAT). The SAT-based techniques will not be described in this book. Instead we will explain the main ideas of the approach with binary decision diagrams.

We first explain the general ideas behind symbolic approaches which operate on *sets of states* rather than single states and rely on a representation of transition systems by switching functions. In the sequel, let $TS = (S, \rightarrow, I, AP, L)$ be a "large", but finite transition system. The set of actions is irrelevant here and has been skipped. That is, the transition relation $\rightarrow$ is a subset of $S \times S$. Let $n \geqslant \lceil \log |S| \rceil$. (Since we suppose a large transition system, we can safely assume that $|S| \geqslant 2$.) We choose an arbitrary (injective) encoding $enc : S \rightarrow \{0,1\}^n$ of the states by bit vectors of length n. Although enc might not be surjective, it is no restriction to suppose that $enc(S) = \{0,1\}^n$, since all elements $(b_1, \ldots, b_n) \in \{0,1\}^n \setminus enc(S)$ can be treated as the encoding of pseudo states that cannot be reached from any proper state $s \in S$. The transitions of these pseudo states are arbitrary. The idea is now to identify the states $s \in S = enc^{-1}(\{0,1\}^n)$ with their encoding $enc(s) \in \{0,1\}^n$ and to represent any subset T of S by its characteristic function $\chi_T : \{0,1\}^n \rightarrow \{0,1\}$, which evaluates to true exactly for the (encodings of the) states $s \in T$. Similarly, the transition relation $\rightarrow \subseteq S \times S$ can be represented by a Boolean function $\Delta : \{0,1\}^{2n} \rightarrow \{0,1\}$ that assigns 1 to exactly those pairs (s, s') of bit vectors of length n each where $s \rightarrow s'$.

On the basis of this encoding, the CTL model-checking procedure can be reformulated to an algorithm that operates on the representation of TS by binary decision diagrams for Δ and the characteristic functions $\chi_{Sat(a)}$ for the satisfaction sets of the atomic propositions $a \in AP$. The remainder of this section is concerned with explanations on this approach. Section 6.7.1 summarizes our notations for switching functions and operations on them. The encoding of transition systems by switching functions and a corresponding reformulation of the CTL model-checking algorithm will be presented in Section 6.7.2. The main concepts of (ordered) binary decision diagrams are summarized in Section 6.7.3.

6.7.1 Switching Functions

For technical reasons, it is more appropriate to consider switching functions as mappings from evaluations for certain Boolean variables to the values 0 or 1 rather than functions $\{0,1\}^n \rightarrow \{0,1\}$. This permits simpler definitions of composition operators as we just have to identify the common variables, rather than refer to common arguments via their positions in bit tuples. Furthermore, the reference of the arguments of a switching function by means of variable names is also central for binary decision diagrams.

Let $z_1 \ldots, z_m$ be Boolean variables and $Var = \{z_1, \ldots, z_m\}$. Let $Eval(z_1, \ldots, z_m)$ denote the set of evaluations for $z_1, \ldots, z_m$, i.e., functions $\eta : Var \rightarrow \{0,1\}$. Evaluations are written as $[z_1 = b_1, \ldots, z_m = b_m]$. We often use tuple-notations such as $\overline{z}$ for the variable tuple $(z_1, \ldots, z_m)$, $\overline{b}$ for a bit tuple $(b_1, \ldots, b_m) \in \{0,1\}^m$, and $[\overline{z} = \overline{b}]$ as a shorthand for the evaluation $[z_1 = b_1, \ldots, z_m = b_m]$.

Notation 6.49. Switching Function

A *switching function* for $Var = \{z_1, \ldots, z_m\}$ is a function $f : Eval(Var) \to \{0,1\}$. The special case $m = 0$ (i.e., $Var = \varnothing$) is allowed. The switching functions for the empty variable set are just constants 0 or 1. ■

To indicate the underlying set of variables of a switching function we often write $f(\overline{z})$ or $f(z_1, \ldots, z_m)$ rather than f. When an enumeration of the variables in Var is clear from the context, say $z_1, \ldots, z_m$, then we often simply write $f(b_1, \ldots, b_m)$ or $f(\overline{b})$ instead of $f([z_1 = b_1, \ldots, z_m = b_m])$ (or $f([\overline{z} = \overline{b}])$).

Disjunction, conjunction, negation and other Boolean connectives are defined for switching functions in the obvious way. For example, if f_1 is a switching function for $\{z_1, \ldots, z_n, \ldots z_m\}$ and f_2 a switching function for $\{z_n, \ldots z_m, \ldots, z_k\}$, where the z_i's are supposed to be pairwise distinct and $0 \leqslant n \leqslant m \leqslant k$, then $f_1 \vee f_2$ is a switching function for $\{z_1, \ldots, z_k\}$ and the values of $f_1 \vee f_2$ are given by

$$\begin{aligned}&(f_1 \vee f_2)([z_1 = b_1, \ldots, z_k = b_k])\\ &= \max\{\, f_1([z_1 = b_1, \ldots, z_m = b_m]),\ f_2([z_n = b_n, \ldots, z_k = b_k])\,\}.\end{aligned}$$

We often simply write z_i for the *projection function* $\mathrm{pr}_{z_i} : Eval(\overline{z}) \to \{0,1\}$, $\mathrm{pr}_{z_i}([\overline{z} = \overline{b}]) = b_i$ and 0 or 1 for the constant switching functions. With these notations, switching functions can be represented by Boolean connections of the variables z_i (viewed as projection functions) and constants. For instance, $z_1 \vee (z_2 \wedge \neg z_3)$ stands for a switching function.

Notation 6.50. Cofactor and Essential Variable

Let $f : Eval(z, y_1, \ldots, y_m) \to \{0,1\}$ be a switching function. The *positive cofactor* of f for variable z is the switching function $f|_{z=1} : Eval(z, y_1, \ldots, y_m) \to \{0,1\}$ given by

$$f|_{z=1}(\mathfrak{c}, b_1, \ldots, b_m) \;=\; f(1, b_1, \ldots, b_m)$$

where the bit tuple $(\mathfrak{c}, b_1, \ldots, b_m) \in \{0,1\}^{m+1}$ is short for the evaluation $[z = \mathfrak{c}, y_1 = b_1, \ldots, y_m = b_m]$. Similarly, the *negative cofactor* of f for variable z is the switching function $f|_{z=0} : Eval(z, y_1, \ldots, y_m) \to \{0,1\}$ given by $f|_{z=0}(\mathfrak{c}, b_1, \ldots, b_m) = f(0, b_1, \ldots, b_m)$. If f is a switching function for $\{z_1, \ldots, z_k,\ y_1, \ldots, y_m\}$, then we write $f|_{z_1=b_1,\ldots,z_k=b_k}$ for the *iterated cofactor*, also simply called *cofactor* of f, given by

$$f|_{z_1=b_1,\ldots,z_k=b_k} \;=\; (\ldots(f|_{z_1=b_1})|_{z_2=b_2}\ldots)|_{z_k=b_k}.$$

Variable z is called *essential* for f if $f|_{z=0} \neq f|_{z=1}$. ■

The values of $f|_{z_1=b_1,\ldots,z_k=b_k}$ for $f = f(z_1, \ldots, z_k, y_1, \ldots, y_m)$ are given by

$$f|_{z_1=b_1,\ldots,z_k=b_k}(\mathfrak{c}_1, \ldots, \mathfrak{c}_k, a_1, \ldots, a_m) \;=\; f(b_1, \ldots, b_k, a_1, \ldots, a_m)$$

where $(\mathfrak{c}_1, \ldots, \mathfrak{c}_k, a_1, \ldots, a_m)$ is identified with the evaluation $[z_1 = \mathfrak{c}_1, \ldots, z_k = \mathfrak{c}_k, y_1 = a_1, \ldots, y_m = a_m]$. As a consequence, the definition of (iterated) cofactors does not depend on the order in which the cofactors for single variables are considered, i.e.:

$$f|_{z_1=b_1,\ldots,z_k=b_k} \;=\; (\ldots(f|_{z_{i_1}=b_{i_1}})|_{z_{i_2}=b_{i_2}}\ldots)|_{z_{i_k}=b_{i_k}}$$

for each permutation $(i_1, \ldots, i_k)$ of $(1, \ldots, k)$.

Obviously, variable z is not essential for the cofactors $f|_{z=0}$ and $f|_{z=1}$. Thus, at most the variables in $\mathit{Var}\setminus\{z_1, \ldots, z_k\}$ are essential for $f|_{z_1=b_1,\ldots,z_k=b_k}$, provided that f is a switching function for Var.

Example 6.51. Cofactors and Essential Variables

Consider the switching function $f(z_1, z_2, z_3)$ given by $(z_1 \vee \neg z_2) \wedge z_3$. Then, $f|_{z_1=1} = z_3$ and $f|_{z_1=0} = \neg z_2 \wedge z_3$. In particular, variable z_1 is essential for f.

When we fix the variable set $\{z_1, z_2, z_3\}$, then variables z_2 and z_3 are not essential for the projection function $pr_{z_1} = z_1$. In fact, we have $z_1|_{z_2=0} = z_1|_{z_2=1} = z_1$. On the other hand, z_1 is essential for the projection function z_1 as we have $z_1|_{z_1=1} = 1 \neq 0 = z_1|_{z_1=0}$.

For another example, variables z_1 and z_2 are essential for $f(z_1, z_2, z_3) = z_1 \vee \neg z_2 \vee (z_1 \wedge z_2 \wedge \neg z_3)$, while variable z_3 is not, as $f|_{z_3=1} = z_1 \vee \neg z_2$ agrees with $f|_{z_3=0} = z_1 \vee \neg z_2 \vee (z_1 \wedge z_2)$. ■

The following lemma yields a decomposition of f into its cofactors. This simple observation relies on the fact that for any evaluation where z is assigned to 0 the value of $f(z, \overline{y})$ agrees with the value of $f|_{z=0}$ under the same evaluation for $\overline{y}$. And similarly, $f([z = 1, \overline{y} = \overline{b}]) = f|_{z=1}([\overline{y} = \overline{b}])$.

Lemma 6.52. Shannon Expansion

If f is a switching function for Var , then for each variable $z \in$ Var:

$$f \;=\; (\neg z \wedge f|_{z=0}) \;\vee\; (z \wedge f|_{z=1}).$$

A simple consequence of the Shannon expansion is that z is not essential for f if and only if $f = f|_{z=0} = f|_{z=1}$.

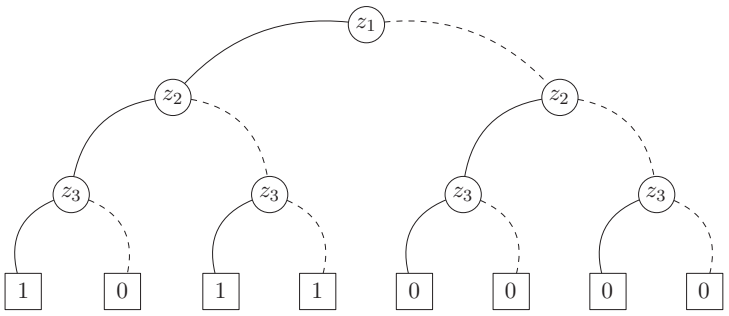


Figure 6.20: Binary decision tree for $z_1 \wedge (\neg z_2 \vee z_3)$.

Remark 6.53. Binary Decision Trees

The Shannon expansion is inherent in the representation of switching functions by *binary decision trees*. Given a switching function f for some variable set *Var*, one first fixes an arbitrary enumeration $z_1, \ldots, z_m$ for the variables in *Var* and then represents f by a binary tree of height m such that the two outgoing edges of the inner nodes at level i stand for the cases $z_i = 0$ (depicted by a dashed line) and $z_i = 1$ (depicted by a solid line). Thus, the paths from the root to a leaf in that tree represent the evaluations and the corresponding value. The leaves stand for the function values 0 or 1 of f. That is, given the evaluation $s = [z_1 = b_1, \ldots, z_m = b_m]$, then $f(s)$ is the value of the terminal node that is reached by traversing the tree from the root using the branch $z_i = b_i$ for the node at level i. The subtree of node v of the binary decision tree for f and the variable ordering $z_1, \ldots, z_m$ yields a representation of the iterated cofactor $f|_{z_1=b_1,\ldots,z_{i-1}=b_{i-1}}$ (viewed as a switching function for $\{z_i, \ldots, z_m\}$) where $z_1 = b_1$, …, $z_{i-1} = b_{i-1}$ is the sequence of decisions made along the path from the root to node v.

An example of a binary decision tree for $f(z_1, z_2, z_3) = z_1 \wedge (\neg z_2 \vee z_3)$ is given in Figure 6.20. We use dashed lines for the edges from an inner node for variable z representing the case $z = 0$ and solid edges for the case $z = 1$. ■

Further operators on switching functions that will be needed later are existential quantification over variables and renaming of variables.

Notation 6.54. Existential and Universal Quantification

Let f be a switching function for *Var* and $z \in Var$. Then, $\exists z.f$ is the switching function given by:

$$\exists z.f \;=\; f|_{z=0} \vee f|_{z=1}.$$

If $\overline{z} = (z_1, \ldots, z_k)$ and $z_i \in \mathit{Var}$ for $1 \leqslant i \leqslant k$, then $\exists \overline{z}.f$ is a short-form notation for $\exists z_1.\exists z_2. \ldots \exists z_k.f$. Similarly, universal quantification is defined by

$$\forall z.f \;=\; f|_{z=0} \wedge f|_{z=1}$$

and $\forall \overline{z}.f \;=\; \forall z_1.\forall z_2. \ldots \forall z_k.f$. ■

For example, if $f(z, y_1, y_2)$ is given by $(z \vee y_1) \wedge (\neg z \vee y_2)$, then

$$\exists z.f \;=\; f|_{z=0} \vee f|_{z=1} \;=\; y_1 \vee y_2$$

and $\forall z.f \;=\; f|_{z=0} \wedge f|_{z=1} \;=\; y_1 \wedge y_2$.

The rename operator simply replaces certain variables with other ones. E.g., renaming variable z into y in $f(z, x) = \neg z \vee x$ yields the switching function $\neg y \vee x$. Formally:

Notation 6.55. Rename Operator

Let $\overline{z} = (z_1, \ldots, z_m)$, $\overline{y} = (y_1, \ldots, y_m)$ be variable tuples of the same length and let $\overline{x} = (x_1, \ldots, x_k)$ be a variable tuple such that none of the variables y_i or z_i is contained in $\overline{x}$. For the evaluation $s = [\overline{y} = \overline{b}] \in \mathit{Eval}(\overline{y}, \overline{x})$, $s\{\overline{y} \leftarrow \overline{z}\}$ denotes the evaluation in $\mathit{Eval}(\overline{z}, \overline{x})$ that is obtained by composing the variable renaming function $y_i \mapsto z_i$, for $1 \leqslant i \leqslant m$ with the evaluation s. That is, $s\{\overline{y} \leftarrow \overline{z}\}$ agrees with s for the variables in $\overline{x}$ and $s\{\overline{y} \leftarrow \overline{z}\}$ assigns the same value $b \in \{0, 1\}$ to variable z_i as s to variable y_i. Given a switching function $f : \mathit{Eval}(\overline{z}, \overline{x}) \to \{0, 1\}$, then the switching function $f\{\overline{z} \leftarrow \overline{y}\} : \mathit{Eval}(\overline{y}, \overline{x}) \to \{0, 1\}$ is given by

$$f\{\overline{z} \leftarrow \overline{y}\}(s) \;=\; f(\, s\{\overline{y} \leftarrow \overline{z}\}\,),$$

i.e., $f\{\overline{z} \leftarrow \overline{y}\}([\overline{y} = \overline{b}, \overline{x} = \overline{c}]) \;=\; f([\overline{z} = \overline{b}, \overline{x} = \overline{c}])$. If it is clear from the context which variables have to be renamed, then we simply write $f(\overline{y}, \overline{x})$ rather than $f\{\overline{z} \leftarrow \overline{y}\}$. ■

6.7.2 Encoding Transition Systems by Switching Functions

After this excursus on switching functions, we return to the question of a symbolic representation of a transition system $TS = (S, \to, I, AP, L)$. As mentioned above (see page 382), the action set is irrelevant for our purposes and therefore omitted. For the encoding of the states $s \in S$ we use n Boolean variables $x_1, \ldots, x_n$ and identify any evaluation $[x_1 = b_1, \ldots, x_n = b_n] \in \mathit{Eval}(\overline{x})$ with the unique state $s \in S$ such that $\mathit{enc}(s) = (b_1, \ldots, b_n)$. In the sequel, we suppose $S = \mathit{Eval}(\overline{x})$. Given a subset B of S, then the *characteristic function* $\chi_B : S \to \{0, 1\}$ of B assigns 1 to all states $s \in B$

and 0 to all states $s \notin B$. As we assume $S = \mathit{Eval}(\overline{x})$, the characteristic function is the switching function given by

$$\chi_B : \mathit{Eval}(\overline{x}) \to \{0,1\}, \quad \chi_B(s) \;=\; \begin{cases} 1 & \text{if } s \in B \\ 0 & \text{otherwise.} \end{cases}$$

In particular, for any atomic proposition $a \in AP$, the satisfaction set $\mathit{Sat}(a) = \{s \in S \mid s \models a\}$ can be represented by the switching function $f_a = \chi_{\mathit{Sat}(a)}$ for $\overline{x}$. This yields a symbolic representation of the labeling function by means of a family $(f_a)_{a \in AP}$ of switching functions for $\overline{x}$.

The symbolic representation of the transition relation $\to \subseteq S \times S$ relies on the same idea: we identify $\to$ with its characteristic function $S \times S \to \{0,1\}$ where truth-value 1 is assigned to the state pair (s,t) if and only if $s \to t$. Formally, we deal with the variable tuple $\overline{x} = (x_1, \ldots, x_n)$ to encode the starting state s of a transition and a copy $\overline{x}' = (x_1', \ldots, x_n')$ of $\overline{x}$ to encode the target state. That is, for each of the variables x_i we introduce a new variable x_i'. The original (unprimed) variables x_i serve to encode the current state, while the primed variables x_i' are used to encode the next state. Then, we identify the transition relation $\to$ of TS with the switching function

$$\Delta : \mathit{Eval}(\overline{x}, \overline{x}') \to \{0,1\}, \quad \Delta(s, t\{\overline{x}' \leftarrow \overline{x}\}) \;=\; \begin{cases} 1 & \text{if } s \to t \\ 0 & \text{otherwise} \end{cases}$$

where s and t are elements of the state space $S = \mathit{Eval}(\overline{x})$ and the second argument $t\{\overline{x}' \leftarrow \overline{x}\}$ in $\Delta(s, t\{\overline{x}/\overline{x}'\})$ is the evaluation for $\overline{x}'$ that assigns the same value (1 or 0) to variable x_i' as t assigns to variable x_i (cf. Notation 6.55).

Example 6.56. Symbolic Representation of Transition Relation

Suppose that TS has two states s_0 and s_1 and the transitions $s_0 \to s_0$, $s_0 \to s_1$ and $s_1 \to s_0$, then we need a single Boolean variable $x_1 = x$ for the encoding. Say we identify state s_0 by 0 and state s_1 by 1. The transition relation $\to$ is represented by the switching function $\Delta : \mathit{Eval}(x, x') \to \{0,1\}$,

$$\Delta \;=\; \neg x \;\vee\; \neg x'.$$

Let us check why. The satisfying assignments for Δ are $[x = 0, x' = 0]$, $[x = 0, x' = 1]$ and $[x = 1, x' = 0]$. The first two evaluations (where $x = 0 = 0$) represent the two outgoing transitions from state $s_0 = 0$, while $[x = 1, x' = 0]$ stands for the transition $s_1 \to s_0$. ■

Example 6.57. Symbolic Representation of a Ring

Consider a transition system TS with states $\{s_0, \ldots, s_{k-1}\}$ where $k = 2^n$ that are organized in a ring, i.e., TS has the transitions

$$s_i \;\to\; s_{(i+1) \bmod k}$$

for $0 \leqslant i < k$. We use the encoding that identifies any state s_i with the binary encoding of its index i. E.g., if $k = 16$, then $n = 4$ and state s_1 is identified with 0001, state s_{10} with 1010, and state s_{15} with 1111. We use the Boolean variables $x_1, \ldots, x_n$ where x_n represents the most significant bit (i.e., the evaluation $[x_n = b_n, \ldots, x_1 = b_1]$ stands for state $\sum_{1 \leqslant i \leqslant n} b_i 2^{i-1}$). Then, Δ is a function with $2n$ variables, namely $x_1, x_2, \ldots, x_n$ and their copies $x'_1, x'_2, \ldots x'_n$. The values of the switching function $\Delta(\overline{x}, \overline{x}')$ are given by

$$\bigwedge_{1 \leqslant i < n} \Big(x_1 \wedge \ldots \wedge x_i \wedge \neg x_{i+1} \;\rightarrow\; x' \wedge \ldots \wedge x'_i \wedge x'_{i+1} \wedge \bigwedge_{j < i \leqslant n} (x_j \leftrightarrow x'_j) \Big)$$
$$\wedge\; (x_1 \wedge \ldots \wedge x_n \;\rightarrow\; \neg x'_1 \wedge \ldots \wedge \neg x'_n).$$

The set $B = \{\, s_{2i} \mid 0 \leqslant i < 2^{n-1} \,\}$ is given by the switching function $\chi_B(\overline{x}) = x_1$. ■

Given the switching function Δ and a state $s \in S = \mathit{Eval}(\overline{x})$, then the successor set $\mathit{Post}(s) = \{s' \in S \mid s \rightarrow s'\}$ arises from Δ by fixing evaluation s for $\overline{x}$. More precisely, if $s = [x_1 = b_1, \ldots, x_n = b_n]$, then a switching function $\chi_{\mathit{Post}(s)}$ for $\mathit{Post}(s)$ is obtained from Δ by building the cofactor for the variables $x_1, \ldots, x_n$ and the values $b_1, \ldots, b_n$:

$$\chi_{\mathit{Post}(s)} \;=\; \Delta|_s\{\overline{x} \leftarrow \overline{x}'\}$$

where $\Delta|_s$ stands for the iterated cofactor $\Delta|_{x_1=b_1,\ldots,x_n=b_n}$. As $\Delta|_s$ is a switching function for $\{x'_1, \ldots, x'_n\}$, the renaming operator $\{\overline{x} \leftarrow \overline{x}'\}$ yields a representation of $\mathit{Post}(s)$ by the variables $x_1, \ldots, x_n$.

Example 6.58.

The successor set $\mathit{Post}(s_0) = \{\, s_0, s_1 \,\}$ for the simple system in Example 6.56 is obtained symbolically by

$$\Delta|_{x=0}\{x \leftarrow x'\} \;=\; \underbrace{(\neg x \vee \neg x')|_{x=0}}_{=1}\{x \leftarrow x'\} \;=\; 1,$$

which reflects the fact that after identifying state s_0 with the evaluation $[x = 0]$ and state s_1 with $[x = 1]$ the successor set of s_0 is $\mathit{Eval}(x) = \{s_0, s_1\}$, and its characteristic function is the constant 1. For state $s_1 = [x = 1]$, a symbolic representation of the successor set $\mathit{Post}(s_1) = \{s_0\} = \{[x = 0]\}$ is obtained by

$$\Delta|_{x=1}\{x \leftarrow x'\} \;=\; \underbrace{(\neg x \vee \neg x')|_{x=1}}_{=\neg x'}\{x \leftarrow x'\} \;=\; \neg x$$

■

Remark 6.59. Symbolic Composition Operators

As we explained in Chapter 2, a crucial aspect for the feasibility of any algorithmic verification technique is the automatic construction of "large" transition systems to be analyzed by means of operators that compose several "small" transition systems $TS_1, \ldots, TS_m$ representing the processes to be executed in parallel. Let us suppose that we have appropriate representations for the switching functions $\Delta_1, \ldots, \Delta_m$ for transition systems $TS_1, \ldots, TS_m$ at our disposal. If TS arises by $TS_1, \ldots, TS_m$ through the synchronous product operator of TS, then the transition relation of TS is given by

$$\Delta(\overline{x}_1, \ldots, \overline{x}_m, \overline{x}'_1, \ldots, \overline{x}'_m) \;=\; \bigwedge_{1 \leqslant i \leqslant n} \Delta_i(\overline{x}_i, \overline{x}'_i)$$

where $\overline{x}_i$ denotes the variable tuple that is used to encode the states in TS_i. The justification for the above equation is that each transition $\langle s_1, \ldots, s_m \rangle \to \langle s'_1, \ldots, s'_m \rangle$ of $TS = TS_1 \otimes \ldots \otimes TS_m$ is composed of individual transitions $s_i \to s'_i$ in TS_i for each of the transition systems TS_i. For the other extreme, suppose that $TS = TS_1 \,|||\, \ldots \,|||\, TS_m$ arises by the interleaving operator of the TS_i's (without any synchronization or communication). Then,

$$\Delta(\overline{x}_1, \ldots, \overline{x}_m, \overline{x}'_1, \ldots, \overline{x}'_m) \;=\; \bigvee_{1 \leqslant i \leqslant n} \Big(\Delta_i(\overline{x}_i, \overline{x}'_i) \wedge \bigwedge_{\substack{1 \leqslant j \leqslant m \\ i \neq j}} \overline{x}_j = \overline{x}'_j \Big)$$

where for $\overline{x}_j = (x_{1,j}, \ldots, x_{n_j,j})$ and $\overline{x}'_j = (x'_{1,j}, \ldots, x'_{n_j,j})$ notation $\overline{x}_j = \overline{x}'_j$ abbreviates $\bigwedge_{1 \leqslant k \leqslant n_j} (x_{k,j} \leftrightarrow x'_{k,j})$. The justification for the above equation for Δ is that each transition in TS has the form

$$\langle s_1, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_m \rangle \to \langle s_1, \ldots, s_{i-1}, s'_i, s_{i+1}, \ldots, s_m \rangle$$

where exactly one transition system makes a move $s_i \to s'_i$, while the others do not change their local state. For the parallel operator $\|_H$ which relies on an interleaving semantics for actions outside H and synchronization over the actions in H, we need a combination of the above formulae for Δ. Here, in fact, we need a refined representation of the transition relation in TS_i with actions for the transitions by means of a switching function $\Delta_i(\overline{x}_i, \overline{z}, \overline{x}'_i)$ where the variable tuple $\overline{z}$ serves to encode the action names. In case $m = 2$, the switching function $\Delta(\overline{x}_1, \overline{x}_2, \overline{x}'_1, \overline{x}'_2)$ for the transition relation in $TS = TS_1 \|_H TS_2$ is given by

$$\exists \overline{z}. \Big(\begin{array}{rcl} (\quad \chi_H(\overline{z}) & \wedge & \Delta_1(\overline{x}_1, \overline{z}, \overline{x}'_1) \;\wedge\; \Delta_2(\overline{x}_1, \overline{z}, \overline{x}'_2) \quad) \quad \vee \\ (\; \neg\chi_H(\overline{z}) & \wedge & \Delta_1(\overline{x}_1, \overline{z}, \overline{x}'_1) \;\wedge\; \overline{x}_2 = \overline{x}'_2 \quad\quad) \quad \vee \\ (\; \neg\chi_H(\overline{z}) & \wedge & \Delta_2(\overline{x}_2, \overline{z}, \overline{x}'_2) \;\wedge\; \overline{x}_1 = \overline{x}'_1 \quad\quad) \quad \Big). \end{array}$$

In case TS is the transition system of a program graph or channel system then the Boolean variables x_i serve to encode the locations, the potential values of the variables, and the

channel contents. The effect of the actions has then to be rewritten in terms of these variables. ■

Given the switching function $\Delta(\overline{x},\overline{x}')$ and the characteristic function $\chi_B(\overline{x})$ for some set B of states, we can now describe the backward BFS-based reachability analysis to compute all states in $Pre^*(B) = \{s \in S \mid s \models \exists\Diamond B\}$ on the basis of switching functions. Initially, we start with the switching function $f_0 = \chi_B$ that characterizes the set $T_0 = B$. Then, we successively compute the characteristic functions $f_{j+1} = \chi_{T_{j+1}}$ of

$$T_{j+1} \;=\; T_j \cup \{s \in S \mid \exists s' \in S \text{ s.t. } s' \in \mathit{Post}(s) \;\wedge\; s' \in T_j \;\}$$

The set of states s where the condition $\exists s' \in S$ s.t. $s' \in \mathit{Post}(s)$ and $s' \in T_j$ holds is given by the switching function:

$$\exists\overline{x}'.\,(\;\underbrace{\Delta(\overline{x},\overline{x}')}_{s' \,\in\, \mathit{Post}(s)} \;\wedge \underbrace{f_j(\overline{x}')}_{s' \in T_j}\;).$$

Recall that $f_j(\overline{x}')$ is just a short notation for $f_j\{\overline{x}' \leftarrow \overline{x}\}$, i.e., arises from f_j by renaming the unprimed variables x_i into their primed copies x_i' for $1 \leqslant i \leqslant n$. This BFS-based technique can easily be adapted to treat constrained reachability properties $\exists(C \,\mathsf{U}\, B)$ for subsets B, C of S, as shown in Algorithm 20 on page 390.

Algorithm 20 Symbolic computation of $\mathit{Sat}(\exists(C \,\mathsf{U}\, B))$

$f_0(\overline{x}) := \chi_B(\overline{x})$;
$j := 0$;
repeat
  $f_{j+1}(\overline{x}) := f_{j+1}(\overline{x}) \;\vee\; (\;\chi_C(\overline{x}) \;\wedge\; \exists\overline{x}'.\,(\;\Delta(\overline{x},\overline{x}') \wedge f_j(\overline{x}')\;)\;)$;
  $j := j + 1$
until $f_j(\overline{x}) = f_{j-1}(\overline{x})$;
return $f_j(\overline{x})$.

If we are just interested in the one-step predecessors, i.e., properties of the form $\exists \bigcirc B$, then no iteration is needed and the characteristic function of the set of states $s \in S$ with $\mathit{Post}(s) \cap B \neq \varnothing$ is obtained by

$$\exists\overline{x}'.\,(\;\underbrace{\Delta(\overline{x},\overline{x}')}_{s' \,\in\, \mathit{Post}(s)} \;\wedge \underbrace{\chi_B(\overline{x}')}_{s' \in B}\;).$$

In a similar way, the set $\mathit{Sat}(\exists\Box B)$ of all states s that have an infinite path consisting of states in a given set B can be computed symbolically. Here, we mimic the computation of the largest set $T \subseteq B$ with $\mathit{Post}(t) \cap T \neq \varnothing$ for all $t \in T$ by the iteration $T_0 = B$ and

$$T_{j+1} \;=\; T_j \cap \{s \in S \mid \exists s' \in S \text{ s.t. } s' \in \mathit{Post}(s) \;\wedge\; s' \in T_j \;\}$$

Algorithm 21 Symbolic computation of $Sat(\exists\Box B)$

$f_0(\overline{x}) := \chi_B(\overline{x})$;
$j := 0$;
repeat
  $f_{j+1}(\overline{x}) := f_{j+1}(\overline{x}) \wedge \exists\overline{x}'.\,(\,\Delta(\overline{x},\overline{x}') \wedge f_j(\overline{x}')\,)$;
  $j := j + 1$
until $f_j(\overline{x}) = f_{j-1}(\overline{x})$;
return $f_j(\overline{x})$.

in a symbolic way, as shown in Algorithm 21 on page 391.

These considerations show that the CTL model checking problem "Does CTL formula Φ hold for TS?" can be solved symbolically by means of switching functions f_Ψ that represent the satisfaction sets $Sat(\Psi)$ of the subformulae Ψ of Φ. The satisfaction sets f_a for the atomic propositions $a \in AP$ are supposed to be given. Union, intersection, and complementation of sets of states correspond to disjunction, conjunction, and negation on the level of switching functions. E.g., the satisfaction set for $\Psi_1 \wedge \neg\Psi_2$, is obtained by $f_{\Psi_1} \wedge \neg f_{\Psi_2}$. The treatment of formulae $\exists(\Psi_1 \,\mathsf{U}\, \Psi_2)$ and $\exists\Box\Psi$ relies on the techniques sketched in Algorithms 20 and 21. To treat full CTL, we can either transform any CTL formula into an equivalent CTL formula in existential normal form or use analogous symbolic algorithms for universally quantified formulae such as $\forall(\Phi \,\mathsf{U}\, \Psi)$.

The major challenge is to find an appropriate data structure for the switching functions. Besides yielding compact representations of the satisfaction sets and the transition relation, this data structure has to support the Boolean connectives (disjunction, conjunction, complementation) and the comparison of two switching functions. The latter is needed in the termination criteria for the repeat loops in Algorithms 20 and 21.

Before presenting the definition of binary decision diagrams that have proven to be very efficient in many applications, let us first discuss other potential data structures. Truth tables can be ruled as they always have the size 2^n for switching functions with n variables. The same holds for binary decision trees since they always have $2^{n+1} - 1$ nodes. Conjunctive or disjunctive normal forms for the representation of switching functions by propositional logic formulae yield the problem that checking equivalence (i.e., equality for the represented switching functions) is expensive, namely coNP-complete. Furthermore, there are switching functions f_m with m essential variables where the length of any representation by a conjunctive normal form formula grows exponentially in m. The same holds for disjunctive normal forms. However, the latter is no proper argument against conjunctive or disjunctive normal forms, because there is *no* data structure that ensures representations of polynomial size for all switching functions. The reason for this is that the

number of switching functions for m variables $z_1, \ldots, z_m$ is double exponential in m. Note that $|Eval(z_1, \ldots, z_m)| = 2^m$, and hence, the number of functions $Eval(z_1, \ldots, z_m) \to \{0, 1\}$ is 2^{2^m}. Suppose we are given a universal data structure for switching functions (i.e., a data structure that can represent any switching function) such that K_m is the number of switching functions for $z_1, \ldots, z_m$ that can be represented by at most 2^{m-1} bits. Then:

$$K_m \;\leqslant\; \sum_{i=0}^{2^{m-1}} 2^i \;=\; 2^{2^{m-1}+1} - 1 \;<\; 2^{2^{m-1}+1}.$$

But then there are at least

$$2^{2^m} \;-\; 2^{2^{m-1}+1} \;=\; 2^{2^{m-1}+1} \cdot \left(2^{2^m - 2^{m-1} - 1} \;-\; 1\right) \;=\; 2^{2^{m-1}+1} \cdot \left(2^{2^{m-1}-1} - 1\right)$$

switching functions for $z_1, \ldots, z_m$ where the representation requires more than 2^{m-1} bits. This calculation shows that we cannot expect a data structure which is efficient for all switching functions. Nevertheless there are data structures which yield compact representations for many switching functions that appear in practical applications. A data structure that has been proven to be very successful for model checking purposes, in particular in the area of hardware verification, is *ordered binary decision diagrams (OBDDs)*. Besides yielding compact representation for many "realistic" transition systems, they enjoy the property that the Boolean connectives can be realized in time linear in the size of the input OBDDs and that (with appropriate implementation techniques) equivalence checking can even be performed in constant time.

In the remainder of this section, we explain those aspects of ordered binary decision diagrams that are relevant to understanding the main concepts of symbolic model checking with these data structures. Further theoretical aspects on binary decision diagrams, their variants and applications, can be found, e.g., in the textbooks [134, 180, 292, 300, 418]. For details on OBDD-based symbolic model checking we refer to [74, 92, 288, 374].

6.7.3 Ordered Binary Decision Diagrams

Ordered binary decision diagrams (OBDDs for short), originally proposed by Bryant [70], yield a data structure for switching functions that relies on a compactification of binary decision trees. The rough idea is to skip redundant fragments of a binary decision tree. This means collapsing constant subtrees (i.e., subtrees where all terminal nodes have the same value) into a single node and identifying nodes with isomorphic subtrees. In this way, we obtain a directed acyclic graph of outdegree 2 where – as in binary decision trees – the inner nodes are labeled by variables and their outgoing edges stand for the possible evaluations of the corresponding variable. The terminal nodes are labeled by the function value.

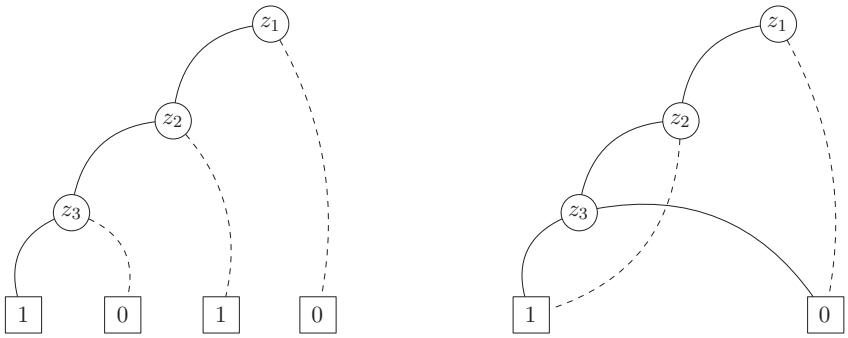


Figure 6.21: Binary decision diagram for $z_1 \wedge (\neg z_2 \vee z_3)$.

Example 6.60. From Binary Decision Tree to OBDD

Before presenting the formal definition of binary decision diagrams, we explain the main ideas by means of the function $f(z_1, z_2, z_3) = z_1 \wedge (\neg z_2 \vee z_3)$. A binary decision tree for f has been shown in Figure 6.20 on page 385. Since all terminal nodes in the right subtree of the root have value 0 (which reflects the fact that the cofactor $f|_{z_1=0}$ agrees with the constant function 0), the inner tests for variables z_2 and z_3 in that subtree are redundant and the whole subtree can be replaced by a terminal node with value 0. Similarly, the subtree of the z_3-node representing the cofactor $f|_{z_1=1,z_2=0} = 1$ can be replaced with a terminal node for the value 1. This leads to the graph shown on the left of Figure 6.21. Finally, we may identify all terminal nodes with the same value, which yields the graph shown on the right of Figure 6.21. ■

Example 6.61. From Binary Decision Tree to OBDD

As another example, consider the switching function $f = (z_1 \wedge z_3) \vee (z_2 \wedge z_3)$. The upper part of Figure 6.22 on page 394 shows a binary decision tree for f. The subtree of the z_3 node on the right is constant and can be replaced by a terminal node. The three subtrees of the z_3-nodes for the cofactors $f|_{z_1=0,z_2=1}$, $f|_{z_1=1,z_2=0}$, and $f|_{z_1=1,z_2=1}$ are isomorphic and can thus be collapsed. This yields the decision graph shown in the middle part of Figure 6.22. But now the z_2-node for the cofactor $f|_{z_1=1}$ becomes redundant, as regardless whether $z_2 = 0$ or $z_2 = 1$, the same node will be reached. This permits removing this z_2-node and redirecting its incoming edge. This yields the binary decision diagram depicted in the lower part of Figure 6.22. ■

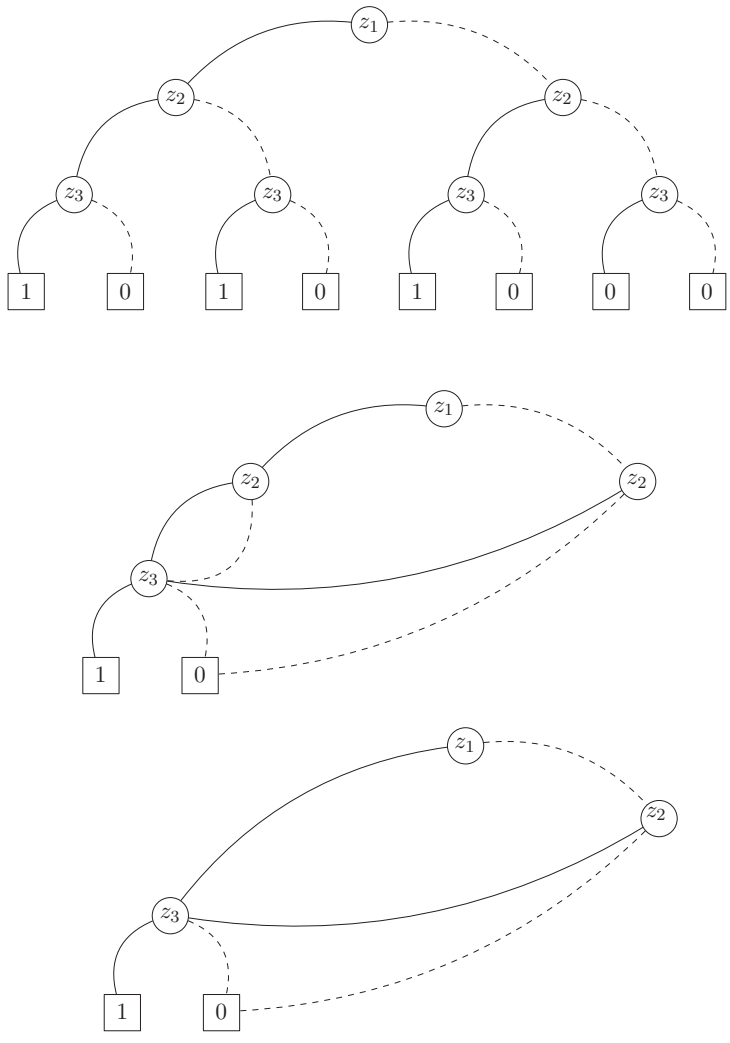


Figure 6.22: Binary decision diagrams for $f = (z_1 \wedge z_3) \vee (z_2 \wedge z_3)$.

Notation 6.62. Variable Ordering

Let *Var* be a finite set of variables. A *variable ordering* for *Var* denotes any tuple $\wp = (z_1, \ldots, z_m)$ such that $Var = \{z_1, \ldots, z_m\}$ and $z_i \neq z_j$ for $1 \leqslant i < j \leqslant m$. We write $<_\wp$ for the induced total order on *Var*. I.e., for $\wp = (z_1, \ldots, z_m)$ the binary relation $<_\wp$ on *Var* is given by $z_i <_\wp z_j$ if and only if $i < j$. We write $z_i \leqslant_\wp z_j$ iff either $z_i <_\wp z_j$ or $i = j$. ■

Definition 6.63. Ordered Binary Decision Diagram (OBDD)

Let $\wp$ be a variable ordering for *Var*. An *$\wp$-ordered binary decision diagram* ($\wp$-OBDD for short) is a tuple

$$\mathfrak{B} \;=\; (V, V_I, V_T, succ_0, succ_1, var, val, v_0)$$

consisting of

- a finite set V of nodes, disjointly partitioned into V_I and V_T where the nodes in V_I are called *inner nodes*, while the nodes in V_T are called *terminal nodes* or *drains*;
- successor functions $succ_0$, $succ_1 : V_I \to V$ that assign to each inner node v a 0-successor $succ_0(v) \in V$ and a 1-successor $succ_1(v) \in V$;
- a variable labeling function $var : V_I \to Var$ that assigns to each inner node v a variable $var(v) \in Var$;
- a value function $val : V_T \to \{0, 1\}$ that assigns to each drain a function value 0 or 1;
- a *root (node)* $v_0 \in V$.

Consistency of the variable labeling function with the variable ordering $\wp$ is required in the following sense. If $\wp = (z_1, \ldots, z_m)$, then for each inner node v: if $var(v) = z_i$ and $w \in \{succ_0(v), succ_1(v)\} \cap V_I$, then $var(w) = z_j$ for some $j > i$. Furthermore, it is required that each node $v \in V \setminus \{v_0\}$ has at least one predecessor, i.e., $v = succ_b(w)$ for some $w \in V$ and $b \in \{0, 1\}$. ■

The underlying digraph of a $\wp$-OBDD is obtained by using V as node set and establishing an edge from node v to w if and only if w is an successor of v, i.e., if $w \in \{succ_0(v), succ_1(v)\}$. The last condition in Definition 6.63, which states that each node of an OBDD, except for the root, has a predecessor, is equivalent to the condition that all nodes of an OBDD are reachable from the root. The *size* of an OBDD $\mathfrak{B}$, denoted $size(\mathfrak{B})$, is the number of nodes in $\mathfrak{B}$.

An equivalent formulation for the order consistency of $\mathfrak{B}$ (i.e., the condition stating that the variable of an inner node appears in $\wp$ before the variables of its successors, provided they are inner nodes) is the requirement that for each path $v_0\, v_1\, \ldots v_n$ in (the underlying graph of) $\mathfrak{B}$ we have $v_i \in V_I$ for $1 \leqslant i < n$ and

$$\mathit{var}(v_0) <_\wp \mathit{var}(v_1) <_\wp \ldots <_\wp \mathit{var}(v_n),$$

where for the drains we put $\mathit{var}(v) = \bot$ (undefined) and extend $<_\wp$ by $z <_\wp \bot$ for all variables $z \in \mathit{Var}$. That is, we regard $<_\wp$ as a total order on $\mathit{Var} \cup \{\bot\}$ and consider the variable labeling function as a function $\mathit{var} : V \to \mathit{Var} \cup \{\bot\}$. This yields that the underlying graph of an OBDD is acyclic. In particular, $v \neq \mathit{succ}_b(v)$ for all inner nodes v and $b \in \{0, 1\}$.

Examples of OBDDs are the binary decision trees and graphs shown in Figures 6.20, 6.21, and 6.22. All these OBDDs rely on the variable ordering $\wp = (z_1, z_2, z_3)$.

In the sequel, we often refer to an inner node v with $\mathit{var}(v) = z$ as a z-node. As the pictures for OBDDs suggest, we may think of the node set V of an OBDD to be partitioned into *levels*. Dealing with the ordering $\wp = (z_1, \ldots, z_m)$, then the z_i-nodes constitute the nodes at level i. The drains yields the bottom level, while the top level consists of the root node.

Definition 6.64. Semantics of OBDDs

Let $\mathfrak{B}$ be an $\wp$-OBDD as in Definition 6.63. The semantics $\mathfrak{B}$ is the switching function $f_\mathfrak{B}$ for *Var* where $f_\mathfrak{B}([z_1 = b_1, \ldots, z_m = b_m])$ is the value of the drain that will be reached when traversing the graph starting in the root v_0 and branching according to the evaluation $[z_1 = b_1, \ldots, z_m = b_m]$. That is, if the current node v is a z_i-node, then the traversal continues with the b_i-successor of v. ■

Definition 6.65. Sub-OBDD, Switching Function for the Nodes

Let $\mathfrak{B}$ be a $\wp$-OBDD. If v is a node in $\mathfrak{B}$, then the *sub-OBDD* induced by v, denoted $\mathfrak{B}_v$ arises from $\mathfrak{B}$ by declaring v as the root node and removing all nodes that are not reachable from v. The switching function for node, denoted f_v, is the switching function for *Var* that is given by the sub-OBDD $\mathfrak{B}_v$. ■

Clearly, at most the variables x with $\mathit{var}(v) \leqslant_\wp x$ can be essential for f_v, since none of the variables z with $z <_\wp \mathit{var}(v)$ appear in the sub-OBDD. Hence, f_v could also be viewed as a switching function for the whole variable set *Var* or as a switching function for the variables x with $\mathit{var}(v) \leqslant_\wp x$, but this is irrelevant here. In particular, if v is a z-node, then the order condition $z = \mathit{var}(v) <_\wp \mathit{var}(\mathit{succ}_b(v))$ yields that $f_{\mathit{succ}_b(v)}|_{z=\mathfrak{c}} = f_{\mathit{succ}_b(v)}$,

since variable z is not essential for $f_{succ_b(v)}$. But then:

$$f_v|_{z=0} \;=\; (\neg z \wedge f_{succ_0(v)})|_{z=0} \vee \underbrace{(z \wedge f_{succ_1(v)})|_{z=0}}_{=0}$$

$$= \; f_{succ_0(v)}|_{z=0} \;=\; f_{succ_0(v)}$$

and, similarly, $f_v|_{z=1} = f_{succ_1(v)}$. Thus, the *Shannon expansion* yields the following bottom-up characterization of the functions f_v:

Lemma 6.66. Bottom-up Characterization of the Functions f_v

Let $\mathfrak{B}$ be a $\wp$-OBDD. The switching functions f_v for the nodes $v \in V$ are given as follows:

- *If v is a drain, then f_v is the constant switching function with value val(v).*
- *If v is a z-node, then $f_v = (\neg z \wedge f_{succ_0(v)}) \wedge (z \wedge f_{succ_1(v)})$.*

Furthermore, $f_{\mathfrak{B}} = f_{v_0}$ for the root v_0 of $\mathfrak{B}$.

This yields that $f_v = f_{\mathfrak{B}}|_{z_1=b_1,\ldots,z_i=b_i}$ where $[z_1 = b_1, \ldots, z_i = b_i]$ is an evaluation which leads from the root v_0 of $\mathfrak{B}$ to node v. In fact, all concepts of OBDD-based approaches rely on the decomposition of switching functions into their cofactors. However, only the cofactors are relevant that arise from f by assigning values to the first i variables of $\wp$ for some i.

Notation 6.67. $\wp$-Consistent Cofactor

Let f be a switching function for *Var* and $\wp = (z_1, \ldots, z_m)$ a variable ordering for *Var*. A switching function f' for *Var* is called a *$\wp$-consistent cofactor* of f if there exists some $i \in \{0, 1, \ldots, m\}$ such that $f' = f|_{z_1=b_1,\ldots,z_i=b_i}$. (Including the case $i = 0$ means that we regard f as a cofactor of itself.) ■

For instance, if $f = z_1 \wedge (z_2 \vee \neg z_3)$ and $\wp = (z_1, z_2, z_3)$, then the $\wp$-consistent cofactors of f are the switching functions f, $f|_{z_1=1} = z_2 \vee \neg z_3$, $f|_{z_1=1,z_2=0} = \neg z_3$ and the constants 0 and 1. The cofactors $f|_{z_3=0} = z_1$ and $f|_{z_2=0} = z_1 \wedge \neg z_3$ are not $\wp$-consistent. Since it is possible that some cofactors of f arise by several evaluations, it is possible that cofactors $f|_{z_{i_1}=b_1,\ldots,z_{i_k}=b_k}$ are $\wp$-consistent for the variable ordering $\wp = (z_1, \ldots, z_m)$, even if $(z_{i_1}, \ldots, z_{i_k})$ is *not* a permutation of $(z_1, \ldots, z_k)$. For example, for $f = z_1 \wedge (z_2 \vee \neg z_3)$ and $\wp = (z_1, z_2, z_3)$ the cofactor $f|_{z_2=0,z_3=1}$ is $\wp$-consistent as it agrees with the cofactors $f_{z_1=0}$ or $f|_{z_1=1,z_2=0,z_3=1}$. (They all agree with the constant function 0.)

The observation made above can now be rephrased as follows:

Lemma 6.68. Nodes in OBDDs and $\wp$-Consistent Cofactors

For each node v of an $\wp$-OBDD $\mathfrak{B}$, the switching function f_v is a $\wp$-consistent cofactor of $f_{\mathfrak{B}}$. Vice versa, for each $\wp$-consistent cofactor f' of $f_{\mathfrak{B}}$ there is at least one node v in $\mathfrak{B}$ such that $f_v = f'$.

However, given a $\wp$-OBDD $\mathfrak{B}$ and a $\wp$-consistent cofactor f' of $f_{\mathfrak{B}}$ there could be more than one node in $\mathfrak{B}$ representing f'. This, for instance, applies to the binary decision tree shown in Figure 6.20 on page 385, viewed as $\wp$-OBDD for $\wp = (z_1, z_2, z_3)$, where we have $f_{\mathfrak{B}} = z_1 \wedge (\neg z_2 \vee z_3)$ and the $\wp$-consistent cofactors represented by the nodes in the left subtree of the root agree as we have

$$f|_{z_1=0} \;=\; f|_{z_1=0,z_2=b} \;=\; f|_{z_1=0,z_2=b,z_3=\mathfrak{c}} \;=\; 0$$

for all b, $\mathfrak{c} \in \{0,1\}$. For the $\wp$-OBDD shown on the left of Figure 6.21 on page 393, all inner nodes represent different switching functions. However, the two drains with value 0 represent the same cofactors of $f_{\mathfrak{B}}$. The same holds for the two drains with value 1. However, in the $\wp$-OBDD shown on the right of Figure 6.21, every $\wp$-conistent cofactor is represented by a single node. In this sense, this $\wp$-OBDD is free of redundancies, and therefore called reduced:

Definition 6.69. Reduced OBDD

Let $\mathfrak{B}$ be a $\wp$-OBDD. $\mathfrak{B}$ is called *reduced* if for every pair (v, w) of nodes in $\mathfrak{B}$: $v \neq w$ implies $f_v \neq f_w$. Let $\wp$-ROBDD denote a reduced $\wp$-OBDD. ■

Thus, in reduced $\wp$-OBDDs any $\wp$-consistent cofactor is represented by *exactly one* node. This is the key property to prove that reduced OBDDs yield a universal and canonical data structure for switching functions. Universality means that any switching function can be represented by an OBDD. Canonicity means that any two $\wp$-OBDDs for the same function agree up to isomorphism, i.e., renaming of the nodes.

Theorem 6.70. Universality and Canonicity of ROBDDs

Let Var be a finite set of Boolean variables and $\wp$ a variable ordering for Var. Then:

(a) For each switching function f for Var there exists a $\wp$-ROBDD $\mathfrak{B}$ with $f_{\mathfrak{B}} = f$.

(b) Given two $\wp$-ROBDDs $\mathfrak{B}$ and $\mathfrak{C}$ with $f_{\mathfrak{B}} = f_{\mathfrak{C}}$, then $\mathfrak{B}$ and $\mathfrak{C}$ are isomorphic, i.e., agree up to renaming of the nodes.

Proof: ad (a). Clearly, the constant functions 0 and 1 can be represented by a $\wp$-ROBDD consisting of a single drain. Given a nonconstant switching function f for *Var* and a variable ordering $\wp$ for *Var*, we construct a reduced $\wp$-OBDD $\mathfrak{B}$ for f as follows. Let V be the set of $\wp$-consistent cofactors of f. The root of $\mathfrak{B}$ is f. The constant cofactors are the drains with the obvious values. For $f' \in V$, $f' \notin \{0,1\}$, let

$$\mathit{var}(f') \;=\; \min\{\, z \in \mathit{Var} \mid z \text{ is essential for } f' \,\}$$

be the first essential variable where the minimum is taken according to the total order $<_\wp$ induced by $\wp$. (We use here the trivial fact that any nonconstant switching function has at least one essential variable.) The successor functions are given by

$$\mathit{succ}_0(f') \;=\; f'|_{z=0}, \quad \mathit{succ}_1(f') = f'|_{z=1}$$

where $z = \mathit{var}(f')$. The definition of $\mathit{var}(\cdot)$ yields that $\mathfrak{B}$ is a $\wp$-OBDD. By the Shannon expansion we get that the semantics of $f' \in V$ (i.e., the switching function of f' as a node of $\mathfrak{B}$) is f'. In particular, this yields that $f_{\mathfrak{B}} = f$ (the function for the root f) and the reducedness of $\mathfrak{B}$ (as any two nodes represent different cofactors of f).

To prove the statement in (b), it suffices to show that any reduced $\wp$-OBDD $\mathfrak{C}$ with $f_{\mathfrak{C}} = f$ is isomorphic to the $\wp$-ROBDD $\mathfrak{B}$ constructed above. Let $V^{\mathfrak{C}}$ be the node set of $\mathfrak{C}$, $v_0^{\mathfrak{C}}$ the root of $\mathfrak{C}$, $\mathit{var}^{\mathfrak{C}}$ the variable labeling function, and $\mathit{succ}_0^{\mathfrak{C}}$, $\mathit{succ}_1^{\mathfrak{C}}$ the successor functions in $\mathfrak{C}$. Let function $\imath : V^{\mathfrak{C}} \to V$ be given by $\imath(v) = f_v$. (Recall that the functions f_v are $\wp$-consistent cofactors of $f_{\mathfrak{C}} = f$. This ensures that $f_v \in V$.) Since $\mathfrak{C}$ is reduced, $\imath$ is a bijection. It remains to show that $\imath$ preserves the variable labeling of inner nodes and maps the successors of an inner node v of $\mathfrak{C}$ to the successors of $f_v = \imath(v)$ in $\mathfrak{B}$.

Let v be an inner node of $\mathfrak{C}$, say a z-node, and let w_0 and w_1 be the 0- and 1-successors of v in $\mathfrak{C}$. Then, the cofactor $f_v|_{z=0}$ agrees with f_{w_0}, and similarly, $f_{w_1} = f_v|_{z=1}$. (This holds in any OBDD.) As $\mathfrak{C}$ is reduced, f_v is nonconstant (since otherwise $f_v = f_{w_0} = f_{w_1}$). Variable z must be the first essential variable of f_v according to $<_\wp$, i.e., $z = \mathit{var}(f_v)$. Let us see why. Let $y = \mathit{var}(f_v)$. The assumption $z <_\wp y$ yields that z is not essential for f_v, and therefore $f_{w_0} = f_v|_{z=0} = f = f_v|_{z=1} = f_{w_1}$. But then w_0, w_1 and v represent the same function. Since $w_0 \neq v$ and $w_1 \neq v$, this contradicts the assumption that $\mathfrak{C}$ is reduced. The assumption $y <_\wp z$ is also impossible since then no y-node would appear in the sub-OBDD $\mathfrak{C}_v$, which is impossible as $y = \mathit{var}(f_v)$ is essential for f_v by definition.

But then $\mathit{var}(\imath(v)) = z = \mathit{var}^{\mathfrak{C}}(v)$ and, for $b \in \{0,1\}$:

$$\mathit{succ}_b(\imath(v)) \;=\; f_v|_{z=b} \;=\; f_{\mathit{succ}_b^{\mathfrak{C}}(v)} \;=\; \imath(\mathit{succ}_b^{\mathfrak{C}}(v))$$

Hence, $\imath$ is an isomorphism. ∎

Theorem 6.70 permits speaking about "the $\wp$-ROBDD" for a given switching function f for *Var*. The $\wp$-ROBDD-size of f denotes the size (i.e., number of nodes) in the $\wp$-ROBDD for f.

Corollary 6.71. Minimality of Reduced OBDDs

Let $\mathfrak{B}$ be a $\wp$-OBDD for f. Then, $\mathfrak{B}$ is reduced if and only if size$(\mathfrak{B}) \leqslant$ size$(\mathfrak{C})$ for each $\wp$-OBDD $\mathfrak{C}$ for f.

Proof: This follows from the facts that (i) each $\wp$-consistent confactor of f is represented in any $\wp$-OBDD $\mathfrak{C}$ for f by at least one node, and (ii) a $\wp$-OBDD $\mathfrak{B}$ for f is reduced if and only if the nodes of $\mathfrak{B}$ stand in one-to-one-correspondence to the $\wp$-consistent cofactors of f. ■

Reduction rules. When the variable ordering $\wp$ for *Var* is fixed, then reduced $\wp$-OBDDs provide unique representations of switching functions for *Var*. (Of course, uniqueness is up to isomorphism.) Although reducedness is a global condition for an OBDD, there are two simple local *reduction rules* (see Figure 6.23), which can successively be applied to transform a given nonreduced $\wp$-OBDD into an equivalent $\wp$-ROBDD.

Elimination rule: If v is an inner node of $\mathfrak{B}$ with $succ_0(v) = succ_1(v) = w$, then remove v and redirect all incoming edges $u \to v$ to w.

Isomorphism rule: If v, w are nodes in $\mathfrak{B}$ with $v \neq w$ and either v, w are drains with $val(v) = val(w)$ or v, w are inner nodes with

$$\langle var(v), succ_1(v), succ_0(v)\rangle \;=\; \langle var(w), succ_1(w), succ_0(w)\rangle,$$

then remove node v and redirect all incoming edges $u \to v$ to node w.

Both rules delete node v. The redirection of the incoming edges $u \to v$ to node w means the replacement of the edges $u \to v$ which $u \to w$. Formally, this means that we deal with the modified successor functions given by

$$succ'_b(u) \;=\; \begin{cases} succ_b(u) & \text{if } succ_b(u) \neq v \\ w & \text{if } succ_b(u) = v \end{cases}$$

for $b = 0, 1$. The transformations described in Examples 6.60 and 6.61 rely on the application of the isomorphism and elimination rule.

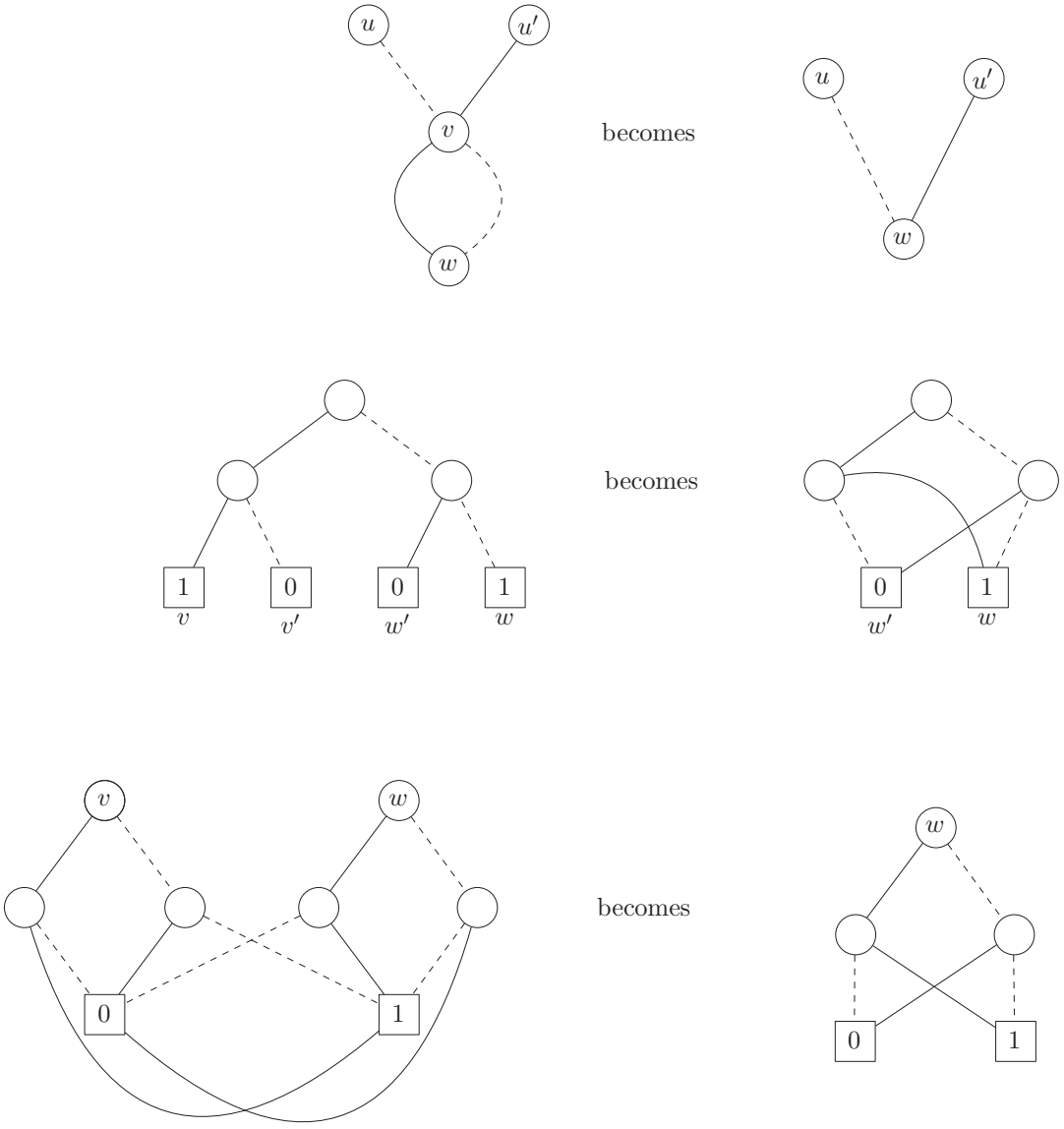


Figure 6.23: Reduction rules for OBDDs.

Both reduction rules are sound in the sense that they do not affect the semantics, i.e., if $\mathfrak{C}$ arises from a $\wp$-OBDD $\mathfrak{B}$ by applying the elimination or isomorphism rule, then $\mathfrak{C}$ is again a $\wp$-OBDD and $f_{\mathfrak{B}} = f_{\mathfrak{C}}$. This is due to the fact that both rules simply collapse two nodes v and w with $f_v = f_w$. For the elimination rule applied to a z-node v with $w = succ_0(v) = succ_1(v)$, we have

$$f_v \;=\; (\neg z \wedge f_{succ_0(v)}) \wedge (z \wedge f_{succ_1(v)}) \;=\; (\neg z \wedge f_w) \wedge (z \wedge f_w) \;=\; f_w.$$

Similarly, if the isomorphism rule is applied to z-nodes v and w then

$$f_v \;=\; (\neg z \wedge f_{succ_0(v)}) \wedge (z \wedge f_{succ_1(v)}) \;=\; (\neg z \wedge f_{succ_0(w)}) \wedge (z \wedge f_{succ_1(w)}) \;=\; f_w.$$

Since the application of the reduction rules decreases the number of nodes, the process to generate an equivalent $\wp$-OBDD by applying the reduction rules as long as possible always terminates. In fact, the resulting OBDD is reduced:

Theorem 6.72. Completeness of Reduction Rules

$\wp$-OBDD $\mathfrak{B}$ is reduced if and only if no reduction rule is applicable to $\mathfrak{B}$.

Proof: $\Rightarrow$: The applicability of a reduction rule implies the existence of at least two nodes representing the same switching function. Thus, if $\mathfrak{B}$ is reduced, then no reduction rule is applicable.

$\Leftarrow$: We prove the other direction by induction on the number of variables. More precisely, suppose that we are given a $\wp$-OBDD $\mathfrak{B}$ for the variable ordering $\wp = (z_1, \ldots, z_m)$ such that neither the elimination nor the isomorphism rule is applicable and show by induction on i that

$$f_v \neq f_w \text{ for all nodes } v,\, w \in V_i \text{ where } v \neq w.$$

Here, V_i denotes the set of all nodes $v \in V$ on level i or lower. Formally, V_i is the set of all nodes v in $\mathfrak{B}$ such that $z_i \leqslant_\wp var(v)$. Recall that $var(v) = \perp$ (undefined) for each drain v and that $z <_\wp \perp$ for all variables z.

We start the induction with the bottom level $i = m + 1$. The statement $f_v \neq f_w$ for all drains v, w where $v \neq w$ is trivial since the nonapplicability of the isomorphism rule yields that there is at most one drain with value 0 and at most one drain with value 1. In the induction step $i + 1 \Longrightarrow i$ $(m \geqslant i \geqslant 0)$ we suppose that $f_v \neq f_w$ for all nodes $v, w \in V_{i+1}$ with $v \neq w$ (induction hypothesis). Suppose there are two nodes v, $w \in V_i$ with $v \neq w$ and $f_v = f_w$. At least one of the nodes v or w must be on level i. Say $v \in V_i \setminus V_{i+1}$. Then, $var(v) = z_i$.

Let us first suppose that $w \in V_{i+1}$. Then, either w is a drain or a z_j-node for some $j > i$. In either case, variable z_i is not essential for $f_v = f_w$. As v is a z_i-node this yields that f_v agrees with the switching functions f_{v_0} and f_{v_1} of its successors $v_0 = \mathit{succ}_0(v)$ and $v_1 = \mathit{succ}_1(v)$. But then $v_0, v_1 \in V_{i+1}$ and $f_{v_0} = f_{v_1}$. The induction hypothesis yields that $v_0 = v_1$. But then the elimination rule would be applicable. Contradiction.

Suppose now that w is a z_i-node too. Let $v_0 = \mathit{succ}_0(v)$, $v_1 = \mathit{succ}_1(v)$ and $w_0 = \mathit{succ}_0(w)$, $w_1 = \mathit{succ}_1(w)$. The assumption $f_v = f_w$ yields that

$$f_{v_0} \;=\; f_v|_{z_i=0} \;=\; f_w|_{z_i=0} \;=\; f_{w_0}$$

and $f_{v_1} = f_{w_1}$. As $v_0, v_1, w_0, w_1 \in V_{i+1}$ the induction hypothesis yields that $v_0 = w_0$ and $v_1 = w_1$. But then the ismorphism rule is applicable. Contradiction. ■

Theorem 6.70 suggests a *reduction algorithm* which takes as input a nonreduced $\wp$-OBDD $\mathfrak{B}$ and constructs an equivalent $\wp$-OBDD by applying the reduction rules as long as possible. According to the inductive proof of the completeness of the reduction rules in Theorem 6.72, this technique is complete if the candidate nodes for the reduction rules are considered in a bottom-up manner. That is, initially we identify all drains with the same value. Then, for the levels $m, m-1, \ldots, 1$ (in this order) we apply the elimination and isomorphism rule. At level i, we first remove all nodes with identical successors (elimination rule) and then check the pairs of z_i-nodes where the isomorphism rule is applicable. To support the isomorphism rule a bucket technique can be used that groups together (1) all z_i-nodes with the same 0-successor and (2) splits all buckets consisting of z_i-nodes with the same 0-successor into buckets consisting of z_i-nodes with exactly the same successors. Then, application of the isomorphism rule simply means that the nodes in the buckets resulting from step (2) have to be collapsed into a single node. The time complexity of this algorithm is in $\mathcal{O}(\mathit{size}(\mathfrak{B}))$. In particular, given two $\wp$-OBDDs $\mathfrak{B}$ and $\mathfrak{C}$, the equivalence problem "Does $f_{\mathfrak{B}} = f_{\mathfrak{C}}$ hold?" can be solved by applying the reduction algorithm to both and then checking isomorphism for the reduced $\wp$-ROBDDs (see Exercise 6.12 on page 436). We will see later that with tricky implementation techniques, which integrate the steps of the reduction algorithm in the synthesis algorithms for ROBDDs and thus ensure that at any time the decision graph is reduced, the equivalence problem for ROBDDs can even be solved in constant time.

The Variable Ordering Problem The result stating the canonicity of reduced OBDDs crucially depends on the fact that the variable ordering $\wp$ is assumed to be fixed. Varying the variable ordering can lead to totally different ROBDDs, possibly ranging from ROBDDs of linear size to ROBDDs of exponential size. The results established before yield that the size (i.e., number of nodes) in the $\wp$-ROBDD for a switching function f agrees

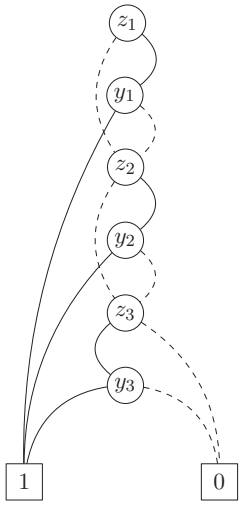


Figure 6.24: ROBDD for the function $f_3 = (z_1 \wedge y_1) \vee (z_2 \wedge y_2) \vee (z_3 \wedge y_3)$ for the variable ordering $\wp = (z_1, y_1, z_2, y_2, z_3, y_3)$.

with the number of $\wp$-consistent cofactors of f. Thus, reasoning about the memory requirements of ROBDD-based approaches relies on counting the number of order-consistent cofactors.

Example 6.73. A Function with Small and Exponential-Size ROBDDs

To illustrate how the size of ROBDDs can be determined by analyzing the cofactors we consider a simple switching function which has both ROBDDs of linear size and ROBDDs with exponentially many nodes. Let $m \geqslant 1$ and

$$f_m = (z_1 \wedge y_1) \vee (z_2 \wedge y_2) \vee \ldots \vee (z_m \wedge y_m).$$

For the variable ordering $\wp = (z_m, y_m, z_{m-1}, y_{m-1}, \ldots, z_1, y_1)$, the $\wp$-ROBDD for f_m has $2m + 2$ nodes, while $\Omega(2^m)$ nodes are required for the ordering $\wp' = (z_1, z_2, \ldots, z_m, y_1, \ldots, y_m)$. Figures 6.25 and 6.24 show the ROBDDs for the linear-size and exponential-size variable orderings for m=3. We first consider the ordering $\wp$ which groups the variables z_i and y_i that appear in the same clause. In fact, the variable ordering $\wp$ is optimal for f_m as the $\wp$-ROBDD for f_m contains one node for each variable. (This is the best we can hope to have as all $2n$ variables are essential for f_m and must appear in any ROBDD for

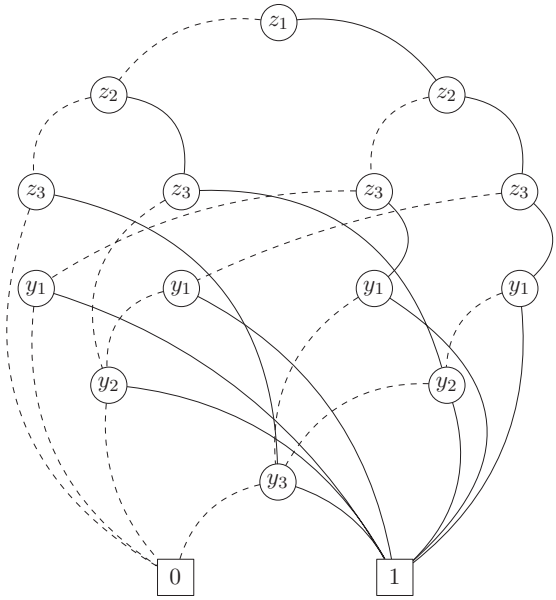


Figure 6.25: ROBDD for the function $f_3 = (z_1 \wedge y_1) \vee (z_2 \wedge y_2) \vee (z_3 \wedge y_3)$ for the variable ordering $\wp = (z_1, z_2, z_3, y_1, y_2, y_3)$.

f_m.) Note that for $1 \leqslant i \leqslant m$:

$$f_m|_{z_m=a_m, z_m=b_m, \ldots, z_i=a_i, z_i=b_i} = \begin{cases} 1 & \text{if } a_j = b_j = 1 \\ & \text{for some } j \in \{i, \ldots, m\} \\ f_{i-1} & \text{otherwise} \end{cases}$$

$$f_m|_{z_m=a_m, z_m=b_m, \ldots, z_{i+1}=a_{i+1}, z_{i+1}=b_{i+1}, z_i=a_i} \in \{\, 1,\ y_i \vee f_{i-1} \,\}$$

where $f_0 = 0$. Hence, the $\wp$-ROBDD for f_m has exactly one z_i-node representing the function f_i, exactly one y_i-node for the function $y_i \vee f_{i-1}$ (for $1 \leqslant i \leqslant m$), and two drains.

To see why the variable ordering $\wp'$ leads to a $\wp'$-ROBDD of exponential size, we consider the $\wp'$-consistent cofactors

$$f_{\overline{b}} = f_m|_{z_1=b_1, \ldots, z_m=b_m} = \bigvee_{i \in I_{\overline{b}}} y_i$$

where $\overline{b} = (b_1, \ldots, b_n)$ and $I_{\overline{b}} = \{i \in \{1, \ldots, m\} \mid b_i = 1$. The set of essential variables of $f_{\overline{b}}$ is $\{y_i \mid i \in I_{\overline{b}}\}$. As f $\overline{b}$, $\overline{\mathfrak{c}} \in \{0,1\}^m$, $\overline{b} \neq \overline{\mathfrak{c}}$ the index sets $I_{\overline{b}}$ and $I_{\overline{\mathfrak{c}}}$ are different, the $f_{\overline{b}}$ and $f_{\overline{\mathfrak{c}}}$ have different essential variables. Therefore, $f_{\overline{b}} \neq f_{\overline{\mathfrak{c}}}$ if $\overline{b} \neq \overline{\mathfrak{c}}$. But then the number of $\wp'$-consistent cofactors is at least 2^m. This yields that the $\wp'$-ROBDD for f_m has at most 2^m nodes. ■

Since for many switching functions the ROBDD sizes for different variable ordering can vary enormously, the efficiency of BDD-based computations crucially relies on the use of techniques that improve a given variable ordering. Although the problem to find an optimal variable ordering is known to be computationally hard (already the problem to decide whether a given variable ordering is optimal is NP-hard [56, 386]), there are several efficient heuristics for improving the current ordering. Most promiment is the so-called *sifting algorithm* [358] which relies on a local search for the best position for each variable, when the ordering of the other variables is supposed to be fixed. Explanations on such variable reordering algorithms and further discussions on the variable ordering problem are beyond the scope of this monograph. We refer the interested reader to the textbooks [292, 418] and conclude the comments on the influence of the variable ordering by the remark that there are also types of switching functions with ROBDDs of polynomial size under all variable orderings and types of switching functions where any variable ordering leads to a ROBDD of exponential size. An example of the latter is the middle bit of the *multiplication function* [71]. Examples of switching functions where each variable ordering leads to a ROBDD of at most quadratic size are *symmetric functions*. These are switching functions where the function values just depend on the number of variables that are assigned to 1. Stated differently, $f \in \mathit{Eval}(z_1, \ldots, z_m)$ is symmetric if and only if

$$f([z_1 = b_1, \ldots, z_m = b_m]) \;=\; f([z_1 = b_{i_1}, \ldots, z_m = b_{i_m}])$$

for each permutation $(i_1, \ldots, i_m)$ of $(1, \ldots, m)$. Examples of symmetric functions for $\mathit{Var} = \{z_1, \ldots, z_m\}$ are $z_1 \vee z_2 \vee \ldots \vee z_m$, $z_1 \wedge z_2 \wedge \ldots \wedge z_m$, the parity function $z_1 \oplus z_2 \oplus \ldots \oplus z_m$ (which returns 1 iff the number of variables that are assigned to 1 is odd), and the majority function (which returns 1 iff the number of variables that are assigned to 1 is greater than the number of variables that are assigned to 0). The ROBDDs for symmetric functions have the same topological structure for all variable orderings. This follows from the fact that the $\wp$-ROBDD for a symmetric function can be transformed into the $\wp'$-ROBDD by just modifying the variable labeling function.

Lemma 6.74. ROBDD-Size for Symmetric Functions

If f is a symmetric function with m essential variables, then for each variable ordering $\wp$ the $\wp$-ROBDD has size $\mathcal{O}(m^2)$.

Proof: Given a symmetric function f for m variables and a variable ordering $\wp$, say $\wp = (z_1, \ldots, z_m)$, then the $\wp$-consistent cofactors $f|_{z_1=b_1,\ldots,z_i=b_i}$ and $f|_{z_1=\mathfrak{c}_1,\ldots,z_i=\mathfrak{c}_i}$ agree for all bit tuples $(b_1, \ldots, b_i)$ and $(\mathfrak{c}_1, \ldots, \mathfrak{c}_i)$ that contain the same number of 1's. Thus, there are at most $i + 1$ different $\wp$-consistent cofactors of f that arise by assigning values to the first i variables. Hence, the total number of $\wp$-consistent cofactors is bounded above by $\sum_{i=0}^{m}(i + 1) = \mathcal{O}(m^2)$. ∎

ROBDDs vs. CNF/DNF Both the parity and the majority function provide examples for switching functions with small ROBDD representations, while any representation of them by conjunctive or disjunctive normal forms (CNF, DNF) requires formulae of exponential length. Vice versa, there are also switching functions with short conjunctive or disjunctive normal forms, while the ROBDD under any variable ordering has exponential length (see, e.g., [418]). In fact, ROBDDs yield a totally different picture for complexity theoretic considerations than CNF or DNF. For example, given a CNF representation for a switching function f, the task to generate a CNF for $\neg f$ is expensive, as f might be expressible by a CNF of polynomial length, while any CNF for $\neg f$ has at least exponentially many clauses. For ROBDDs, however, negation is trivial as we may simply swap the values of the drains. In particular, for any variable ordering $\wp$, the $\wp$-ROBDDs for f and $\neg f$ have the same size. For another example, regard the satisfiability problem, which is known to be NP-complete for CNF, but again trivial for ROBDDs, since $f \neq 0$ if and only if the $\wp$-ROBDD for f does not contain a 0-drain. Similarly, the question whether two CNFs are equivalent is computationally hard (coNP-complete), but can be solved for $\wp$-ROBDDs $\mathfrak{B}$ and $\mathfrak{C}$ by checking isomorphism. The latter can be performed by a simultaneous traversal of the $\wp$-OBDDs in time linear in the sizes of $\mathfrak{B}$ and $\mathfrak{C}$. See Exercise 6.12, page 436. Note that these results do not contradict the complexity theoretic lower bounds, since "linear time" for a ROBDD-based algorithm means linear in the size of the input ROBDDs, which could be exponentially larger than equivalent input formulae (e.g., CNF).

6.7.4 Implementation of ROBDD-Based Algorithms

The efficiency of ROBDD-based algorithms to manipulate switching functions crucially relies on appropriate implementation techniques. In fact, with tricky implementation techniques, equivalence checking for ROBDDs can even be realized in *constant time*. In the sequel, we will explain the main ideas of such techniques, which provide the basis for most BDD packages and serve as a platform for an efficient realization of synthesis algorithms on ROBDDs. The purpose of *synthesis algorithms* is to construct a $\wp$-ROBDD for a function $f_1 \mathit{op} f_2$ (where *op* is a Boolean connective such as disjunction, conjunction, implication, etc.), when $\wp$-ROBDDs for f_1 and f_2 are given. Recall that the symbolic realization of the CTL model-checking procedure relies on such synthesis operations.

The idea, originally proposed in [301], is to use a single reduced decision graph with one global variable ordering $\wp$ to represent several switching functions, rather than using separate $\wp$-ROBDDs for each of the switching functions. All computations on these decision graphs are interleaved with the reduction rules to guarantee redundance-freedom at any time. Thus, the comparison of two represented functions simply requires checking equality of the nodes for them, rather than analyzing their sub-OBDDs.

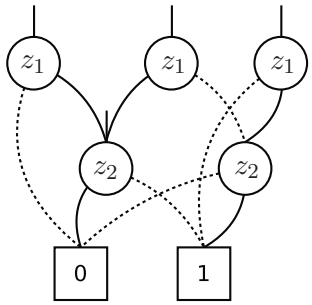


Figure 6.26: Example of a shared OBDD.

We start with the formal definition of a shared $\wp$-OBDD which is the same as a $\wp$-ROBDD, the only difference being that we can have more than more root node.

Definition 6.75. Shared OBDD

Let *Var* be a finite set of Boolean variables and $\wp$ a variable ordering for *Var*. A shared $\wp$-OBDD ($\wp$-SOBDD for short) is a tuple $\overline{\mathfrak{B}} = (V, V_I, V_T, succ_0, succ_1, var, val, \overline{v}_0)$ where V, V_I, V_T, $succ_0$, $succ_1$, *var*, and *val* are as in $\wp$-OBDDs (see Definition 6.63 on page 395). The last component is a tuple $\overline{v}_0 = (v_0^1, \ldots, v_0^k)$ of roots. The requirements are as in $\wp$-ROBDDs, i.e., for all nodes v, $w \in V$, (1) $var(v) <_\wp var(succ_b(v))$ if $v \in V_I$ and $b \in \{0,1\}$ and (2) $v \neq w$ implies $f_v \neq f_w$, where the switching function f_v for the nodes $v \in V$ is defined as for OBDDs. ■

Figure 6.26 shows a shared OBDD with four root nodes that represent the functions $z_1 \wedge \neg z_2$, $\neg z_2$, $z_1 \oplus z_2$ and $\neg z_1 \vee z_2$.

If v is a node in a $\wp$-SOBDD $\overline{\mathfrak{B}}$, then the sub-OBDD $\overline{\mathfrak{B}}_v$ is the $\wp$-ROBDD that results from $\overline{\mathfrak{B}}$ by removing all nodes that are not reachable from v and declaring v to be the root node. In fact, $\overline{\mathfrak{B}}_v$ is the $\wp$-ROBDD for f_v, and thus, the size N_v of $\overline{\mathfrak{B}}_v$ is with the $\wp$-ROBDD size of f_v. Thus, an alternative view of an SOBDD is the combination of several ROBDDs for the same variable ordering $\wp$ by sharing nodes for common $\wp$-consistent cofactors. In particular, an SOBDD has exactly two drains for the constant functions 0 and 1 (where we ignore the pathological case of an SOBDD with a single root node representing a constant function). Thus, if $f_1, \ldots, f_k$ are the functions for the root nodes $v_0^1, \ldots, v_0^k$ of $\overline{\mathfrak{B}}$, then the size (i.e., total number of nodes) of $\overline{\mathfrak{B}}$ is often smaller than but at most $N_{f_1} + \ldots + N_{f_k}$ where N_f denotes the $\wp$-ROBDD size of f.

For the symbolic representation of a transition system by means of switching functions $\Delta(\overline{x}, \overline{x}')$ for the transition relation and switching functions $f_a(\overline{x})$, $a \in AP$, for the satisfaction sets of the atomic propositions (see page 386), one might use a shared OBDD with

root nodes for Δ and the f_a's. As we mentioned before, the chosen variable ordering $\wp$ can be crucial for the size of the SOBDD representing a transition system. Experimental studies have shown that typically good variable orderings are obtained when grouping together the unprimed variables x_i and their copies x_i'. Later we will give some formal arguments why such interleaved variable orderings, like $\wp = (x_1, x_1', \ldots, x_n, x_n')$, are advantageous.

To perform the CTL model-checking procedure in a symbolic way, the shared OBDD $\overline{\mathfrak{B}}$ with root nodes for Δ and the f_a's has to be extended by new root nodes representing the characteristic functions of the satisfaction sets $Sat(\Psi)$ for the state subformulae Ψ of the CTL formula Φ to be checked. For instance, if $\Phi = a \wedge \neg b$ with atomic propositions a, b, then we first have to insert a root node for the characteristic function $f_{\neg b} = \neg f_b$ for $Sat(\neg b)$ and then a root node for the switching function $f_a \wedge f_{\neg b}$. The treatment of formulae of the form, e.g., $\exists\Diamond\Psi$ or $\exists\Box\Psi$ by means of Algorithms 20 or 21, requires creating additional root nodes for the functions f_i representing the current approximations of satisfaction sets. Of course, adding a new root node for some switching function f also means that we have to add nodes for all order-consistent cofactors of f that are not yet represented by a node in $\overline{\mathfrak{B}}$.

To support such dynamic changes of the set of switching functions to be represented, the realization of shared OBDDs typically relies on the use of two tables: the *unique table*, which contains the relevant information about the nodes and serves to keep the diagram reduced during the execution of synthesis algorithms, and a *computed table*, which is needed for efficiency reasons. Let us first explain the ideas behind the use of the unique table. The implementation of synthesis algorithms and the use of the computed table will be explained later.

The Unique Table The entries of the unique table are triples of the form

$$\mathit{info}(v) \;=\; \langle \mathit{var}(v), \mathit{succ}_0(v), \mathit{succ}_0(v) \rangle$$

for each inner node v. Note that these info-triples contain the relevant information which is necessary for the applicability of the isomorphism rule. Accessing the unique table is supported by a *find_or_add*-operation which takes as argument a triple $\langle z, v_1, v_0 \rangle$ consisting of a variable z and two nodes v_1 and v_0 with $v_1 \neq v_0$. The task of the *find_or_add*-operation is to check whether there exists a node v in the shared OBDD $\overline{\mathfrak{B}}$ such that $\mathit{info}(v) = \langle z, v_1, v_0 \rangle$. If so, then it returns node v, otherwise it creates a new z-node v with 1-successor v_1 and 0-successor v_0, and makes a corresponding entry in the unique table. Thus, the *find_or_add* operation can be viewed as the SOBDD realization of the *isomorphism rule*. In most BDD packages, the unique table is organized using appropriate hashing techniques. We skip such details here and assume constant expected time to access the into-triple for any node, and to perform the *find_or_add*-operation.

Boolean Operators Let us now consider how *synthesis algorithms* can be realized on SOBDDs, using the unique table. An elegant, but also very efficient way is to support a ternary operator, called ITE for "if-then-else", that covers all Boolean connectives. The *ITE operator* takes as arguments three switching functions g, f_1, f_2 and composes them according to "if g then f_1 else f_2". Formally:

$$ITE(g, f_1, f_2) \;=\; (g \wedge f_1) \;\vee\; (\neg\, g \wedge f_2)$$

For the special case where g is constant we have $ITE(0, f_1, f_2) = f_2$ and $ITE(1, f_1, f_2) = f_1$. The ITE operator fits very well with the representation of the SOBDD nodes in the unique table by their info-triples as we have:

$$f_v \;=\; ITE(\, z,\, f_{succ_1(v)},\, f_{succ_0(v)}\,).$$

The negation operator is obtained by $\neg f = ITE(f, 0, 1)$. Also all other Boolean connectives can be expressed by the ITE-operator. For example:

$$\begin{array}{lcl} f_1 \vee f_2 & = & ITE(f_1, 1, f_2) \\ f_1 \wedge f_2 & = & ITE(f_1, f_2, 0) \\ f_1 \oplus f_2 & = & ITE(f_1, \neg f_2, f_2) \;=\; ITE(f_1, ITE(f_2, 0, 1), f_2) \\ f_1 \rightarrow f_2 & = & ITE(f_1, f_2, 1) \end{array}$$

The realization of the ITE operator on an SOBDD $\overline{\mathfrak{B}}$ requires a procedure that takes as input three nodes u, v_1, v_2 of $\overline{\mathfrak{B}}$ and returns a possibly new node w such that $f_w = ITE(f_u, f_{v_1}, f_{v_2})$, by reusing existing nodes whenever possible and adding new nodes to $\overline{\mathfrak{B}}$ if necessary. For this, the sub-OBDDs for the input nodes u, v_1, and v_2 are traversed simultaneously in a top-down fashion, while the synthesis of the sub-ROBDD for w (and generation of new nodes) is performed in a bottom-up manner. This method relies on the following observation.

Lemma 6.76. Cofactors of $ITE(\cdot)$

If g, f_1, f_2 are switching functions for Var, $z \in Var$ and $b \in \{0, 1\}$, then

$$ITE(g, f_1, f_2)|_{z=b} \;=\; ITE(g|_{z=b}, f_1|_{z=b}, f_2|_{z=b}).$$

Proof: For simplicity, let us assume that g, f_1, f_2 are switching functions for the same variable set $Var = \{z, y_1, \ldots, y_m\}$. This, in fact, is no proper restriction as we may simply take the union of the variable sets of g, f_1 and f_2 and regard all three functions as switching functions for the resulting variable sets. Let $(a, \overline{\mathfrak{c}})$ be a short-form notation for

the evaluation $[z = a, \overline{y} = \overline{\mathfrak{c}}] \in \mathit{Eval}(\mathit{Var})$. Then, we have

$$\begin{aligned}
& \mathit{ITE}(g, f_1, f_2)|_{z=b}(a, \overline{\mathfrak{c}}) \\
= \;& \mathit{ITE}(g, f_1, f_2)(b, \overline{\mathfrak{c}}) \\
= \;& (\, g(b, \overline{\mathfrak{c}}) \wedge f_1(b, \overline{\mathfrak{c}}) \,) \vee (\, \neg g(b, \overline{\mathfrak{c}}) \wedge f_2(b, \overline{\mathfrak{c}}) \,) \\
= \;& (\, g|_{z=b}(a, \overline{\mathfrak{c}}) \wedge f_1|_{z=b}(a, \overline{\mathfrak{c}}) \,) \vee (\, \neg g|_{z=b}(a, \overline{\mathfrak{c}}) \wedge f_2|_{z=b}(a, \overline{\mathfrak{c}}) \,) \\
= \;& \mathit{ITE}(g|_{z=b}, f_1|_{z=b}, f_2|_{z=b})(a, \overline{\mathfrak{c}}).
\end{aligned}$$

■

Thus, a node in a $\wp$-SOBDD for representing $\mathit{ITE}(g, f_1, f_2)$ is a node w such that $\mathit{info}(w) = \langle z, w_1, w_0 \rangle$ where

- z is the minimal essential variable of $\mathit{ITE}(g, f_1, f_2)$ according to $<_\wp$,
- w_1, w_0 are SOBDD nodes with:

$$f_{w_1} = \mathit{ITE}(g|_{z=1}, f_1|_{z=1}, f_2|_{z=1}) \quad \text{and} \quad f_{w_0} = \mathit{ITE}(g|_{z=0}, f_1|_{z=0}, f_2|_{z=0}).$$

This observation suggests a recursive algorithm which determines z and then recursively computes the nodes for ITE applied to the cofactors of $g = f_u$, $f_1 = f_{v_1}$, $f_2 = f_{v_2}$ for variable z. Since the explicit computation of z can be hard, we use the decomposition into cofactors for the minimal variable z that is essential for f_u, f_{v_1} or f_{v_2}:

$$z = \min\{\mathit{var}(u), \mathit{var}(v_1), \mathit{var}(v_2)\}$$

where the minimum is taken according to the total order $<_\wp$ on $\mathit{Var} \cup \{\bot\}$. (Recall that we put $\mathit{var}(v) = \bot$ for any drain v and that $x <_\wp \bot$ for all $x \in \mathit{Var}$.) If z' is the first essential variable of $\mathit{ITE}(f_u, f_{v_1}, f_{v_2})$, then $z \leqslant_\wp z'$, since no variable $y <_\wp z$ appears in the sub-OBDDs of nodes u, v_1, v_2, and hence, no such variable y can be essential for $\mathit{ITE}(f_u, f_{v_1}, f_{v_2})$. The case $z <_\wp z'$ is possible, if accidentally the cofactors $\mathit{ITE}(f_u, f_{v_1}, f_{v_2})|_{z=0}$ and $\mathit{ITE}(f_u, f_{v_1}, f_{v_2})|_{z=1}$ agree. In this case, however, we are in the situation of the elimination rule and the ITE algorithm returns the node representing $\mathit{ITE}(f_u, f_{v_1}, f_{v_2})|_{z=0}$. Otherwise, i.e., if the nodes w_0 and w_1 that have been recursively determined for $\mathit{ITE}(f_u, f_{v_1}, f_{v_2})|_{z=0}$ and $\mathit{ITE}(f_u, f_{v_1}, f_{v_2})|_{z=1}$, respectively, are different, then $z' = z$ and a node for representing $\mathit{ITE}(f_u, f_{v_1}, f_{v_2})$ is obtained by the *find_or_add*-operation applied to the info-triple $\langle z, w_1, w_0 \rangle$.

The question remains how to obtain the cofactors $f_u|_{z=b}$, $f_{v_1}|_{z=b}$, and $f_{v_2}|_{z=b}$. Nodes that represent these functions are obtained easily since (by choice of variable z) nodes u, v_1, and v_2 are on the z-level or below, i.e., $z \leqslant_\wp var(v)$ for $v \in \{u, v_1, v_2\}$. If $var(v) = z$, then $f_v|_{z=b}$ is represented by the b-successor of v. If $z <_\wp var(v)$, then z is not essential for f_v and we have $f_v|_{z=b} = f_v$. Thus, if we define

$$v|_{z=b} \;=\; \begin{cases} succ_b(v) & \text{if } var(v) = z \\ u & \text{if } z <_\wp var(v), \end{cases}$$

then $v|_{z=b}$ is the node in $\overline{\mathfrak{B}}$ representing $f_v|_{z=b}$. Hence, a node representing the function $ITE(f_u, f_{v_1}, f_{v_2})|_{z=b}$ is obtained by a recursive call of the ITE algorithm with the arguments $u|_{z=b}$, $v_1|_{z=b}$ and $v_2|_{z=b}$ (Lemma 6.76). Note that these are already existing nodes in the SOBDD $\overline{\mathfrak{B}}$.

The steps to realize the ITE-operator on a shared OBDD by means of a DFS-based traversal of the sub-OBDDs of u, v_1, and v_2 to determine the relevant cofactors (where recursive calls of the ITE-algorithm are required) have been summarized in Algorithm 22 on page 412.

Algorithm 22 $ITE(u, v_1, v_2)$ (first version)

```
if u is terminal then
  if val(u) = 1 then
    w := v1                                   (* ITE(1, f_v1, f_v2) = f_v1 *)
  else
    w := v2                                   (* ITE(0, f_v1, f_v2) = f_v2 *)
  fi
else
  z := min{var(u), var(v1), var(v2)};
  w1 := ITE(u|z=1, v1|z=1, v2|z=1);
  w0 := ITE(u|z=0, v1|z=0, v2|z=0);
  if w0 = w1 then
    w := w1;                                  (* elimination rule *)
  else
    w := find_or_add(z, w1, w0);              (* isomorphism rule (?) *)
  fi
fi
return w
```

Before discussing the complexity of the ITE algorithms, we will first study how the size of the SOBDD can change through performing the ITE algorithm. The size of the sub-OBDD for the (possibly new) node w representing $ITE(u, v_1, v_2)$ is bounded by $N_u \cdot N_{v_1} \cdot N_{v_2}$ where N_v denotes the number of the nodes in the sub-OBDD $\overline{\mathfrak{B}}_v$ for node v. This follows from

the fact that each node w' in the generated sub-OBDD for $ITE(u, v_1, v_2)$ corresponds to one or more triples (u', v_1', v_2'), where u' is a node in $\overline{\mathfrak{B}}_u$ and v_i' a node in $\overline{\mathfrak{B}}_{v_i}$ Formally:

Lemma 6.77. ROBDD Size of $ITE(g, f_1, f_2)$

The size of the $\wp$-ROBDD for $ITE(g, f_1, f_2)$ is bounded by $N_g \cdot N_{f_1} \cdot N_{f_2}$ where N_f denotes the size of the $\wp$-ROBDD for f.

Proof: Let $\wp = (z_1, \ldots, z_m)$ and let $\mathfrak{B}_f$ denote the $\wp$-ROBDD for f where the nodes are the $\wp$-consistent cofactors of f (see the proof of part (a) of Theorem 6.70). We write V_f for the node set of $\mathfrak{B}_f$, i.e.,

$$V_f \;=\; \{\, f|_{z_1=b_1,\ldots,z_i=b_i} \mid 0 \leqslant i \leqslant m,\, b_1, \ldots, b_i \in \{0,1\} \,\}.$$

Note that several of the cofactors $f|_{z_1=b_1,\ldots,z_i=b_i}$ might agree, and hence, they stand for the same element (node) of V_f. By Lemma 6.76, the node set $V_{ITE(g,\,f_1,\,f_2)}$ of the $\wp$-ROBDD $\mathfrak{B}_{ITE(g,\,f_1,\,f_2)}$ for $ITE(g, f_1, f_2)$ agrees with the set of switching functions

$$ITE(g|_{z_1=b_1,\ldots,z_i=b_i}, f_1|_{z_1=b_1,\ldots,z_i=b_i}, f_2 z_1 = b_1, \ldots, z_i = b_i)$$

where $0 \leqslant i \leqslant m$, $b_1, \ldots, b_i \in \{0,1\}$. Thus, the function

$$\imath : V_g \times V_{f_1} \times V_{f_2} \to V_{ITE(g,\,f_1,\,f_2)}, \quad \imath(g', f_1', f_2') \;=\; ITE(g', f_1', f_2')$$

that maps any triple (g', f_1', f_2') where g' is a node in $\mathfrak{B}_g$ (i.e., a $\wp$-consistent cofactor of g) and f_i' a node in $\mathfrak{B}_{f_i}$ (i.e., a $\wp$-consistent cofactor of f_i) to the node $ITE(g', f_1', f_2')$ of $\mathfrak{B}_{ITE(g,\,f_1,\,f_2)}$ yields a surjective mapping from $V_g \times V_{f_1} \times V_{f_2}$ to some superset of $V_{ITE(g,\,f_1,\,f_2)}$. Hence:

$$N_{ITE(g,\,f_1,\,f_2)} \;=\; |V_{ITE(g,\,f_1,\,f_2)}| \;\leqslant\; |V_g \times V_{f_1} \times V_{f_2}| \;=\; N_g \cdot N_{f_1} \cdot N_{f_2}$$

Observe that only the triples $(g', f_1', f_2') \in V_g \times V_{f_1} \times V_{f_2}$ where g', f_1', f_2' arise from g, f_1, f_2 by the same evaluation $[z_1 = b_1, \ldots, z_i = b_i]$ are mapped via $\imath$ to nodes of $\mathfrak{B}_{ITE(g,\,f_1,\,f_2)}$. Furthermore, $ITE(g', f_1', f_2') = ITE(g'', f_1'', f_2'')$ is possible for $(g', f_1', f_2') \neq (g'',\ f_1'',\ f_2'')$. Therefore, $N_{ITE(g,\,f_1,\,f_2)}$ can be much smaller than $N_g \cdot N_{f_1} \cdot N_{f_2}$. ■

As a consequence of Lemma 6.77, the size of the $\wp$-ROBDD for $f_1 \vee f_2$ is bounded above by the product of the sizes of the $\wp$-ROBDDs for f_1 and f_2. Recall that $f_1 \vee f_2 = ITE(f_1, 1, f_2)$ and hence

$$N_{f_1 \vee f_2} \;\leqslant\; N_{f_1} \cdot N_1 \cdot N_{f_2} \;=\; N_{f_1} \cdot N_{f_2}.$$

The same holds for conjunction and even for any other binary Boolean connective. This also applies to operators like $\oplus$ (xor, parity) where $f \oplus g = ITE(f, \neg g, g)$, i.e., negation

is needed to express $f \oplus g$ by ITE, f and g. Lemma 6.77 yields the bound $N_f \cdot N_g^2$ for the $\wp$-ROBDD size for $f \oplus g$. However, since the $\wp$-ROBDDs for g and $\neg g$ are isomorphic up to exchanging the values of the drains, the recursive calls in the ITE-algorithms have the form $ITE(u, v, w)$ where $f_v = \neg f_w$. Hence, the number of nodes in the $\wp$-ROBDD for $f \oplus g$ is bounded by the number of triples $(f', \neg g', g')$ where f' is a $\wp$-consistent cofactor of f and g' a $\wp$-consistent cofactor of g. This yields the upper bound $N_f \cdot N_g$ for the size of the $\wp$-ROBDD of $f \oplus g$.

Computed Table The problem with Algorithm 22 is that its worst-case running time is exponential. The reason for this is that, if there are several paths that lead from (u, v_1, v_2) to (u', v_1', v_2'), then $ITE(u', v_1', v_2')$ is invoked several times and the whole sub-OBDDs of these nodes u', v_1', v_2' are traversed in each of these recursive invocations. To avoid such redundant recursive invocations one uses a *computed table* that stores the tuples (u, v_1, v_2), where $ITE(u, v_1, v_2)$ has already been executed, together with the result, i.e., the SOBDDD-node w with $f_w = ITE(f_u, f_{v_1}, f_{v_2})$. Thus, we may refine the ITE-algorithm as shown in Algorithm 23 on page 414.

Algorithm 23 $ITE(u, v_1, v_2)$

if there is an entry for (u, v_1, v_2, w) in the computed table **then**
  return node w
else
  (* no entry for $ITE(u, v_1, v_2)$ in the computed table *)
  if u is terminal **then**
    if $val(u) = 1$ **then** $w := v_1$ **else** $w := v_2$ **fi**
  else
    $z := \min\{var(u), var(v_1), var(v_2)\}$;
    $w_1 := ITE(u|_{z=1}, v_1|_{z=1}, v_2|_{z=1})$;
    $w_0 := ITE(u|_{z=0}, v_1|_{z=0}, v_2|_{z=0})$;
    if $w_0 = w_1$ **then** $w := w_1$ **else** $w := find\_or\_add(z, w_1, w_0)$ **fi**;
    insert (u, v_1, v_2, w) in the computed table;
    return node w
  fi
fi

The number of recursive calls in Algorithm 23 for the input-nodes u, v_1, v_2 agrees with the $\wp$-ROBDD size of $ITE(f_u, f_{v_1}, f_{v_2})$, which is bounded by $N_u \cdot N_{v_1} \cdot N_{v_2}$, where $N_v = N_{f_v}$ denotes the number of nodes in the sub-OBDD of node v. The cost per recursive call is constant when the time to access the computed table and to perform the *find_or_add*-operation is assumed to be constant. This is an adequate assumption if suitable hashing techniques are used to organize both tables. However, in practice the running time of the ITE algorithm is often much better than this upper bound. First, only in extreme cases

is the size of the $\wp$-ROBDD for $ITE(f_u, f_{v_1}, f_{v_2})$ roughly $N_u \cdot N_{v_1} \cdot N_{v_2}$. Second, the use of the ITE operator yields the advantage that all synthesis algorithms that are expressible via ITE rely on the same computed table. This increases the hit rate and makes it possible that the computation aborts due to an entry in the computed table that has been made during the synthesis of another function. Moreover, there are several tricks to intensify this phenomenon. One simple trick is to make use of equivalence rules, such as

$$f_1 \vee f_2 = ITE(f_1, 1, f_2) = ITE(f_2, 1, f_1) = ITE(f_1, f_1, f_2) = \ldots,$$

and to transform the arguments of ITE into so-called *standard triples*. Furthermore, the ITE-algorithm can be refined by terminating the traversal in certain special cases. E.g., we have $ITE(g, f, f) = f$ and $ITE(g, 1, 0) = g$, which allows aborting the computation if either the last two arguments agree or they equal the pair consisting of the 1- and 0-drains.

Remark 6.78. The Negation Operator

We noticed above that the negation operator can be realized as an instance of the ITE operator as we have $\neg f = ITE(f, 0, 1)$. However, applying the ITE algorithm seems to be unnecessarily complicated since the $\wp$-ROBDDs for f and $\neg f$ just differ in the values of the drains. In fact, swapping the values of the drains is an adequate technique to realize negation on ROBDDs, but it is not for shared OBDDs (since changing the values of the drains also affects the functions of all other root nodes). However, there is a simple trick to perform negation in SOBDDs in constant time. It relies on the use of *complement bits* for the edges. This permits the representation of f and $\neg f$ by a single node. Negation then just means swapping the value of the complement bit of the incoming edge. Besides leading to smaller SOBDD sizes, the use of complement bits also tightens the effect of standard triples, since now more equivalence rules can be used to identify the input triples for ITE or for early termination. E.g., we have

$$ITE(f_1, 1, f_2) = f_1 \vee f_2 = \neg(\neg f_1 \wedge \neg f_2) \;=\; \neg ITE(\neg f_1, \neg f_2, 0)$$

and $ITE(g, 0, 1) = \neg g$. However, to ensure canonicity some extra requirements are needed. For instance, the constant function 0 could be represented by the 0-drain (with unnegated complement bit) or the 1-drain with negated complement bit. To guarantee the uniqueness, one could require that only the 1-drain be used and that complement bits be used only for the 0-edges (i.e., edges from the inner nodes to their 0-successors) and the pointers to the root nodes. For more information on such advanced implementation techniques, further theoretical considerations on OBDDs and variants thereof, we refer to textbooks on BDDs, such as [134, 292, 300, 418]. ■

Other Operators on OBDDs Although all Boolean connectives can be expressed by the ITE operator, some more operators are required to perform, e.g., the CTL model

checking procedure with an SOBDD representation of the transition system. Recall that in the symbolic computation of $Sat(\exists\Diamond B)$ and $Sat(\exists\Box B)$ (see Algorithms 20 and 21 on page 391) we use iterations of the form

$$f_{j+1}(\overline{x}) \;:=\; f_j(\overline{x}) \;\; op \;\; \exists\overline{x}'.(\,\Delta(\overline{x},\overline{x}') \,\wedge\, f_j(\overline{x}')\,)$$

where $op \in \{\vee, \wedge\}$ and $f_j = \chi_T$ is the characteristic function of some set T. Thus, f_{j+1} is the characteristic function of $T \cap Pre(T)$ (if $op = \wedge$) and $T \cup Pre(T)$ (if $op = \vee$). (For the treatment of constrained reachability properties like $\exists(C \,\mathsf{U}\, B)$, we have an additional conjunction with χ_C, but this is just a technical detail.) Besides disjunction and conjunction, these iterations use existential quantification and renaming. The major difficulty is the *preimage* computation, i.e., the computation of the symbolic representation of $Pre(T)$ by means of the expression $\exists\overline{x}'.(\Delta(\overline{x},\overline{x}') \wedge f_j(\overline{x}'))$, which is often called a *relational product*.

Let us start with the *rename* operator which is inherent in $f_j(\overline{x}')$ since f_j is a switching function for the variables in $\overline{x}$ and $f_j(\overline{x}') = f_j\{\overline{x}' \leftarrow \overline{x}\}$ means the function that results from f_j when renaming the unprimed variables x_i into their primed copies x_i'. At first glance, the renaming operator appears to be trivial as we simply may modify the variable labeling function by replacing x_i with x_i'. This operation certainly transforms a given ROBDD for $f(\overline{x})$ into a ROBDD for $f(\overline{x}')$. However, if we are given an arbitrary variable ordering $\wp$ where the relative order of the unprimed variables can be different from the relative order of the primed variables (i.e., $x_i <_\wp x_j$ while $x_j' <_\wp x_i'$) then this renaming operator is no longer adequate, since the resulting ROBDD for $f(\overline{x}')$ would rely on another ordering than $\wp$. Furthermore, for an implementation with shared OBDDs, modifying existing nodes is not appropriate since then the functions for all root nodes might be affected. In fact, it is not possible to design a general rename operator which runs in time polynomial in the size of the input ROBDD. To see why, consider the function

$$f \;=\; (z_1 \wedge y_1) \vee (z_2 \wedge y_2) \vee \ldots \vee (z_m \wedge y_m)$$

of Example 6.73 (page 404). Suppose $Var = \{z_i, y_i, z_i', y_i' \mid 1 \leqslant i \leqslant m\}$ and $\wp = (z_m, y_m, \ldots, z_1, y_1, z_1', \ldots, z_m', y_1', \ldots, y_m')$ and that we are given the $\wp$-ROBDD $\mathfrak{B}_f$ for f in the form of a root node v of a $\wp$-SOBDD. The goal is now to rename z_i into z_i' and y_i into y_i' in f, i.e., to compute the $\wp$-ROBDD representation of

$$f\{z_i' \leftarrow z_i, y_i' \leftarrow y_i \mid 1 \leqslant i \leqslant m\} \;=\; (z_1' \wedge y_1') \vee \ldots \vee (z_m' \wedge y_m').$$

By the results of Example 6.73: while the $\wp$-ROBDD size of f is $2m + 2$, the $\wp$-ROBDD size of $f\{\ldots\}$ is $\Omega(2^m)$. This observation shows that there is no linear-time algorithm that realizes the rename operator for arbitrary variable orderings. However, if we suppose that x_i and x_i' are neighbors in the ordering $\wp$, e.g., $x_i <_\wp x_i'$ and there is no variable z with $x_s <_\wp z <_\wp x_i'$, then renaming x_i into x_i' for a function $f(\overline{x})$ is simple. As for the ITE operator, we can work with a DFS-based traversal of the sub-OBDD for the node representing $f(\overline{x})$ and

generate the ROBDD for $f(\overline{x}')$ in a bottom-up manner; see Algorithm 24. The algorithm takes as input a node v of a π-SOBDD and tuples $\overline{x} = (x_1, \ldots, x_n)$, $\overline{x}' = (x_1', \ldots, x_n')$, of pairwise distinct variables such that $x_1', \ldots, x_n'$ are not essential for f_v and x_i und x_i' are neighbors in π. The output is a node w such that $f_w = f_v\{\overline{x} \leftarrow \overline{x}'\}$. To avoid multiple invocations of the algorithms with the same input node v, we use a computed table that stores all nodes v where $\mathit{Rename}(v, \overline{x} \leftarrow \overline{x}')$ has already been executed together with the output node w, i.e., the node w with $f_w = f_v\{\overline{x} \leftarrow \overline{x}'\}$.

Algorithm 24 $\mathit{Rename}(v, \overline{x} \leftarrow \overline{x}')$

```
if there is an entry (v, w) in the computed table then
  return w
else
  if v ist terminal then
    w := v
  else
    w_0 := Rename(succ_0(v), x̄ ← x̄');
    w_1 := Rename(succ_1(v), x̄ ← x̄');
    if var(v) = z_j for some j ∈ {1, ..., n} then
      z := z_j'                                  (* replace z_j with z_j' *)
    else
      z := var(v)
    fi
    w := find_or_add(z, w_1, w_0);
  fi
  insert (u, w) in the computed table;
  return w
fi
```

Remark 6.79. Interleaved Variable Orderings for Transition Systems

We noticed before (page 409) that interleaved variable orderings, such as $(x_1, x_1', \ldots, x_n, x_n')$, are favorable for the representation of transition systems. The rename operator yields one reason, as interleaved variable orderings permit use of the renaming that is inherent in the OBDD algorithms for the preimage computation by the above algorithm. Another formal argument for interleaved variable orderings is that they are beneficial for the construction of the ROBDD representation for the transition relation of a composite transition system. In Remark 6.59 (page 389) we saw that if TS arises from the synchronous product of transition systems $TS_1, \ldots, TS_m$, then the switching function $\Delta(\overline{x}_1, \ldots, \overline{x}_n, \overline{x}_1', \ldots, \overline{x}_n')$ for TS's transition relation is obtained by the conjunction of the switching functions $\Delta_i(\overline{x}_i, \overline{x}_i')$ for the transition relations in TS_i, $i = 1, \ldots, m$. Since the Δ_i's do not have common variables the $\wp$-ROBDD size of Δ is bounded by

$$N_\Delta \leqslant N_{\Delta_1} + \ldots + N_{\Delta_m}$$

whenever $\wp$ is an interleaved variable ordering where all variables in $\overline{x}_i$ and $\overline{x}'_i$ are grouped together. Thus, there is no exponential blowup for the ROBDD sizes! Although Lemma 6.77 yields the upper bound $N_{\Delta_1} \cdot \ldots \cdot N_{\Delta_m}$ for any variable ordering, for such interleaved variable orderings $\wp$ at most linear growth of the $\wp$-ROBDD sizes is guaranteed. This is due to the fact that the $\wp$-ROBDD for Δ arises by linking the $\wp$-ROBDDs for $\Delta_1, \ldots, \Delta_m$. E.g., if we suppose that all variables in $\overline{x}_i, \overline{x}'_i$ appear after the variables in $\overline{x}_1, \overline{x}'_1, \ldots, \overline{x}_{i-1}, \overline{x}'_{i-1}$ in $\wp$, then we may simply redirect any edge to the 1-drain in the $\wp$-ROBDD for Δ_{i-1} to the root of the $\wp$-ROBDD for Δ_i (for $1 \leqslant i < m$). This yields the $\wp$-ROBDD for Δ.

A slightly more involved argument applies to the interleaving $TS = TS_1 \,|||\, \ldots \,|||\, TS_m$ where Δ arises by the disjunction of the Δ_i's together with the side conditions $\overline{x}_j = \overline{x}'_j$ for $i \neq j$. With an interleaved variable ordering where the variables for TS_i appear before the variables of $TS_{i+1}, \ldots, TS_m$, we can guarantee that the $\wp$-ROBDD size N_Δ is bounded by $\mathcal{O}(\,(N_{\Delta_1} + \ldots + N_{\Delta_m}) \cdot n^2\,)$ where the n is the total number of variables in TS. The additional factor $\mathcal{O}(n^2)$ stands for the representation of the conditions $\overline{x}_i = \overline{x}'_i$. ■

Existential quantification is reducible to ITE and the cofactor operator, as we have $\exists x.f = f|_{x=0} \vee f|_{x=1}$. As we mentioned in the explanations for the ITE operator, the cofactor operator $f \mapsto f|_{x=b}$ is trivial if $x \leqslant_\wp z$ for the first essential variable z of f in the given variable ordering $\wp$, since then $f|_{z=b}$ is represented by the b-successor of the node representing f, if $x = z$, and $f|_{x=b} = f$, if $x <_\wp z$ or f is constant. If a representation for $f|_{x=b}$ is required where $z <_\wp x$, then we may use the fact that $(f|_{x=b})|_{z=\mathfrak{c}} = (f|_{z=\mathfrak{c}})|_{x=b}$ and apply the cofactor-operator recursively to the successors of the node representing f. This leads to Algorithm 25 on 419. Here, again, we use a computed table that organizes all pairs (v, w) where the cofactor $f_v|_{x=b}$ is known to be represented by node w.

The time complexity of the renaming algorithm, as well as the algorithm for obtaining cofactors is bounded by $\mathcal{O}(\mathit{size}(\overline{\mathfrak{B}}_v))$ as both rely on a DFS-based traversal of the sub-OBDD $\overline{\mathfrak{B}}_v$, assuming constant time for accessing the entries in the unique and computed table. The $\wp$-ROBDD size of f agrees with the $\wp$-ROBDD size of $f\{\overline{x} \leftarrow \overline{x}'\}$ under the assumptions we made for $\overline{x}$, $\overline{x}'$ and $\wp$. The $\wp$-ROBDD size of $f|_{x=b}$ is at most the $\wp$-ROBDD size of f. This follows from the fact that given the $\wp$-ROBDD $\mathfrak{B}_f$ for f, an $\wp$-ROBDD for $f|_{x=\overline{b}}$ is obtained by redirecting any edge $w \to u$ that leads to an x-node u to the b-successor of u and then removing all x-nodes.[7] In summary, the preimage computation via the *relational product*

$$\exists \overline{x}.(\Delta \wedge f\{\overline{x}' \leftarrow \overline{x}\})$$

– which is required for, e.g., the symbolic computation of $\mathit{Sat}(\exists \Box B)$ – could be performed by first applying the rename operator to f, then the conjunction operator (as an instance of

[7] Although this yields a correct operator for constructing the $\wp$-ROBDD for $f|_{x=b}$ from the $\wp$-ROBDD for f, the redirection of edges is not adequate for an implementation with shared OBDDs.

Algorithm 25 *Cof*(v, x, b)

if there is an entry (v, w) in the computed table **then**
  return w
else
  if v ist terminal **then**
    $w := v$
  else
    if $x \leqslant_\wp var(v)$ **then**
      $w := v|_{x=b}$
    else
      $z := var(v)$;
      $w_1 := Cof(succ_1(v), x, b)$; $w_0 := Cof(succ_0(v), x, b)$;
      if $w_0 = w_1$ **then** $w := w_1$ **else** $w := find\_or\_add(z, w_1, w_0)$ **fi**
    fi
  fi
  insert (u, w) in the computed table;
  return node w
fi

ITE operator) to the nodes representing Δ and $f(\overline{x}')$, and finally computing the existential quantifications by cofactors and disjunctions. This naive approach is very time-consuming as it relies on several top-down traversals in the shared OBDD. It also yields the problem that the ROBDD representation for $\Delta \wedge f\{\overline{x}' \leftarrow \overline{x}\}$ could be very large.

A more elegant approach for a BDD-based preimage computation by means of the relational product $\exists \overline{x}'.\,(\,\Delta \wedge f\{\overline{x}' \leftarrow \overline{x}\}\,)$ is to realize the existential quantifications, renaming and conjunction simultaneously by a single DFS traversal of the sub-OBDDs of the nodes representing Δ and f, with intermediate calls of the ITE operator to realize the disjunctions that are inherent in the existential quantification. Algorithm 26 on page 420 summarizes the main steps of this approach.

The input of Algorithm 26 are two nodes u and v of a $\wp$-SOBDD such that $f_u = \Delta(\overline{x}, \overline{x}')$ and $f_v = f(\overline{x}')$. The assumptions about the variable tuples $\overline{x}$, $\overline{x}'$ are as above, i.e., $\overline{x} = (x_1, \ldots, x_n)$ and $\overline{x}' = (x_1', \ldots, x_n')$ are tuples consisting of pairwise distinct variables such that the unprimed variables x_i and their primed copies x_i' are neighbours in $\wp$. For simplicity, let us suppose that $\wp$ interleaves the unprimed and primed variables, say

$$x_1 <_\wp x_1' <_\wp x_2 <_\wp x_2' <_\wp \ldots <_\wp x_n <_\wp x_n'.$$

The output of Algorithm 26 is a (possibly new) node w with $f_w = \exists \overline{x}'.\,(\,\Delta(\overline{x}, \overline{x}') \wedge f(\overline{x}')\,) = \exists \overline{x}'.(f_u \wedge f_v)$. The termination condition of Algorithm 26 is given by the cases: (i) there exists an entry in the computed table, or (ii) $\Delta = 0$ or $f = 0$ in which case $\exists \overline{x}'.(\Delta \wedge f) = 0$,

Algorithm 26 Relational product $RelProd(u, v)$

if there exists an entry (u, v, w) in the computed table **then return** w **fi**;
if u or v is the 0-drain **then return** the 0-drain **fi**;
if u and v are the 1-drain **then return** the 1-drain **fi**;

$y := \min\{var(u), var(v)\}$, say $y \in \{x_i, x_i'\}$
if $y = x_i$ **then**
  $w_{1,0} := RelProd(u|_{x_i=1,x_i'=0}, v|_{x_i=0})$;
  $w_{1,1} := RelProd(u|_{x_i=1,x_i'=1}, v|_{x_i=1})$;
  $w_1 := ITE(w_{1,0}, 1, w_{1,1})$;

  $w_{0,0} := RelProd(u|_{x_i=0,x_i'=0}, v|_{x_i=0}, \overline{x}, \overline{x}')$;
  $w_{0,1} := RelProd(u|_{x_i=0,x_i'=1}, v|_{x_i=1}, \overline{x}, \overline{x}')$;
  $w_0 := ITE(w_{0,0}, 1, w_{0,1})$;

  if $w_1 = w_0$ **then**
    $w := w_1$ (* elimination rule *)
  else
    $w := find\_or\_add(x_i, w_1, w_0)$
  fi
else
  $w_0 := RelProd(u|_{x_i'=0}, v)$; $w_1 := RelProd(u|_{x_i'=1}, v)$;
  $w := ITE(w_0, 1, w_1)$
fi
insert (u, v, w) in the computed table;
return w

or (iii) $\Delta = f = 1$ in which case $\exists \overline{x}'.(\Delta \wedge f) = 1$. In the remaining cases, the traversal of the sub-OBDDs of u and v relies on the expansion rule:

$$\begin{aligned} & \exists x_1' \exists x_2' \ldots \exists x_n'.\,(\,\Delta \;\wedge\; f\{x_1' \leftarrow x_1, x_2' \leftarrow x_2, \ldots, x_n' \leftarrow x_n\}\,)\,)|_{x_1=b} \\ = \;\; & \exists x_2' \ldots \exists x_n'.\,(\,\Delta|_{x_1=b,x_1'=0} \;\wedge\; f|_{x_1=0}\{x_2' \leftarrow x_2, \ldots, x_n' \leftarrow x_n\}\,) \quad \vee \\ & \exists x_2' \ldots \exists x_n'.\,(\,\Delta|_{x_1=b,x_1'=1} \;\wedge\; f|_{x_1=1}\{x_2' \leftarrow x_2, \ldots, x_n' \leftarrow x_n\}\,) \end{aligned}$$

In the literature, several techniques have been proposed that serve to improve the image or preimage computation. These range from techniques that rely on partitionings of the variables that attempt to perform the existential quantification as soon as possible (as they decrease the number of essential variables and often lead to smaller ROBDD sizes). Other techniques rely on so-called input- or output splitting (which use alternative expansion rules), and special ROBDD operators that attempt to replace, e.g., Δ with other switching functions $\tilde{\Delta}$ such that $\Delta \wedge f = \tilde{\Delta} \wedge f$ and such that the $\wp$-ROBDD for $\overline{\Delta}$ is smaller than the $\wp$-ROBDD for Δ. In the case of an iterated preimage computation by $T_0 = B$ and $T_{j+1} = T_j \cup \mathit{Pre}(T_j)$ for the symbolic computation of $\mathit{Pre}^*(B)$, one might also switch from T_j to any set $\tilde{T}$ such that $T_j \setminus T_{j-1} \subseteq \tilde{T} \subseteq T_j$ and compute $T_j \cup \mathit{Pre}(\tilde{T})$. For such advanced techniques, we refer the interested reader to [92, 292, 374] and the literature mentioned there.

Summary We now have all ingredients for a symbolic realization of the standard CTL model-checking approach which recursively computes the satisfaction sets of the subformulae by means of shared OBDDs. Initially, one has to construct the ROBDD representation of the transition system to be analyzed. This can be done in a compositional way by means of synthesis operators (disjunction, conjunction, etc.) as mentioned in Remark 6.59. Furthermore, we assume that ROBDD representations of the satisfaction sets for the atomic propositions are given. This assumption is justified since often the atomic propositions can serve as variables for the encoding of the states. (And in this case their satisfaction set is just given by a projection function.) The CTL model-checking procedure can then be performed by means of the ITE algorithm (to treat the propositional logic fragment of CTL) and the symbolic BFS-based algorithms sketched in Algorithms 20 and 21. Both rely on an iterative preimage computation. Techniques to do this efficiently have been discussed above. The termination condition requires checking the equality of two switching functions. This, in fact, is trivial for shared OBDDs since it simply amounts to the comparison of the corresponding nodes and can be performed in constant time.

6.8 CTL*

We saw in Theorem 6.21 on page 337 that CTL and LTL have incomparable expressiveness. An extension of CTL, proposed by Emerson and Halpern, called CTL*, combines the features of both logics, and thus is more expressive than either of them.

6.8.1 Logic, Expressiveness, and Equivalence

CTL* is an extension of CTL as it allows path quantifiers $\exists$ and $\forall$ to be arbitrarily nested with linear temporal operators such as $\bigcirc$ and U. In contrast, in CTL each linear temporal operator must be immediately preceded by a path quantifier. As in CTL, the syntax of CTL* distinguishes between state and path formulae. The syntax of CTL* state formulae is roughly as in CTL, while the CTL* path formulae are defined as LTL formulae, the only difference being that arbitrary CTL* state formulae can be used as atoms. For example, $\forall \bigcirc \bigcirc a$ is a legal CTL* formula, but does not belong to CTL. The same applies to the CTL* formulae $\exists \Box \Diamond a$ and $\forall \Box \Diamond a$. (However, $\forall \Box \Diamond a$ is equivalent to the CTL formula $\forall \Box \forall \Diamond a$.)

Definition 6.80. Syntax of CTL*

CTL* *state formulae* over the set AP of atomic propositions, briefly called CTL* formulae, are formed according to the following grammar:

$$\Phi ::= \text{true} \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi$$

where $a \in AP$ and φ is a path formula. The syntax of CTL* *path formulae* is given by the following grammar:

$$\varphi ::= \Phi \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc \varphi \mid \varphi_1 \,\mathsf{U}\, \varphi_2$$

where Φ is a state formula, and φ, φ_1, and φ_2 are path formulae. ■

As for LTL or CTL, we use derived propositional logic operators like $\vee, \rightarrow, \ldots$ and let

$$\Diamond\varphi \;=\; \text{true}\,\mathsf{U}\,\varphi \quad \text{and} \quad \Box\varphi \;=\; \neg\Diamond\neg\varphi.$$

The universal path quantifier $\forall$ can be defined in CTL* by existential quantification and negation:

$$\forall\varphi \;=\; \neg\exists\neg\varphi.$$

(Note that this is not the case for CTL.)

For example, the following formulae are syntactically correct CTL* formulae:

$$\forall\Box\,(\,\bigcirc\Diamond a \;\wedge\; \neg(b\,\mathsf{U}\,\Box c)\,)$$

and

$$\forall\bigcirc\Box\neg a \quad\wedge\quad \exists\Diamond\Box\,(a \;\vee\; \forall(b\,\mathsf{U}\,a)).$$

Note that these formulae are not CTL formulae.

Definition 6.81. Satisfaction Relation for CTL*

Let $a \in AP$ be an atomic proposition, $TS = (S, Act, \to, I, AP, L)$ be a transition system without terminal states, state $s \in S$, Φ, Ψ be CTL* state formulae, and φ, φ_1 and φ_2 be CTL* path formulae. The satisfaction relation $\models$ is defined for state formulae by

$$\begin{array}{lll}
s \models a & \text{iff} & a \in L(s),\\
s \models \neg\,\Phi & \text{iff} & \text{not } s \models \Phi,\\
s \models \Phi \wedge \Psi & \text{iff} & (s \models \Phi) \text{ and } (s \models \Psi),\\
s \models \exists\varphi & \text{iff} & \pi \models \varphi \text{ for some } \pi \in \mathit{Paths}(s).
\end{array}$$

For path π, the satisfaction relation $\models$ for path formulae is defined by:

$$\begin{array}{lll}
\pi \models \Phi & \text{iff} & s_0 \models \Phi,\\
\pi \models \varphi_1 \wedge \varphi_2 & \text{iff} & \pi \models \varphi_1 \text{ and } \pi \models \varphi_2,\\
\pi \models \neg\varphi & \text{iff} & \pi \not\models \varphi,\\
\pi \models \bigcirc\,\varphi & \text{iff} & \pi[1..] \models \varphi,\\
\pi \models \varphi_1\,\mathsf{U}\,\varphi_2 & \text{iff} & \exists\, j \geqslant 0.\ (\pi[j..] \models \varphi_2 \;\wedge\; (\forall\, 0 \leqslant k < j.\ \pi[k..] \models \varphi_1))
\end{array}$$

where for path $\pi = s_0\, s_1\, s_2 \ldots$ and integer $i \geqslant 0$, $\pi[i..]$ denotes the suffix of π from index i on. ■

Definition 6.82. CTL* Semantics for Transition Systems

For CTL*-state formula Φ, the *satisfaction set* $\mathit{Sat}(\Phi)$ is defined by

$$\mathit{Sat}(\Phi) \;=\; \{\, s \in S \mid s \models \Phi \,\}.$$

The transition system TS satisfies CTL* formula Φ if and only if Φ holds in all initial states of TS:

$$TS \models \Phi \quad \text{if and only if} \quad \forall s_0 \in I.\ s_0 \models \Phi.$$

■

Thus, $TS \models \Phi$ if and only if all initial states of TS satisfy the formula Φ.

LTL formulae are CTL* path formulae in which the elementary state formulae Φ are restricted to atomic propositions. Apparently, the interpretation of LTL over the paths of a transition system (see Definition 5.7, page 237) corresponds to the semantics of LTL obtained as a sublogic of CTL*. The following theorem demonstrates that the corresponding claim also holds for states. Thereby, every LTL formula φ is identified with the CTL* formula $\forall\varphi$, and the semantics of LTL over states is taken as a reference. Recall that according to the LTL semantics in paths, $s \models \varphi$ if and only if $\pi \models \varphi$ for all $\pi \in \mathit{Paths}(s)$.

Theorem 6.83. Embedding of LTL in CTL*

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system without terminal states. For each LTL formula φ over AP and for each $s \in S$:

$$\underbrace{s \models \varphi}_{\text{LTL semantics}} \quad \text{if and only if} \quad \underbrace{s \models \forall\varphi}_{\text{CTL}^* \text{ semantics}} \quad .$$

In particular, $TS \models \varphi$ (with respect to the LTL semantics) if and only if $TS \models \forall\varphi$ (with respect to the CTL* semantics)

As a result, it is justified to understand LTL (with interpretation over the states of a transition system) as a *sublogic* of CTL*. Theorem 6.21 (page 337) stated that the expressivenesses of LTL and CTL are incomparable. Since LTL is a sublogic of CTL*, it now follows that CTL* subsumes LTL and CTL, i.e., there exist CTL* formulae which can be expressed neither in LTL nor in CTL.

Theorem 6.84. CTL* is More Expressive Than LTL and CTL

For the CTL* formula over $AP = \{a, b\}$,

$$\Phi \;=\; (\forall\Diamond\Box\, a) \;\vee\; (\forall\Box\,\exists\Diamond\, b),$$

there does not exist any equivalent LTL or CTL formula.

Proof: This follows directly from the fact that $\forall\Box\,\exists\Diamond\, b$ is a CTL formula that cannot be expressed in LTL, whereas $\Diamond\Box\, a$ is an LTL formula that cannot be expressed in CTL. Both these facts follow from Theorem 6.21 (page 337). ■

The relationship between LTL, CTL, and CTL* is depicted in Figure 6.27.

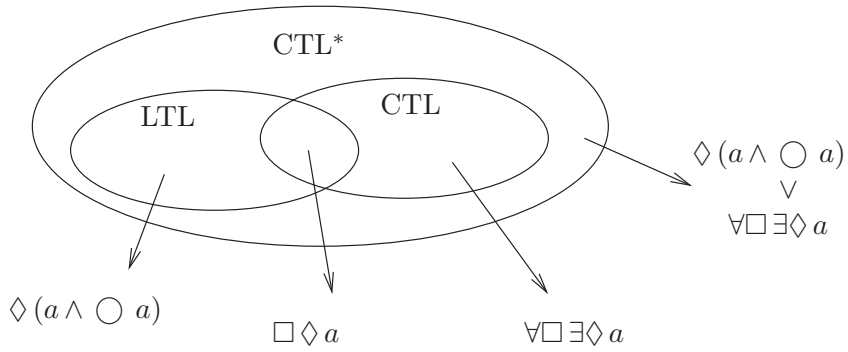


Figure 6.27: Relationship between LTL CTL and CTL*.

One of the consequences of this fact, is that—as in LTL—fairness assumptions can be expressed syntactically in CTL*. For instance, for fairness assumption *fair*, formulae of the form

$$\forall(\mathit{fair} \rightarrow \varphi) \quad \text{or} \quad \exists(\mathit{fair} \ \wedge \ \varphi)$$

are legal CTL* formulae. As for CTL or LTL, semantic equivalence $\equiv$ can be defined for CTL* formulae. Apart from the equivalence laws for CTL and the laws resulting from the equivalence laws for LTL, there exists a series of other important laws that are specific to CTL*. This includes, among others, the laws listed in Figure 6.28.

Example instances of the duality laws for the path quantifiers are

$$\neg\forall\Box\Diamond a \ \equiv \ \exists\Diamond\Box\neg a \quad \text{and} \quad \neg\exists\Box\Diamond a \ \equiv \ \forall\Diamond\Box\neg a.$$

As usual, universal quantification does not distribute over disjunction, and the same applies to existential quantification and conjunction:

$$\forall(\varphi \ \vee \ \psi) \ \not\equiv \ \forall\varphi \ \vee \ \forall\psi \quad \text{and} \quad \exists(\varphi \wedge \psi) \ \not\equiv \ \exists\varphi \wedge \exists\psi.$$

Path quantifier elimination should also be considered with care. For instance,

$$\forall\Diamond\,\Box\,\varphi \ \not\equiv \ \forall\Diamond\,\forall\Box\,\varphi \quad \text{and} \quad \exists\Box\,\Diamond\,\Phi \ \not\equiv \ \exists\Box\,\exists\Diamond\,\varphi.$$

Finally, we remark that for CTL*-state formula Φ we have that

$$\exists\Phi \ \equiv \Phi \quad \text{and} \quad \forall\Phi \ \equiv \ \Phi.$$

This is illustrated by means of an example. Consider the CTL* formula $\exists\forall\Diamond\, a$. This formula holds in state s whenever there exists an infinite path fragment $\pi = s_0\, s_1\, s_2 \ldots \in \mathit{Paths}(s)$, such that $\pi \models \forall\Diamond\, a$. Since $\pi \models \forall\Diamond\, a$ holds if and only if $s = s_0 \models \forall\Diamond\, a$, the formula $\exists\forall\Diamond a$ is equivalent to $\forall\Diamond a$.

duality laws for path quantifiers

$$\begin{aligned} \neg\forall\varphi &\equiv \exists\neg\varphi \\ \neg\exists\varphi &\equiv \forall\neg\varphi \end{aligned}$$

distributive laws

$$\begin{aligned} \forall(\varphi_1 \wedge \varphi_2) &\equiv \forall\varphi_1 \wedge \forall\varphi_2 \\ \exists(\varphi_1 \vee \varphi_2) &\equiv \exists\varphi_1 \vee \exists\varphi_2 \end{aligned}$$

quantifier absorption laws

$$\begin{aligned} \forall\Box\Diamond\varphi &\equiv \forall\Box\,\forall\Diamond\varphi \\ \exists\Diamond\Box\varphi &\equiv \exists\Diamond\,\exists\Box\varphi \end{aligned}$$

Figure 6.28: Some equivalence laws for CTL*.

Remark 6.85. Extending CTL with Boolean Connectors for Path Formulae (CTL$^+$)
Consider the following fragment, called CTL$^+$, of CTL*, which extends CTL by allowing Boolean operators in path formulae. CTL$^+$ state formulae over the set AP of atomic proposition are formed according to the following grammar:

$$\Phi ::= \text{true} \;\Big|\; a \;\Big|\; \Phi_1 \wedge \Phi_2 \;\Big|\; \neg\Phi \;\Big|\; \exists\varphi \;\Big|\; \forall\varphi$$

where $a \in AP$ and φ is a path formula. CTL$^+$ path formulae are formed according to the following grammar:

$$\varphi ::= \varphi_1 \wedge \varphi_2 \;\Big|\; \neg\varphi \;\Big|\; \bigcirc\,\Phi \;\Big|\; \Phi_1\,\mathsf{U}\,\Phi_2$$

where Φ, Φ_1, and Φ_2 are state formulae, and φ_1, φ_2 are path formulae. Surprisingly, CTL$^+$ is as expressive as CTL, i.e., for any CTL$^+$ state formula Φ^+ there exists an equivalent CTL formula Φ. For example:

$$\underbrace{\exists(a\,\mathsf{W}\,b)}_{\text{CTL formula}} \equiv \underbrace{\exists(\ (a\,\mathsf{U}\,b)\ \vee\ \Box a\ )}_{\text{CTL}^+\text{ formula}}$$

or

$$\underbrace{\exists(\Diamond a \wedge \Diamond b)}_{\text{CTL}^+\text{ formula}} \equiv \underbrace{\exists\Diamond(a \wedge \exists\Diamond b)\ \wedge\ \exists\Diamond(b \wedge \exists\Diamond a)}_{\text{CTL formula}}.$$

We do not provide here a full proof for transforming CTL$^+$ formulae into equivalent CTL formulae. The transformation relies on equivalence laws such as

$$\begin{aligned}
\exists(\neg \bigcirc \Phi) &\equiv \exists \bigcirc \neg\Phi \\
\exists(\neg(\Phi_1 \,\mathsf{U}\, \Phi_2)) &\equiv \exists(\ (\Phi_1 \wedge \neg\Phi_2) \,\mathsf{U}\, (\neg\Phi_1 \wedge \neg\Phi_2)\ ) \quad \vee \quad \exists\Box\neg\Phi_2 \\
\exists(\bigcirc \Phi_1 \wedge \bigcirc \Phi_2) &\equiv \exists \bigcirc (\Phi_1 \wedge \Phi_2) \\
\exists(\bigcirc \Phi \wedge (\Phi_1 \,\mathsf{U}\, \Phi_2)) &\equiv (\Phi_2 \wedge \exists \bigcirc \Phi) \quad \vee \quad (\Phi_1 \wedge \exists \bigcirc (\Phi \wedge \exists(\Phi_1 \,\mathsf{U}\, \Phi_2))) \\
\exists((\Phi_1 \,\mathsf{U}\, \Phi_2) \ \wedge \ (\Psi_1 \,\mathsf{U}\, \Psi_2)) &\equiv \exists((\Phi_1 \wedge \Psi_1) \,\mathsf{U}\, (\Phi_2 \wedge \exists(\Psi_1 \,\mathsf{U}\, \Psi_2))) \ \vee \\
&\qquad \exists((\Phi_1 \wedge \Psi_1) \,\mathsf{U}\, (\Psi_2 \wedge \exists(\Phi_1 \,\mathsf{U}\, \Phi_2))) \\
&\ \ \vdots
\end{aligned}$$

Thus, CTL can be expanded by means of a Boolean operator for path formulae without changing the expressiveness. However, CTL$^+$ formulae can be much shorter than the shortest equivalent CTL formulae. ■

6.8.2 CTL* Model Checking

This section treats a model-checking algorithm for CTL*. The CTL* model-checking problem is to establish whether $TS \models \Phi$ holds for a given finite transition system TS (without terminal states) and the CTL* state formula Φ. As we will see, an appropriate combination of the model-checking algorithms for LTL and CTL suffices.

As for CTL, the model-checking algorithm for CTL* is based on a bottom-up traversal of the syntax tree of the formula Φ to be checked. Due to the bottom-up nature of the algorithm, the satisfaction set $Sat(\Psi)$ for any state sub formulae Ψ of Φ has been computed before, and can be used to determine $Sat(\Phi)$. This holds in particular for the maximal proper state subformulae of Φ.

Definition 6.86. Maximal Proper State Subformula

State formula Ψ is a *maximal proper state subformula* of Φ whenever Ψ is a subformula of Φ that differs from Φ and that is not contained in any other proper state subformula of Φ. ■

The basic concept is to replace all maximal proper state subformulae of Φ by fresh atomic propositions $a_1, \ldots, a_k$, say. These propositions do not occur in Φ and are such that $a_i \in L(s)$ if and only if $s \in Sat(\Psi_i)$, the ith maximal state subformula of Φ. For state

subformulae whose "top-level" operator is a Boolean operator (such as negation or conjunction), the treatment is obvious. Let us consider the more interesting case of $\Psi = \exists\varphi$. By replacing all maximal state subformulae in φ, an LTL formula results! Since

$$s \models \exists\varphi \quad \text{iff} \quad \underbrace{s \not\models \forall\neg\varphi}_{\text{CTL}^* \text{ semantics}} \quad \text{iff} \quad \underbrace{s \not\models \neg\varphi}_{\text{LTL semantics}} ,$$

it suffices to compute the satisfaction set

$$Sat_{LTL}(\neg\varphi) \;=\; \{\, s \in S \mid s \models_{LTL} \neg\varphi \,\}$$

by means of an LTL model checker. (Here, the notations $Sat_{LTL}(\cdot)$ and $\models_{LTL}$ are used to emphasize that the basis is the LTL satisfaction relation.) The satisfaction set for $\Phi = \exists\varphi$ is now obtained by complementation:

$$Sat_{CTL^*}(\exists\varphi) \;=\; S \setminus Sat_{LTL}(\neg\varphi).$$

For CTL* formulae where the outermost operator is an universal quantification we simply may deal with

$$Sat_{CTL^*}(\forall\varphi) \;=\; Sat_{LTL}(\varphi)$$

where, as before, it is assumed that φ is an LTL formula resulting from the replacement of the maximal state subformula with fresh atomic propositions.

The main steps of the CTL* model-checking procedure are presented in Algorithm 27 on page 429.

Example 6.87. Abstract Example of CTL* Model Checking

The CTL* model-checking approach is illustrated by considering the CTL* formula:

$$\exists\varphi \quad \text{where} \quad \varphi \;=\; \bigcirc\,(\forall\Box\,\exists\Diamond\, a) \;\wedge\; \Diamond\Box\,\exists(\bigcirc\, a \wedge \Box\, b).$$

The maximal proper state subformulae of φ are

$$\Phi_1 \;=\; \forall\Box\,\exists\Diamond\, a \quad \text{and} \quad \Phi_2 \;=\; \exists(\bigcirc\, a \wedge \Box\, b).$$

Thus:

$$\varphi \;=\; \bigcirc\, \underbrace{(\forall\Box\,\exists\Diamond\, a)}_{\Phi_1} \;\wedge\; \Diamond\Box\, \underbrace{\exists(\bigcirc\, a \wedge \Box\, b)}_{\Phi_2} \;=\; \bigcirc\,\Phi_1 \;\wedge\; \Diamond\Box\,\Phi_2.$$

According to the model-checking algorithm for CTL*, the satisfaction sets $Sat(\Phi_i)$ are computed recursively. Subsequently, Φ_1 and Φ_2 are replaced with the atomic propositions a_1 and a_2, say. This yields the following LTL formula over the set of propositions $AP' = \{\, a_1, a_2 \,\}$:

$$\varphi' \;=\; \bigcirc\, a_1 \;\wedge\; \Diamond\Box\, a_2.$$

Algorithm 27 CTL* model checking algorithm (basic idea)

Input: finite transition system TS with initial states I, and CTL* formula Φ
Output: $I \subseteq Sat(\Phi)$

for all $i \leqslant |\,\Phi\,|$ **do**
  for all $\Psi \in Sub(\Phi)$ with $|\,\Psi\,| = i$ **do**
    switch(Ψ):

true	:	$Sat(\Psi) := S$;
a	:	$Sat(\Psi) := \{\, s \in S \mid a \in L(s) \,\}$;
$a_1 \wedge a_2$	:	$Sat(\Psi) := Sat(a_1) \cap Sat(a_2)$;
$\neg a$	:	$Sat(\Psi) := S \setminus Sat(a)$;
$\exists\varphi$	:	determine $Sat_{LTL}(\neg\varphi)$ by means of an LTL model-checker;
	:	$Sat(\Psi) := S \setminus Sat_{LTL}(\neg\varphi)$

    end switch
    $AP := AP \cup \{\, a_\Psi \,\}$; (* introduce fresh atomic proposition *)
    replace Ψ with a_Ψ
    forall $s \in Sat(\Psi)$ **do** $L(s) := L(s) \cup \{\, a_\Psi \,\}$; **od**
  od
od
return $I \subseteq Sat(\Phi)$

The labeling function $L' : S \to 2^{AP'}$ is given by:

$$a_i \in L'(s) \quad \text{if and only if} \quad s \in Sat(\Phi_i) \quad \text{for } i \in \{\,1,2\,\}.$$

Applying the LTL model-checking algorithm to the formula $\neg\varphi'$ yields the set of states satisfying (with respect to the LTL semantics) $\neg\varphi'$, i.e., $Sat_{LTL}(\neg\varphi')$. The complement of $Sat_{LTL}(\neg\varphi')$ provides the set $Sat_{CTL^*}(\varphi)$. ■

Evidently, the time complexity of the CTL* model-checking algorithm is dominated by the LTL model-checking phases. The additional effort that is necessary for CTL* model checking is polynomial in the size of the transition system and the length of the formula. Hence, a time complexity is obtained which is exponential in the length of the formula and linear in the size of the transition system.

Theorem 6.88. Time Complexity of CTL* Model Checking

For transition system TS with N states and K transitions, and CTL formula Φ, the CTL* model-checking problem $TS \models \Phi$ can be determined in time $\mathcal{O}((N+K)\cdot 2^{|\Phi|})$.*

Note that CTL* model checking can be solved by *any* LTL model-checking algorithm. These considerations show that there is a polynomial reduction of the CTL* model-

	CTL	LTL	CTL*
model checking	PTIME	PSPACE-complete	PSPACE-complete
without fairness	$size(TS) \cdot \lvert\Phi\rvert$	$size(TS) \cdot \exp(\lvert\Phi\rvert)$	$size(TS) \cdot \exp(\lvert\Phi\rvert)$
with fairness	$size(TS) \cdot \lvert\Phi\rvert \cdot \lvert fair\rvert$	$size(TS) \cdot \exp(\lvert\Phi\rvert) \cdot \lvert fair\rvert$	$size(TS) \cdot \exp(\lvert\Phi\rvert) \cdot \lvert fair\rvert$
for fixed specifications (model complexity)	$size(TS)$	$size(TS)$	$size(TS)$
satisfiability check	EXPTIME	PSPACE-complete	2EXPTIME
best known technique	$\exp(\lvert\Phi\rvert)$	$\exp(\lvert\Phi\rvert)$	$\exp(\exp(\lvert\Phi\rvert))$

Figure 6.29: Complexity of the model-checking algorithms and satisfiability checking.

checking problem to the LTL model-checking problem. As a result, the theoretical complexity results for LTL also apply to CTL*. Table 6.29 summarizes the complexity results for model checking CTL, CTL*, and LTL.

Theorem 6.89. Theoretical Complexity of CTL* Model Checking

The CTL* model-checking problem is PSPACE*-complete.*

6.9 Summary

- Computation Tree Logic (CTL) is a logic for formalizing properties over computation trees, i.e., the branching structure of the states.
- The expressivenesses of LTL and CTL are incomparable.
- Although fairness constraints cannot be encoded in CTL formulae, fairness assumptions can be incorporated in CTL by adapting the CTL semantics such that quantification is over fair paths, rather than over all paths.
- The CTL model-checking problem can be solved by a recursive descent procedure over the parse tree of the state formula to be checked. The set of states satisfying $\exists(\Phi \,\mathsf{U}\, \Psi)$ can be determined using a smallest fixed-point procedure; for $\exists\Box\,\Phi$ this is a largest fixed-point procedure.

- The time complexity of the CTL model-checking algorithm is linear in the size of the transition system and the length of the formula. In case fairness constraints are considered, an additional multiplicative factor that is proportional to the number of fairness constraints needs to be taken into account.
- Counterexamples and witnesses for CTL path formulae can be determined using a standard graph analysis.
- The CTL model-checking procedure can be realized symbolically by means of ordered binary decision diagrams. These provide a universal and canonical data structure for switching functions.
- Extended Computation Tree Logic (CTL*) is more expressive than either CTL and LTL.
- The CTL* model-checking problem can be solved by an appropriate combination of the recursive descent procedure (as for CTL) and the LTL model-checking algorithm.
- The CTL* model-checking problem is PSPACE-complete.

6.10 Bibliographic Notes

Branching temporal logics. Various types of branching temporal logic have been proposed in the literature. We mention a few important ones in increasing expressive power: Hennessy-Milner logic (HML [197]), Unified System of Branching-Time Logic [42], Computation Tree Logic (CTL [86]), Extended Computation Tree Logic (CTL* [86]), and modal μ-Calculus [243]. The modal μ-calculus is the most expressive among these languages, and HML is the least expressive. CTL has been extended with fairness by Emerson and Halpern [140] and by Emerson and Lei [143].

Not treated in this textbook is the task of proving satisfiability of formulae by means of algorithms or deductive techniques. The satisfiability problems for CTL, CTL*, and other temporal logics have been addressed by many researchers and used, e.g., in the context of synthesis problems where the goal is to design a system model from a given temporal specification. Emerson [138] has shown that checking CTL satisfiability is in the complexity class EXPTIME. This means that the time complexity of checking CTL satisfiability is exponential in the length of the formula. For CTL* this problem is double exponential [141] in the length of the formula. A complete axiomatization of CTL has been given by Ben-Ari, Manna and Pnueli [42] and Emerson and Halpern [139].

Branching vs. linear temporal logics. The discussion of the relative merits of linear- vs. branching-time logics goes back to the early eighties. Pnueli [338] established that linear

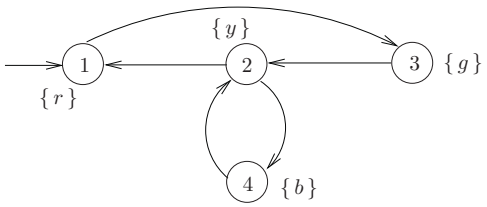
and branching temporal logics are based on two distinct notions of time. Various papers [85, 140, 259] show that the expressivenesses of LTL and CTL are incomparable. A somewhat more practical view on comparing the usefulness of LTL vs. CTL was recently given by Vardi [410]. The logic CTL* that encompasses LTL and CTL was defined by Clarke and Emerson [86].

CTL model checking. The first algorithms for CTL model checking were presented by Clarke and Emerson [86] in 1981 and (for a logic similar to CTL) by Queille and Sifakis [347] in 1982. The algorithm by Clarke and Emerson was polynomial in both the size of the transition system and the length of the formula, and could handle fairness. Clarke, Emerson, and Sistla [87] presented an efficiency improvement using the detection of strongly connected components and backward breadth-first search, yielding an algorithm that is linear in both the size of the system and the length of the formula. CTL model checking based on a forward search has been proposed by Iwashita, Nakata, and Hirose [224]. Emerson and Lei [143] showed that CTL* can be checked with essentially the same complexity as LTL, using a combination of the algorithms for LTL and CTL. The same authors consider in [142] CTL model checking under a broad class of fairness assumptions. Practical aspects of CTL* model checking have been reported by Bhat, Cleaveland, and Grumberg [50], and more recently by Visser and Barringer [414]. Algorithms for generating counterexamples and witnesses originate from the works by Clarke et al. [91] and Hojati, Brayton, and Kurshan [204]. More recent developments are the use of satisfiability solvers for propositional logic or quantified Boolean formulae to find counterexamples up to certain length, as proposed by Clarke et al. [84], and the use of tree-like counterexamples as opposed to linear ones by Clarke [93].

CTL model checkers. Clarke and Emerson [86] reported the first (fair) CTL model checker, called EMC. About the same time, Queille and Sifakis [347] announced CESAR, a model checker for a branching logic very similar to CTL. EMC was improved in [87] and constituted the basis for SMV (Symbolic Model Verifier), an efficient CTL model checker by McMillan based on a symbolic OBDD-based representation of the state space [288]. The concept of ordered binary decision diagrams has been proposed by Bryant [70]. The milestone for symbolic model checking with OBDDs is the paper by Burch et al.[74]. Further references for OBDD-based approaches have been given in Section 6.7. Recent variants of SMV are NuSMV [83] developed by the teams of Cimatti *et al.*, and SMV by McMillan and colleagues at Cadence Berkeley Laboratories that is focused on compositionality. Both tools are freely available. Another symbolic CTL model checker is VIS [62].

6.11 Exercises

EXERCISE 6.1. Consider the following transition system over $AP = \{ b, g, r, y \}$:

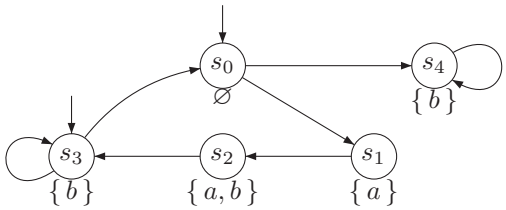


The following atomic propositions are used: r (red), y (yellow), g (green), and b (black). The model is intended to describe a traffic light that is able to blink yellow. You are requested to indicate for each of the following CTL formulae the set of states for which these formulae hold:

(a) $\forall \Diamond\, y$
(b) $\forall \Box\, y$
(c) $\forall \Box\, \forall \Diamond\, y$
(d) $\forall \Diamond\, g$
(e) $\exists \Diamond\, g$
(f) $\exists \Box\, g$
(g) $\exists \Box\, \neg\, g$
(h) $\forall (b\, \mathsf{U}\, \neg\, b)$
(i) $\exists (b\, \mathsf{U}\, \neg\, b)$
(j) $\forall (\neg\, b\, \mathsf{U}\, \exists \Diamond\, b)$
(k) $\forall (g\, \mathsf{U}\, \forall (y\, \mathsf{U}\, r))$
(l) $\forall (\neg\, b\, \mathsf{U}\, b)$

EXERCISE 6.2. Consider the following CTL formulae and the transition system TS outlined on the right:

$$\Phi_1 = \forall (a\, \mathsf{U}\, b) \vee \exists \bigcirc\, (\forall \Box\, b)$$
$$\Phi_2 = \forall \Box\, \forall (a\, \mathsf{U}\, b)$$
$$\Phi_3 = (a \wedge b) \rightarrow \exists \Box\, \exists \bigcirc\, \forall (b\, \mathsf{W}\, a)$$
$$\Phi_4 = (\forall \Box\, \exists \Diamond\, \Phi_3)$$



Determine the satisfaction sets $Sat(\Phi_i)$ and decide whether $TS \models \Phi_i$ $(1 \leqslant i \leqslant 4)$.

EXERCISE 6.3. Which of the following assertions are correct? Provide a proof or a counterexample.

(a) If $s \models \exists \Box\, a$, then $s \models \forall \Box\, a$.
(b) If $s \models \forall \Box\, a$, then $s \models \exists \Box\, a$.
(c) If $s \models \forall \Diamond\, a \ \vee\ \forall \Diamond\, b$, then $s \models \forall \Diamond\, (a \ \vee\ b)$.
(d) If $s \models \forall \Diamond\, (a \ \vee\ b)$, then $s \models \forall \Diamond\, a \ \vee\ \forall \Diamond\, b$.

(e) If $s \models \forall(a \,\mathsf{U}\, b)$, then $s \models \neg\,(\exists(\neg b \,\mathsf{U}\, (\neg a \ \wedge\ \neg b)) \ \vee\ \exists\Box\, \neg b)$.

Exercise 6.4. Let Φ and Ψ be arbitrary CTL formulae. Which of the following equivalences for CTL formulae are correct?

(a) $\forall\bigcirc\, \forall\Diamond\, \Phi \ \equiv\ \forall\Diamond\, \forall\bigcirc\, \Phi$

(b) $\exists\bigcirc\, \exists\Diamond\, \Phi \ \equiv\ \exists\Diamond\, \exists\bigcirc\, \Phi$

(c) $\forall\bigcirc\, \forall\Box\, \Phi \ \equiv\ \forall\Box\, \forall\bigcirc\, \Phi$

(d) $\exists\bigcirc\, \exists\Box\, \Phi \ \equiv\ \exists\Box\, \exists\bigcirc\, \Phi$

(e) $\exists\Diamond\, \exists\Box\, \Phi \ \equiv\ \exists\Box\, \exists\Diamond\, \Phi$

(f) $\forall\Box\, (\Phi \ \Rightarrow\ (\neg\Psi \ \wedge\ \exists\bigcirc\, \Phi)) \ \equiv\ (\Phi \ \Rightarrow\ \neg\forall\Diamond\, \Psi)$

(g) $\forall\Box\, (\Phi \ \Rightarrow\ \Psi) \ \equiv\ (\exists\bigcirc\, \Phi \ \Rightarrow\ \exists\bigcirc\, \Psi)$

(h) $\neg\forall(\Phi \ \mathsf{U}\ \Psi) \ \equiv\ \exists(\Phi \ \mathsf{U}\ \neg\Psi)$

(i) $\exists((\Phi \wedge \Psi) \ \mathsf{U}\ (\neg\Phi \wedge \Psi)) \ \equiv\ \exists(\Phi \ \mathsf{U}\ (\neg\Phi \wedge \Psi))$

(j) $\forall(\Phi \ \mathsf{W}\ \Psi) \equiv \neg\exists(\neg\Phi \ \mathsf{W}\ \neg\Psi)$

(k) $\exists(\Phi \ \mathsf{U}\ \Psi) \equiv \exists(\Phi \ \mathsf{U}\ \Psi) \ \wedge\ \exists\Diamond\Psi$

(l) $\exists(\Psi \ \mathsf{W}\ \neg\Psi) \vee\ \forall(\Psi \ \mathsf{U}\ \text{false}) \equiv \exists\bigcirc\ \Phi \ \vee\ \forall\bigcirc\ \neg\Phi$

(m) $\forall\Box\Phi \ \wedge\ (\neg\Phi \ \vee\ \exists\bigcirc\ \exists\Diamond\neg\Phi) \equiv \exists X\neg\Phi \ \wedge\ \forall\bigcirc\ \Phi$

(n) $\forall\Box\forall\Diamond\Phi \equiv \Phi \ \wedge\ (\forall\bigcirc\ \forall\Box\forall\Diamond\Phi) \ \vee\ \forall\bigcirc\ (\forall\Diamond\Phi \ \wedge\ \forall\Box\forall\Diamond\Phi)$

(o) $\forall\Box\Phi \equiv \Phi \ \vee\ \forall\bigcirc\ \forall\Box\Phi$

Exercise 6.5. Consider an elevator system that services $N > 0$ floors numbered 0 through $N-1$. There is an elevator door at each floor with a call button and an indicator light that signals whether or not the elevator has been called. In the elevator cabin there are N send buttons (one per floor) and N indicator lights that inform to which floor(s) is going to be sent. For simplicity consider $N = 4$. Present a set of atomic propositions—try to minimize the number of propositions—that are needed to describe the following properties of the elevator system as CTL formulae and give the corresponding CTL formulae:

(a) The doors are "safe", i.e., a floor door is never open if the cabin is not present at the given floor.

(b) The indicator lights correctly reflect the current requests. That is, each time a button is pressed, there is a corresponding request that needs to be memorized until fulfillment (if ever).

(c) The elevator only services the requested floors and does not move when there is no request.

(d) All requests are eventually satisfied.

EXERCISE 6.6. Consider the single pulser circuit, a hardware circuit that is part of a set of benchmark circuits for hardware verification. The single pulser has the following informal specification: "For every pulse at the input *inp* there appears exactly one pulse of length 1 at output *outp*, independent of the length of the input pulse". Thus, the single pulser circuit is required to generate an output pulse between two rising edges of the input signal. The following questions require the formulation of the circuit in terms of CTL. Suppose we have the proposition *rise_edge* at our disposal which is true if the input was low (0) at time instant $n-1$ and high (1) at time instant n (for natural $n > 0$). It is assumed that input sequences of the circuit are well behaved, i.e., more that the rising edge appears in the input sequence.
Questions: specify the following requirements of the circuit in CTL:

(a) A rising edge at the inputs leads to an output pulse.

(b) There is at most one output pulse for each rising edge.

(c) There is at most one rising edge for each output pulse.

(This exercise is taken from [246].)

EXERCISE 6.7. Transform the following CTL formulae into ENF and PNF. Show all intermediate steps.

$$\Phi_1 \;=\; \forall\big(\,(\neg a)\,\mathsf{W}\,(b \rightarrow \forall \bigcirc\, c)\,\big)$$

$$\Phi_2 \;=\; \forall \bigcirc \big(\;\; \exists((\neg a)\,\mathsf{U}\,(b \wedge \neg c)) \;\;\vee\;\; \exists\Box\, \forall \bigcirc\, a\,\big)$$

EXERCISE 6.8. Provide two finite transition systems TS_1 and TS_2 (without terminal states, and over the same set of atomic propositions) and a CTL formula Φ such that $\mathit{Traces}(TS_1) = \mathit{Traces}(TS_2)$ and $TS_1 \models \Phi$, but $TS_2 \not\models \Phi$.

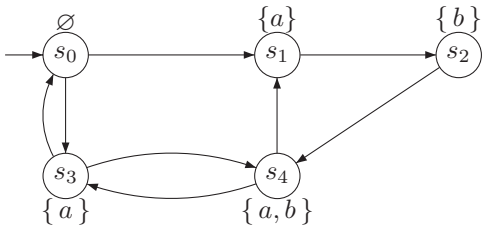
EXERCISE 6.9. Consider the CTL formula

$$\Phi = \forall\Box\,\big(a \rightarrow \forall\Diamond\,(b \wedge \neg a)\big)$$

and the following CTL fairness assumption:

$$\mathit{fair} = \forall\Diamond\,\forall\bigcirc\,(a \wedge \neg b) \rightarrow \forall\Diamond\,\forall\bigcirc\,(b \wedge \neg a) \wedge \Diamond\,\Box\,\exists\Diamond\, b \rightarrow \Box\,\Diamond\, b.$$

Prove that $TS \models_{\mathit{fair}} \Phi$ where transition system TS is depicted below.



EXERCISE 6.10. Let $\wp = (z_1, z_2, z_3, z_4, z_5, z_6)$. Depict the $\wp$-ROBDD for the majority function

$$MAJ([z_1 = b_1, z_2 = b_2, \ldots, z_6 = b_6]) \;=\; \begin{cases} 1 & \text{if } b_1 + b_2 + \ldots + b_6 \geqslant 4 \\ 0 & \text{otherwise.} \end{cases}$$

EXERCISE 6.11. Consider the function

$$F(x_0, \ldots, x_{n-1}, a_0, \ldots, a_{k-1}) = x_m$$

where $n = 2^k$, and $m = \sum_{j=0}^{k-1} a_j 2^j$. Let k=3. Questions:

(a) Depict the $\wp$-ROBDD for $\wp = (a_0, \ldots, a_{k-1}, x_0, \ldots, x_{n-1})$.

(b) Depict the $\wp$-ROBDD for $\wp = (a_0, x_0, \ldots, a_{k-1}, x_{k-1}, x_k, \ldots, x_{n-1})$.

EXERCISE 6.12. Let $\mathfrak{B}$ and $\mathfrak{C}$ be two $\wp$-ROBDDs. Design an algorithm that checks whether $f_{\mathfrak{B}} = f_{\mathfrak{C}}$ and runs in time linear in the sizes of $\mathfrak{B}$ and $\mathfrak{C}$.

(*Hint: It is assumed that* $\mathfrak{B}$ *and* $\mathfrak{C}$ *are given as separate graphs (and not by nodes of a shared OBDD).*)

EXERCISE 6.13. Let TS be a finite transition system (over AP) without terminal states, and Φ and Ψ be CTL state formulae (over AP). Prove or disprove

$$TS \models \exists(\Phi \,\mathsf{U}\, \Psi) \quad \text{if and only if} \quad TS' \models \exists\Diamond\, \Psi$$

where TS' is obtained from TS by eliminating all outgoing transitions from states s such that $s \models \Psi \;\vee\; \neg\Phi$.

EXERCISE 6.14. Check for each of the following formula pairs (Φ_i, φ_i) whether the CTL formula Φ_i is equivalent to the LTL formula φ_i. Prove the equivalence or provide a counterexample that illustrates why $\Phi_i \not\equiv \varphi_i$.

(a) $\Phi_1 \;=\; \forall\Box\forall \bigcirc\, a.$ and $\varphi_1 \;=\; \Box \bigcirc\, a$

(b) $\Phi_2 \;=\; \forall\Diamond\forall \bigcirc\, a$ and $\varphi_2 \;=\; \Diamond \bigcirc\, a.$

(c) $\Phi_3 \;=\; \forall\Diamond(a \wedge \exists \bigcirc\, a)$ and $\varphi_3 \;=\; \Diamond(a \wedge \bigcirc\, a).$

(d) $\Phi_4 \;=\; \forall\Diamond a \;\;\vee\;\; \forall\Diamond b$ and $\varphi_4 \;=\; \Diamond(a \vee b).$

(e) $\Phi_5 \;=\; \forall\Box(a \rightarrow \forall\Diamond b)$ and $\varphi_5 \;=\; \Box(a \rightarrow \Diamond b).$

(f) $\Phi_6 \;=\; \forall(b \,\mathsf{U}\, (a \wedge \forall\Box b))$ and $\varphi_6 \;=\; \Diamond a \;\wedge\; \Box b.$

Exercise 6.15.

(a) Prove, using Theorem 6.18, that there does not exist an equivalent LTL formula for the CTL formula $\Phi_1 = \forall\Diamond\,(a \wedge \exists\bigcirc\, a)$.

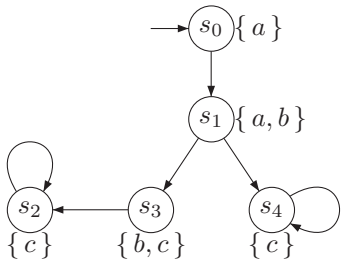
(b) Now prove directly (i.e. without Theorem 6.18), that there does not exist an equivalent LTL formula for the CTL formula $\Phi_2 = \forall\Diamond\,\exists\bigcirc\,\forall\Diamond\,\neg a$. (*Hint: Argument by contraposition.*)

Exercise 6.16.

Consider the following CTL formulae

$$\Phi_1 = \exists\Diamond\,\forall\Box\, c \quad \text{and} \quad \Phi_2 = \forall(a \,\mathsf{U}\, \forall\Diamond\, c)$$

and the transition system TS outlined on the right. Decide whether $TS \models \Phi_i$ for $i = 1, 2$ using the CTL model-checking algorithm. Sketch its main steps.



Exercise 6.17. Provide an algorithm in pseudo code to compute $Sat(\forall(\Phi \,\mathsf{U}\, \Psi))$ in a direct manner, i.e., without transforming the formula into ENF.

Exercise 6.18.

(a) Prove that $Sat(\exists(\Phi \,\mathsf{W}\, \Psi))$ is the largest set T such that

$$T \;\subseteq\; Sat(\Psi) \;\cup\; \big\{s \in Sat(\Phi) \mid Post(s) \cap T \neq \varnothing\big\}.$$

(b) Prove that $Sat(\forall(\Phi \,\mathsf{W}\, \Psi))$ is the largest set T such that

$$T \;\subseteq\; Sat(\Psi) \;\cup\; \big\{s \in Sat(\Phi) \mid Post(s) \subseteq T\big\}.$$

Use the above characterizations to provide efficient algorithms for computing the sets $Sat(\exists(\Phi \,\mathsf{W}\, \Psi))$ and $Sat(\forall(\Phi \,\mathsf{W}\, \Psi))$ in a direct manner.

EXERCISE 6.19. Consider the fragment ECTL of CTL which consists of formulae built according to the following grammar:

$$\begin{aligned} \Phi &::= a \mid \neg a \mid \Phi \wedge \Phi \mid \exists \varphi \\ \varphi &::= \bigcirc \Phi \mid \Box \Phi \mid \Phi \,\mathsf{U}\, \Phi \end{aligned}$$

For two transition systems $TS_1 = (S_1, Act, \rightarrow_1, I_1, AP, L_1)$ and $TS_2 = (S_2, Act, \rightarrow_2, I_2, AP, L_2)$, let $TS_1 \subseteq TS_2$ iff $S_1 \subseteq S_2$, $\rightarrow_1 \subseteq \rightarrow_2$, $I_1 = I_2$ and $L_1(s) = L_2(s)$ for all $s \in S_1$.

(a) Prove that for all ECTL formulae Φ and all transition systems TS_1, TS_2 with $TS_1 \subseteq TS_2$, it holds:

$$TS_1 \models \Phi \quad \Longrightarrow \quad TS_2 \models \Phi.$$

(b) Give a CTL formula which is not equivalent to any other ECTL formula. Justify your answer.

EXERCISE 6.20. In CTL, the release operator is defined by

$$\exists(\Phi \,\mathsf{R}\, \Psi) = \neg\forall((\neg\Phi) \,\mathsf{U}\, (\neg\Psi)) \quad \text{and} \quad \forall(\Phi \,\mathsf{R}\, \Psi) = \neg\exists((\neg\Phi) \,\mathsf{U}\, (\neg\Psi)).$$

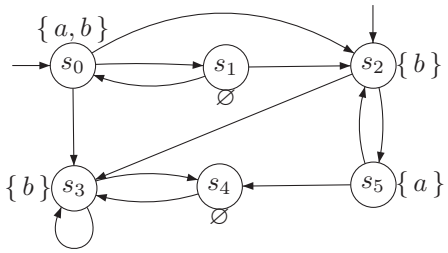
(a) Provide expansion laws for $\exists(\Phi \,\mathsf{R}\, \Psi)$ and $\forall(\Phi \,\mathsf{R}\, \Psi)$.

(b) Give a pseudo code algorithm for computing $Sat(\exists(\Phi \,\mathsf{R}\, \Psi))$ and do the same for computing $Sat(\forall(\Phi \,\mathsf{R}\, \Psi))$.

EXERCISE 6.21. Consider the CTL formula Φ and the strong fairness assumption *sfair*:

$$\begin{aligned} \Phi &= \forall\Box\forall\Diamond\, a \\ \mathit{sfair} &= \Box\Diamond \underbrace{(b \wedge \neg a)}_{\Phi_1} \rightarrow \Box\Diamond \underbrace{\exists\,(b \,\mathsf{U}\, (a \wedge \neg b))}_{\Psi_1} \end{aligned}$$

and transition system TS over $AP = \{\, a, b \,\}$ which is given by



Questions:

(a) Determine $Sat(\Phi_1)$ and $Sat(\Psi_1)$ (without fairness).

(b) Determine $Sat_{sfair}(\exists\Box \text{ true})$.

(c) Determine $Sat_{sfair}(\Phi)$.

EXERCISE 6.22. Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system without terminal states, $a, b \in AP$ and $s \in S$. Furthermore, let *fair* be a CTL fairness assumption and $a_{fair} \in AP$ an atomic proposition with $a_{fair} \in L(s)$ iff $s \models_{fair} \exists\Box\text{true}$.

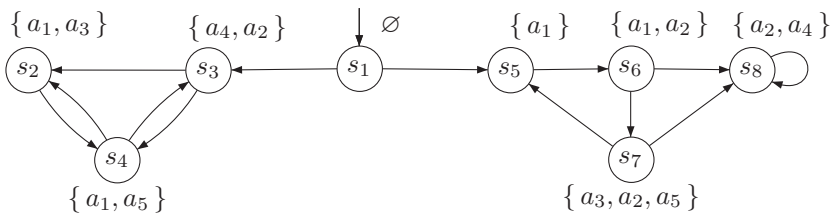
Which of the following assertions are correct? Give a proof or counterexample.

(a) $s \models_{fair} \forall(a \,\mathsf{U}\, b) \quad \text{iff} \quad s \models \forall(a \,\mathsf{U}\, (b \wedge a_{fair}))$.

(b) $s \models_{fair} \exists(a \,\mathsf{W}\, b) \quad \text{iff} \quad s \models \exists(a \,\mathsf{W}\, (b \wedge a_{fair}))$.

(c) $s \models_{fair} \forall(a \,\mathsf{W}\, b) \quad \text{iff} \quad s \models \forall(a \,\mathsf{W}\, (a_{fair} \rightarrow b))$.

EXERCISE 6.23. Consider the following transition system TS over $AP = \{ a_1, \ldots, a_6 \}$.



Let $\Phi = \exists \bigcirc \big(a_1 \rightarrow \exists(a_1 \,\mathsf{U}\, a_2)\big)$ and $\mathit{sfair} = \mathit{sfair}_1 \wedge \mathit{sfair}_2 \wedge \mathit{sfair}_3$ a strong CTL fairness assumption where

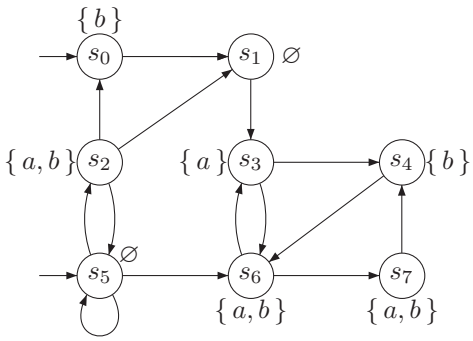
$$\begin{aligned} \mathit{sfair}_1 &= \Box\Diamond\, \forall\Diamond(a_1 \vee a_3) \longrightarrow \Box\Diamond a_4 \\ \mathit{sfair}_2 &= \Box\Diamond(a_3 \wedge \neg a_4) \longrightarrow \Box\Diamond a_5 \\ \mathit{sfair}_3 &= \Box\Diamond(a_2 \wedge a_5) \longrightarrow \Box\Diamond a_6 \end{aligned}$$

Sketch the main steps for computing the satisfaction sets $\mathit{Sat}_{\mathit{sfair}}(\exists\Box\mathrm{true})$ and $\mathit{Sat}_{\mathit{sfair}}(\Phi)$.

Exercise 6.24. Consider the CTL* formula over $AP = \{\, a, b \,\}$:

$$\Phi = \forall\Diamond\,\Box\,\exists\bigcirc\,\big(a \,\mathsf{U}\, \exists\Box\, b\big)$$

and the transition system TS outlined below:



Apply the CTL* model-checking algorithm to compute $\mathit{Sat}(\Phi)$ and decide whether $TS \models \Phi$. (*Hint: You may infer the satisfaction sets for LTL formulae directly.*)

Exercise 6.25. The model-checking algorithm presented in Section 6.5 treats CTL with strong fairness assumptions. Explain which modifications are necessary to deal with weak CTL fairness assumptions:

$$\mathit{wfair} = \bigwedge_{1 \leqslant i \leqslant k} \big(\, \Diamond\Box a_i \longrightarrow \Box\Diamond b_i \,\big).$$

You may assume that $a_i, b_i \in \{\, \mathrm{true} \,\} \cup AP$.

Exercise 6.26. Which of the following equivalences for CTL* are correct? Provide a proof or a counterexample.

(a) $\forall \bigcirc \forall \Box \Phi \equiv \forall \bigcirc \Box \Phi$

(b) $\exists \bigcirc \exists \Box \Phi \equiv \exists \bigcirc \Box \Phi$

(c) $\forall(\varphi \wedge \psi) \equiv \forall\varphi \wedge \forall\psi$

(d) $\exists(\varphi \wedge \psi) \equiv \exists\varphi \wedge \exists\psi$

(e) $\neg\forall(\varphi \rightarrow \psi) \equiv \exists(\varphi \wedge \neg\psi)$

(f) $\exists\Box\exists \bigcirc \Phi \wedge \neg\forall \bigcirc \neg\Phi \equiv \exists\Box(\neg \bigcirc \neg\Phi)$

(g) $\forall(\Diamond\Psi \wedge \Box\Phi) \equiv \forall\Diamond(\Psi \wedge \forall\Box\Phi) \wedge \forall\Box(\Phi \wedge \forall\Diamond\Psi)$

(h) $\exists(\Diamond\Psi \wedge \Box\Phi) \equiv \exists\Diamond(\Psi \wedge \exists\Box\Phi)$

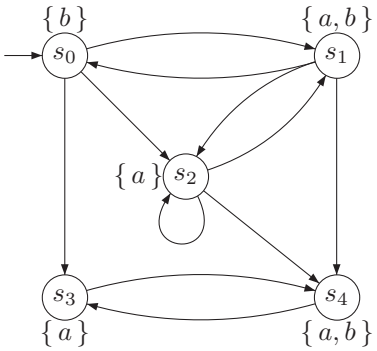
Here, Φ, Ψ are arbitrary CTL* state formulae and ψ, φ are CTL* path formulae.

Exercise 6.27.

Consider the transition system TS and the CTL* formula

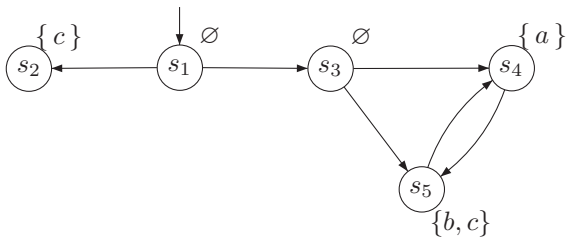
$$\Phi = \exists\big(\bigcirc(a \wedge \neg b) \wedge \bigcirc\forall(b \,\mathsf{U}\, \Box a)\big).$$

Apply the CTL* model-checking algorithm to check whether $TS \models \Phi$ and sketch its main steps as well as its output. (*Hint: You may compute the LTL satisfaction sets directly.*)



Exercise 6.28. Consider the transition system TS depicted below and the CTL* formula

$$\Phi \;=\; \exists(\Box\Diamond b \;\wedge\; \Box\exists\Diamond \bigcirc a) \;\wedge\; \forall\Box\Diamond \bigcirc c.$$



Sketch the main steps that the CTL* model checking algorithm performs for checking whether $TS \models \Phi$.

Exercise 6.29. Provide an example of a CTL* formula which is not a CTL$^+$ formula, but there exists an equivalent CTL formula.

Exercise 6.30. Provide equivalent CTL formulae for the CTL$^+$ formulae $\forall(\Diamond a \wedge \Box b)$ and $\forall(\bigcirc a \wedge \neg(a \,\mathsf{U}\, (\Box b)))$.

Practical Exercises

The remaining exercises are modeling and veriifcation exercises using the CTL model checker NuSMV [83].

Exercise 6.31. The following program is a mutual exclusion protocol for two processes due to Pnueli (taken from [118]). There is a single shared variable s which is either 0 or 1, and initially equals 1. Besides, each process has a local Boolean variable y that initially equals 0. The program text for process P_i $(i = 0, 1)$ is as follows:

l0: **loop forever do**
  begin
  l1: Noncritical section
  l2: $(y_i, s) := (1, i)$;
  l3: **wait until** $((y_{1-i} = 0) \ \vee \ (s \neq i))$;
  l4: Critical section
  l5: $y_i := 0$
end.

Here, the statement $(y_i, s) := (1, i)$ is a *multiple assignment* in which variable $y_i := 1$ and $s := i$ is a single, atomic step.

The intuition behind this protocol is as follows. The variables y_0 and y_1 are used by each process to signal the other process of active interest in entering the critical section. On leaving the noncritical section, process P_i sets its own local variable y_i to 1. In a similar way this variable is reset to 0 once the critical section is left. The global variable s is used to resolve a tie situation between the processes. It serves as a logbook in which each process that sets its y variable to 1 signs at the same time. The test at line l3 says that P_0 may enter its critical section if either y_1 equals 0 – implying that its competitor is not interested in entering its critical section – or if s differs from 0 – implying that the competitor process P_1 performed its assignment to y_1 after p_0 assigned 1 to y_0.

Questions concerning this mutual exclusion protocol:

(a) Model this protocol in NuSMV; formulate the property of mutual exclusion in CTL and check this property.

(b) Check whether Pnueli's protocol ensures absence of unbounded overtaking, i.e., when a process wants to enter its critical section, it eventually will be able to do so. Provide a counterexample (and an explanation thereof) in case this property is violated.

(c) Add the fairness constraint `FAIRNESS running` to the process specification in your NuSMV model of the mutual exclusion program, and check again the property of absence of unbounded overtaking. Compare the obtained results with the results obtained in the previous question without using the fairness constraint.

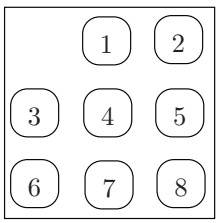
(d) Express in CTL that each process will occupy its critical section infinitely often. Check the property (use again the `FAIRNESS running`).

(e) A practical problem with this mutual exclusion protocol is that it is too demanding in the sense that it enforces performing the assignments to y_i and s (in line l2) in a single step. Most existing hardware systems cannot perform such assignments in one step. Therefore, it is requested to investigate whether any of the four possible realizations of this protocol – in which the aforementioned assignments are not atomic anymore – is a correct mutual exclusion protocol.

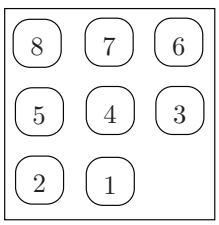
(I) Report for each possible implementation your results, including possible counterexamples and their explanation.

(II) Compare your results with the results of your PROMELA experiments with this exercise in the previous exercise series.

EXERCISE 6.32. In this exercise you are confronted with a nonstandard example for model checking. The purpose of this exercise is to present the model checker as a solver for combinatorial problems rather than a tool for correctness analysis. These problems involve a search (involving backtracking) of optimal or cost-minimizing strategies such as schedulers or puzzle solutions. The exercise is concerned with Loyd's puzzle that consists of an $N \cdot K$ grid in which there are $N \cdot K - 1$ numbered tiles and a single blank space. The goal of the puzzle is to achieve a predetermined order on the tiles. The initial and final configuration of the tiles for $N = 3$ and $K = 3$ is as follows:



initial configuration



final configuration

Note that there are approximately $4 \cdot (N \cdot K)!$ possible moves in this puzzle. For $N = 3$ and $K = 3$ this already amounts to about $1.45\ 10^6$ possible configurations.

Questions concerning Loyd's puzzle:

(a) Complete the (partial) model of Loyd's puzzle in NuSMV that is given below. In this model, there is an array `h` that keeps track of the horizontal positions of the tiles and an array `v` that records the vertical positions of the tiles such that the position of tile `i` is given by the pair `h[i], v[i]`. Tile `0` represents the blank tile. Position `h[i] = 1` and `v[i] = 1` is the lowest left corner of the puzzle.

```
MODULE main

DEFINE N := 3; K := 3;

VAR move: {u, d, l, r};     -- the possible tile-moves
    h: array 0..8 of 1..3;  -- the horizontal positions of all tiles
    v: array 0..8 of 1..3;  -- .... and their vertical positions

ASSIGN -- the initial horizontal and vertical positions of all tiles

init(h[0]) := 1;  init(v[0]) := 3;
init(h[1]) := 2;  init(v[1]) := 3;
init(h[2]) := 3;  init(v[2]) := 3;
init(h[3]) := 1;  init(v[3]) := 2;
init(h[4]) := 2;  init(v[4]) := 2;
init(h[5]) := 3;  init(v[5]) := 2;
init(h[6]) := 1;  init(v[6]) := 1;
init(h[7]) := 2;  init(v[7]) := 1;
init(h[8]) := 3;  init(v[8]) := 1;

ASSIGN

-- determine the next positions of the blank tile

next(h[0]) :=      -- horizontal position of the blank tile
       case
                              -- one position right
                              -- one position left
              1 : h[0];       -- keep the same horizontal position

       esac;

next(v[0]) :=      -- vertical position of the blank tile
       case
                              -- one position down
                              -- one position up
              1 : v[0];       -- keep the same vertical position
       esac;

-- determine the next positions of all non-blank tiles

next(h[1]) :=      -- horizontal position of tile 1
       case


       esac;
```

```
next(v[1]) :=      -- vertical position of tile 1
       case


       esac;

-- and similar for all remaining tiles
```

A possible way to proceed is as follows:

(i) First, consider the possible moves of the blank tile (i.e., the blank space). Notice that the blank space cannot be moved to the left in all positions. The same applies to moves upward, downward and to the right.

(ii) Then try to find the possible moves of tile `[1]`. The code for tiles `[2]` through `[8]` are obtained by simply copying the code for tile `[1]` while replacing all references to `[1]` with references of the appropriate tile number.

(iii) Test the possible moves by running a simulation.

(b) Define an atomic proposition `goal` that describes the desired goal configuration of the puzzle. Add this definition to your NuSMV specification by incorporating the following line(s) in your NuSMV model:

```
DEFINE goal :=   ..............  ;
```

where the dotted lines contain your description of the goal configuration.

(c) Find a solution to the puzzle by imposing the appropriate CTL formula to the NuSMV specification, and running the model checker on this formula.

This exercise has been taken from [95].

Exercise 6.33. Consider the mutual exclusion algorithm by the Dutch mathematician Dekker. There are two processes P_1 and P_2, two Boolean-valued variables b_1 and b_2 whose initial values are false, and a variable k which may take the values 1 and 2 and whose initial value is arbitrary. The ith process (i=1, 2) may be described as follows, where j is the index of the other process:

while true **do**
begin b_i := true;
  while b_j **do**
    if $k = j$ **then begin**
      b_i := false;
      while $k = j$ **do skip**;
      b_i := true
      end;
  ⟨ critical section ⟩;
  $k := j$;
  b_i := false
end

Questions:

(a) Model Dekker's algorithm in NuSMV.

(b) Verify whether this algorithm satisfies the following properties:

(c) Mutual exclusion: two processes cannot be in their critical section at the same time.

(d) Absence of individual starvation: if a process wants to enter its critical section, it is eventually able to do so.

(*Hint: use the* `FAIRNESS running` *statement in your NuSMV specification for proving the latter property in order to prohibit unfair executions that might trivially violate these requirements.*)

Exercise 6.34. In the original mutual exclusion protocol by Dijkstra in 1965 (another Dutch mathematician), it is assumed that there are $n \geqslant 2$ processes, and global variables b, c : **array** $[1 \ldots n]$ of **Boolean** and an integer k. Initially all elements of b and of c have the value true and the value of k belongs to $1, 2, \ldots, n$. The ith process may be represented as follows:

```
var j : integer;
while true do
begin b[i] := false;
      L_i : if k ≠ i then begin c[i] := true;
                                if b[k] then k := i;
                                goto L_i
                          end;
            else begin c[i] := false;
                       for j := 1 to n do
                       if (j ≠ i ∧ ¬(c[j])) then goto L_i
                 end
      ⟨ critical section ⟩;
      c[i] := true;
      b[i] := true
end
```

Questions:

(a) Model this algorithm in NuSMV.

(b) Check the mutual exclusion property (at most one process can be in its critical section at any point in time) in two different ways: by means of a CTL formula using `SPEC` and by using invariants. Try to check this property for n=2 through n=5 by increasing the number of processes gradually and compare the sizes of the state spaces and the runtime needed for the two ways of verifying the mutual exclusion property.

(c) Check the absence of individual starvation property: if a process wants to enter its critical section, it is eventually able to do so.

Exercise 6.35. In order to find a fair solution for N processes, Peterson proposed in 1981 the following protocol. Let $Q[1 \ldots N]$ (Q for queue) and $T[1 \ldots N]$ (T for turn), be two shared arrays which are initially 0 and 1, respectively. The variables i and j are local to the process with i containing the process number. The code of process i is as follows:

while true **do**
for $j := 1$ **to** $N - 1$ **do**
begin
    $Q[i] := j$;
    $T[j] := i$;
    wait until $(T[j] \neq i \ \vee \ (\textbf{forall}\ k \neq i.\, Q[k] < j))$
end;
$\langle$ critical section $\rangle$;
$Q[i] := 0$
end

Questions:

(a) Model Peterson's algorithm in NuSMV.

(b) Verify whether this algorithm satisfies the following properties:

  (i) Mutual exclusion.

  (ii) Absence of individual starvation.

Chapter 7

Equivalences and Abstraction

Transition systems can model a piece of software or hardware at various abstraction levels. The lower the abstraction level, the more implementation details are present. At high abstraction levels, such details are deliberately left unspecified. Binary relations between states (henceforth implementation relations) are useful to relate or to compare transition systems, possibly at different abstraction levels. When two models are related, one model is said to be refined by the other, or, reversely, the second is said to be an *abstraction* of the first. If the implementation relation is an equivalence, then it identifies all transition systems that cannot be distinguished; such models fulfill the same observable properties at the relevant abstraction level.

Implementation relations are predominantly used for comparing two models of the same system. Given a transition system TS that acts as an abstract system specification, and a more detailed system model TS', implementation relations allow checking whether TS' is a correct implementation (or: refinement) of TS. Alternatively, for system analysis purposes, implementation relations provide ample means to *abstract* from certain system details, preferably details that are irrelevant for the analysis of the property, φ say, at hand. In this way, a transition system TS' comprising very many, maybe even infinitely many, states can be abstracted by a smaller model TS. Provided the abstraction preserves the property to be checked, the analysis of the (hopefully small) abstract model TS suffices to decide the satisfaction of the properties in TS'. Formally, from $TS \models \varphi$ we may safely conclude $TS' \models \varphi$.

This chapter will introduce several implementation relations, ranging from very strict ones ("strong" relations) that require transition systems to mutually mimic each transition, to more liberal ones ("weak" relations) in which this is only required for certain transitions.

In fact, in this monograph we have already encountered implementation relations that were aimed at comparing transition systems by considering their linear-time behavior, i.e., (finite or infinite) traces. Examples of such relations are trace inclusion and trace equivalence. Linear-time properties or LTL formulae are preserved by trace relations based on infinite traces; for trace-equivalent TS and TS' and linear-time property P, we have $TS' \models P$ whenever $TS \models P$. A similar result is obtained when replacing P by an LTL formula.

Besides the introduction of a weak variant of trace equivalence, the goal of this chapter is primarily to study relations that respect the branching-time behavior. Classical representatives of such relations are *bisimulation* equivalences and *simulation* preorder relations. Whereas bisimulation relates states that mutually mimic all individual transitions, simulation requires that one state can mimic all stepwise behavior of the other, but not the reverse. Weak variants of these relations only require this for certain ("observable") transitions, and not for other ("silent") transitions. This chapter will formally define strong and weak variants of (bi)simulation, and will treat their relationship to trace-based relations. The preservation of CTL and CTL* formulae is shown; for bisimulation the truth value of all such formulae is preserved, while for simulation this applies to a (large) fragment thereof. These results provide us with the tools to simplify establishing $TS' \models \varphi$, for CTL (or CTL*) formula φ by checking whether $TS \models \varphi$.

This provides the theoretical underpinning of exploiting bisimulation and simulation relations for the purpose of abstraction. The remaining issue is how to obtain the more abstract TS from the (larger, and more concrete) transition system TS'. This chapter will treat polynomial-time algorithms for several notions of (bi)simulation. These algorithms allow checking whether two given transition systems are (bi)similar, and can be used to generate an abstract transition system from a more concrete one in an automated manner.

As in the previous chapters, we will follow the state-based approach. This means that we consider branching-time relations that refer to the state labels, i.e., the atomic propositions that hold in the states. Action labels of the transitions are not considered. All concepts (definitions, theorems, algorithms) treated in this chapter can, however, easily be reformulated for the approach that is focused on action labels rather than state labels. This connection is discussed in Section 7.1.2.

Throughout this chapter, the transition systems under consideration may have terminal states, and hence, may exhibit both finite and infinite paths and traces. When considering temporal logics, though, transition systems without terminal states (thus only having infinite behaviors) are assumed.

7.1 Bisimulation

Bisimulation equivalence aims to identify transition systems with the same branching structure, and which thus can simulate each other in a stepwise manner. Roughly speaking, a transition system TS' can simulate transition system TS if every step of TS can be matched by one (or more) steps in TS'. *Bi*simulation equivalence denotes the possibility of mutual, stepwise simulation. We first introduce bisimulation equivalence as a binary relation between transition systems (over the same set of atomic propositions); later on, bisimulation is also treated as a relation between states of a single transition system. Bisimulation is defined coinductively, i.e., as the largest relation satisfying certain properties.

Definition 7.1. Bisimulation Equivalence

Let $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP, L_i)$, $i = 1, 2$, be transition systems over AP. A *bisimulation* for (TS_1, TS_2) is a binary relation $\mathcal{R} \subseteq S_1 \times S_2$ such that

(A) $\forall s_1 \in I_1\ (\exists s_2 \in I_2.\ (s_1, s_2) \in \mathcal{R})$ and $\forall s_2 \in I_2\ (\exists s_1 \in I_1.\ (s_1, s_2) \in \mathcal{R})$

(B) for all $(s_1, s_2) \in \mathcal{R}$ it holds:

(1) $L_1(s_1) = L_2(s_2)$

(2) if $s_1' \in Post(s_1)$ then there exists $s_2' \in Post(s_2)$ with $(s_1', s_2') \in \mathcal{R}$

(3) if $s_2' \in Post(s_2)$ then there exsist $s_1' \in Post(s_1)$ with $(s_1', s_2') \in \mathcal{R}$.

TS_1 and TS_2 are *bisimulation-equivalent* (bisimilar, for short), denoted $TS_1 \sim TS_2$, if there exists a bisimulation $\mathcal{R}$ for (TS_1, TS_2). ■

Condition (A) asserts that every initial state of TS_1 is related to an initial state of TS_2, and vice versa. According to condition (B.1), the states s_1 and s_2 are equally labeled. This can be considered as ensuring the "local" equivalence of s_1 and s_2. Condition (B.2) states that every outgoing transition of s_1 must be matched by an outgoing transition of s_2; the reverse is stated by (B.3). Figure 7.1 summarises the latter two conditions.

Example 7.2. Two Beverage Vending Machines

Let $AP = \{ pay, beer, soda \}$. Consider the transition systems depicted in Figure 7.2. These model a beverage vending machine, but differ in the number of possibilities for supplying beer. The fact that the right-hand transition system (TS_2) has an additional option to deliver beer is not observable by a user. This suggests an equivalence between

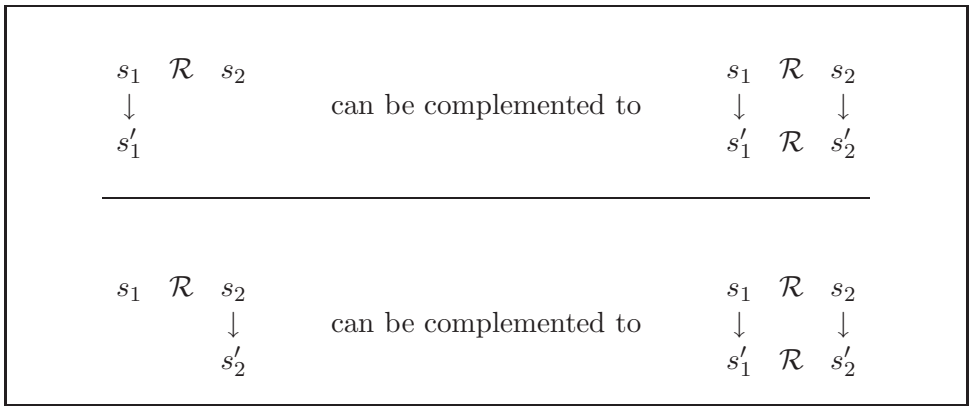


Figure 7.1: Conditions (B.2) and (B.3) of bisimulation equivalence.

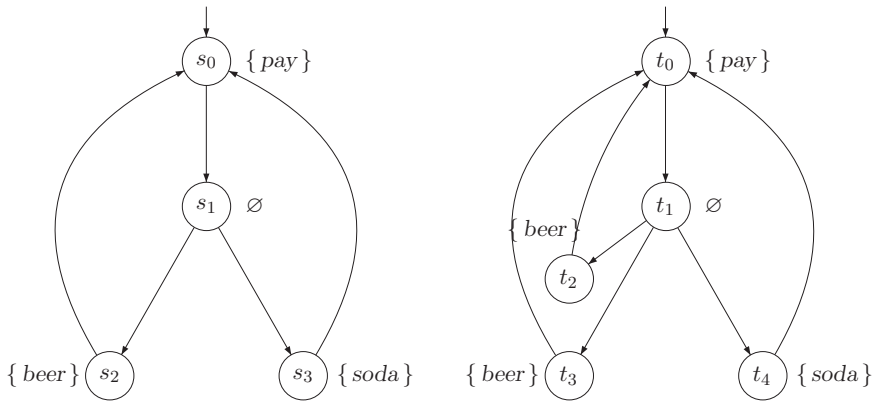


Figure 7.2: Two bisimilar beverage vending machines

the transition systems. Indeed, the equivalence of TS_1 and TS_2 follows from the fact that the relation

$$\mathcal{R} \;=\; \{(s_0,t_0),(s_1,t_1),(s_2,t_2),(s_2,t_3),(s_3,t_4)\}$$

is a bisimulation for (TS_1, TS_2). It can easily be verified that $\mathcal{R}$ indeed satisfies all requirements of Definition 7.1.

Now consider an alternative model (TS_3) of the vending machine where the user selects the drink on inserting a coin, see Figure 7.3, depicting TS_1 (left) and TS_3 (right). AP is as before. It follows that TS_1 and TS_3 are not bisimilar, since the state s_1 in TS_1 cannot be mimicked by a state in TS_3. This can be seen as follows. Due to the labeling condition (B.1), the only candidates for mimicking state s_1 are the states u_1 and u_2 in TS_3. However, neither of these states can mimic all transitions of s_1 in TS_1: either the possibility for *soda* or for *beer* is missing. Thus, $TS_1 \not\sim TS_3$ for $AP = \{\, pay, beer, soda \,\}$.

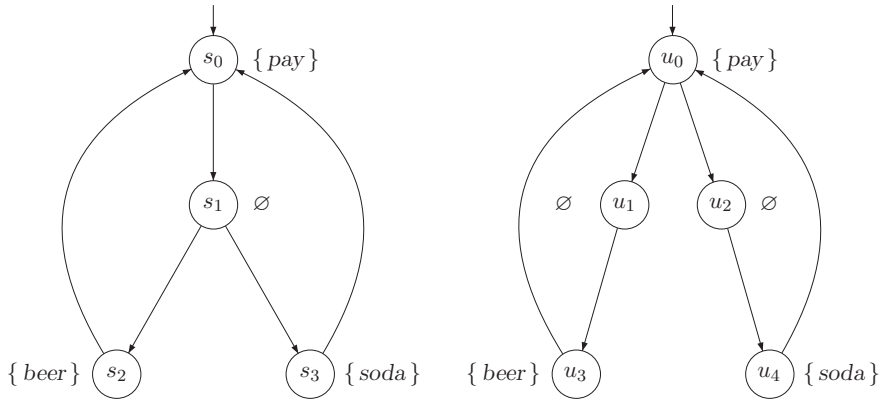


Figure 7.3: Nonbisimulation-equivalent beverage vending machines.

As a final example, reconsider TS_1 and TS_3 for $AP = \{ pay, drink \}$. The labelings of the transition systems are obvious: $L(s_0) = L(u_0) = \{ pay \}$, $L(s_1) = L(u_1) = L(u_2) = \varnothing$, and all remaining states are labeled with $\{ drink \}$. It can now be established that the relation

$$\{ (s_0, u_0), (s_1, u_1), (s_1, u_2), (s_2, u_3), (s_2, u_4), (s_3, u_3), (s_3, u_4) \}$$

is a bisimulation for (TS_1, TS_3). Henceforth, $TS_1 \sim TS_3$ for $AP = \{ pay, drink \}$. ■

Remark 7.3. The Relevant Set of Atomic Propositions

The fixed set AP plays a crucial role in comparing transition systems using bisimulation. Intuitively, AP stands for the set of all "relevant" atomic propositions. All other atomic propositions are understood as negligible and are ignored in the comparison. In case TS is a refinement of TS', e.g., TS is obtained from TS' by incorporating some implementation details, then the set AP of atomic propositions of TS generally is a proper superset of the set AP' of propositions of TS'. To compare TS and TS', the set of common atomic propositions, AP', is a reasonable choice. In this way, it is possible to check whether the branching structure of TS agrees with that of TS' when considering all observable information in AP'. If we are only interested in checking the equivalence of TS and TS' with respect to the satisfaction of a temporal logic formula Φ, it suffices to consider AP as the atomic propositions that occur in Φ. ■

Lemma 7.4. Reflexivity, Transitivity, and Symmetry of $\sim$

For a fixed set AP of atomic propositions, the relation $\sim$ is an equivalence relation.

Proof: Let AP be a set of atomic propositions. We show reflexivity, symmetry, and transitivity of $\sim$:

- Reflexivity: For transition system TS with state space S, the identity relation $\mathcal{R} = \{ (s, s) \mid s \in S \}$ is a bisimulation for (TS, TS).

- Symmetry: Sssume that $\mathcal{R}$ is a bisimulation for (TS_1, TS_2). Consider

$$\mathcal{R}^{-1} \;=\; \{(s_2, s_1) \mid (s_1, s_2) \in \mathcal{R}\}$$

  that is obtained by swapping the states in any pair in $\mathcal{R}$. Clearly, relation $\mathcal{R}^{-1}$ satisfies conditions (A) and (B.1). By symmetry of (B.2) and (B.3), we immediately conclude that $\mathcal{R}^{-1}$ is a bisimulation for (TS_2, TS_1).

- Transitivity: Let $\mathcal{R}_{1,2}$ and $\mathcal{R}_{2,3}$ be bisimulations for (TS_1, TS_2) and $(TS_2,\ TS_3)$, respectively. The relation $\mathcal{R} = \mathcal{R}_{1,2} \circ \mathcal{R}_{2,3}$, given by

$$\mathcal{R} \;=\; \{(s_1, s_3) \mid \exists s_2 \in S_2.\ (s_1, s_2) \in \mathcal{R}_{1,2} \ \wedge\ (s_2, s_3) \in \mathcal{R}_{2,3} \},$$

  is a bisimulation for (TS_1, TS_3) where S_2 denotes the set of states in TS_2. This can be seen by checking all conditions for a bisimulation.

  (A) Consider the initial state s_1 of TS_1. Since $\mathcal{R}_{1,2}$ is a bisimulation, there is an initial state s_2 of TS_2 with $(s_1, s_2) \in \mathcal{R}_{1,2}$. As $\mathcal{R}_{2,3}$ is a bisimulation, there is an initial state s_3 of TS_3 with $(s_2, s_3) \in \mathcal{R}_{2,3}$. Thus, $(s_1, s_3) \in \mathcal{R}$. In the same way we can check that for any initial state s_3 of TS_3, there is an initial state s_1 of TS_1 with $(s_1, s_3) \in \mathcal{R}$.

  (B.1) By definition of $\mathcal{R}$, there is a state s_2 in TS_2 with $(s_1, s_2) \in \mathcal{R}_{1,2}$ and $(s_2, s_3) \in \mathcal{R}_{2,3}$. Then, $L_1(s_1) = L_2(s_2) = L_3(s_3)$.

  (B.2) Assume $(s_1, s_3) \in \mathcal{R}$. As $(s_1, s_2) \in \mathcal{R}_{1,2}$, it follows that if $s_1' \in \mathit{Post}(s_1)$ then $(s_1', s_2') \in \mathcal{R}_{1,2}$ for some $s_2' \in \mathit{Post}(s_2)$. Since $(s_2, s_3) \in \mathcal{R}_{2,3}$, we have $(s_2', s_3') \in \mathcal{R}_{2,3}$ for some $s_3' \in \mathit{Post}(s_3)$. Hence, $(s_1', s_3') \in \mathcal{R}$.

  (B.3) Similar to the proof for (B.2).

■

Bisimulation is defined in terms of the direct successors of states. By using an inductive argument over states, we obtain a relation between (finite or infinite) paths.

Lemma 7.5. Bisimulation on Paths

Let TS_1 and TS_2 be transition systems over AP, $\mathcal{R}$ a bisimulation for (TS_1, TS_2), and $(s_1, s_2) \in \mathcal{R}$. Then for each (finite or infinite) path $\pi_1 = s_{0,1}\, s_{1,1}\, s_{2,1}\, \ldots \in \mathit{Paths}(s_1)$ there exists a path $\pi_2 = s_{0,2}\, s_{1,2}\, s_{2,2}\, \ldots \in \mathit{Paths}(s_2)$ of the same length such that $(s_{j,1}, s_{j,2}) \in \mathcal{R}$ for all j.

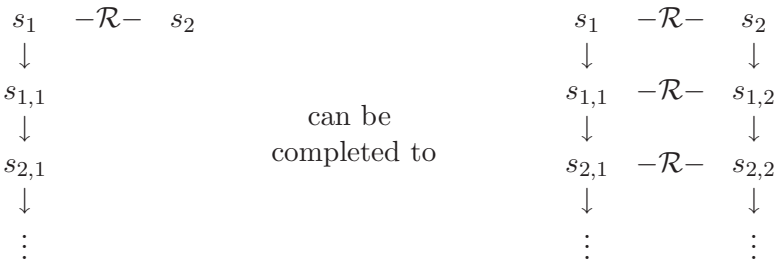


Figure 7.4: Construction of statewise bisimilar paths.

Proof: Let $\pi_1 = s_{0,1}\, s_{1,1}\, s_{2,1} \ldots \in \mathit{Paths}(s_1)$ be a maximal path fragment in TS_1 starting in $s_1 = s_{0,1}$ and assume $(s_1, s_2) \in \mathcal{R}$. We successively define a "corresponding" maximal path fragment in TS_2 starting in $s_2 = s_{0,2}$, where the transitions $s_{i,1} \to_1 s_{i+1,1}$ are matched by transitions $s_{i,2} \to_2 s_{i+1,2}$ such that $(s_{i+1,1}, s_{i+1,2}) \in \mathcal{R}$. This is done by induction on i, see Figure 7.4 on page 455. For each case we distinguish between s_i being a terminal state or not.

- Base case: $i = 0$. In case s_1 is a terminal state, it follows directly from $(s_1, s_2) \in \mathcal{R}$ (by condition (B.3)) that s_2 is a terminal state too. Thus $s_2 = s_{0,2}$ is a maximal path fragment in TS_2. If s_1 is not a terminal state, it follows from $(s_{0,1}, s_{0,2}) = (s_1, s_2) \in \mathcal{R}$ that the transition $s_1 = s_{0,1} \to_1 s_{1,1}$ can be matched by a transition $s_2 \to_2 s_{1,2}$ such that $(s_{1,1}, s_{1,2}) \in \mathcal{R}$. This yields the path fragment $s_2\, s_{1,2}$ in TS_2.

- Induction step: Assume $i \geqslant 0$ and that the path fragment $s_2\, s_{1,2}\, s_{2,2} \ldots s_{i,2}$ is already constructed with $(s_{j,1}, s_{j,2}) \in \mathcal{R}$ for $j = 1, \ldots, i$.

  If π_1 has length i, then π_1 is maximal and $s_{i,1}$ is a terminal state. By condition (B.3), $s_{i,2}$ is terminal too. Thus, $\pi_2 = s_2\, s_{1,2}\, s_{2,2} \ldots s_{i,2}$ is a maximal path fragment in TS_2 which is statewise related to $\pi_1 = s_1\, s_{1,1}\, s_{2,1} \ldots s_{i,1}$.

  Now assume that $s_{i,1}$ is not terminal. We consider the step $s_{i,1} \to_1 s_{i+1,1}$ in π_1. Since $(s_{i,1}, s_{i,2}) \in \mathcal{R}$, there exists a transition $s_{i,2} \to_2 s_{i+1,2}$ with $(s_{i+1,1}, s_{i+1,2}) \in \mathcal{R}$. This yields a path fragment $s_2\, s_{1,2} \ldots s_{i,2}\, s_{i+1,2}$ which is statewise related to the prefix $s_1\, s_{1,1} \ldots s_{i,1}\, s_{i+1,1}$ of π_1.

■

By symmetry, for each path $\pi_2 \in \mathit{Paths}(s_2)$ there exists a path $\pi_1 \in \mathit{Paths}(s_1)$ of the same length which is statewise related to π_2. As one can construct statewise bisimilar paths, bisimilar transition systems are trace-equivalent. It is mostly easier to prove that two transition systems are bisimilar rather than prove their trace equivalence. The intuitive

reason for this discrepancy is that proving bisimulation equivalence just requires "local" reasoning about state behavior instead of considering entire paths. The following result is thus of importance in checking trace equivalence as well.

Theorem 7.6. Bisimulation and Trace Equivalence

$TS_1 \sim TS_2$ *implies* $\mathit{Traces}(TS_1) = \mathit{Traces}(TS_2)$.

Proof: Let $\mathcal{R}$ be a bisimulation for (TS_1, TS_2). By Lemma 7.5, any path $\pi_1 = s_{0,1}\, s_{1,1}\, s_{2,1} \ldots$ in TS_1 can be lifted to a path $\pi_2 = s_{0,2}\, s_{1,2}\, s_{2,2} \ldots$ in TS_2 such that $(s_{i,1}, s_{i,2}) \in \mathcal{R}$ for all indices i. According to condition (B.1), $L_1(s_{i,1}) = L_2(s_{i,2})$ for all i. Thus, $\mathit{trace}(\pi_1) = \mathit{trace}(\pi_2)$. This shows $\mathit{Traces}(TS_1) \subseteq \mathit{Traces}(TS_2)$. By symmetry, one obtains that TS_1 and TS_2 are trace equivalent. ■

As trace-equivalent transition systems fulfill the same linear-time properties, it thus now follows that bisimilar transition systems fulfill the same linear-time properties.

7.1.1 Bisimulation Quotient

So far, bisimulation has been defined as a relation between transition systems. This enables comparing different transition systems. An alternative perspective is to consider bisimulation as a relation between states within a single transition system. By considering the quotient transition system under such a relation, smaller models are obtained. This minimization recipe can be exploited for efficient model checking. In the following, we define bisimulation as a relation on states, relate it to the notion of bisimulation between transition systems, and define the quotient transition system under such relation.

Definition 7.7. Bisimulation Equivalence as Relation on States

Let $TS = (S, \mathit{Act}, \rightarrow, I, \mathit{AP}, L)$ be a transition system. A *bisimulation* for TS is a binary relation $\mathcal{R}$ on S such that for all $(s_1, s_2) \in \mathcal{R}$:

1. $L(s_1) = L(s_2)$.
2. If $s_1' \in \mathit{Post}(s_1)$, then there exists an $s_2' \in \mathit{Post}(s_2)$ with $(s_1', s_2') \in \mathcal{R}$.
3. If $s_2' \in \mathit{Post}(s_2)$, then there exists an $s_1' \in \mathit{Post}(s_1)$ with $(s_1', s_2') \in \mathcal{R}$.

States s_1 and s_2 are *bisimulation-equivalent* (or bisimilar), denoted $s_1 \sim_{TS} s_2$, if there exists a bisimulation $\mathcal{R}$ for TS with $(s_1, s_2) \in \mathcal{R}$. ■

Thus, a bisimulation (on states) for TS is a bisimulation (on transition systems) for the pair (TS, TS), except that condition (A) is not required. This condition could be ensured by adding the pairs (s, s) to $\mathcal{R}$ for any state s. Moreover, for all states s_1 and s_2 in TS it holds that

$$\underbrace{s_1 \sim_{TS} s_2}_{\text{as states of } TS \text{ (Def. 7.7)}} \quad \text{iff} \quad \underbrace{TS_{s_1} \sim TS_{s_2}}_{\text{in the sense of Def. 7.1}},$$

where TS_{s_i} denotes the transition system obtained from TS by declaring s_i as the unique initial state. Vice versa, the definition of bisimulation between transition systems (Definition 7.1) arises from Definition 7.7 as follows. Take transition systems TS_1 and TS_2 over AP, and combine them in a single transition system $TS_1 \oplus TS_2$, which basically results from the disjoint union of state spaces (see below). We then subsequently "compare" the initial states of TS_1 and TS_2 as states of the composite transition system $TS_1 \oplus TS_2$ to ensure condition (A).

The formal definition of $TS_1 \oplus TS_2$ is as follows. For $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP, L_i)$, $i = 1, 2$:

$$TS_1 \oplus TS_2 \quad = \quad (S_1 \uplus S_2, Act_1 \cup Act_2, \rightarrow_1 \cup \rightarrow_2, I_1 \cup I_2, AP, L)$$

where $\uplus$ stands for disjoint union and where $L(s) = L_i(s)$ if $s \in S_i$. Then $TS_1 \sim TS_2$ if and only if, for every initial state s_1 of TS_1, there exists a bisimilar initial state s_2 of TS_2, and vice versa. That is, $s_1 \sim_{TS_1 \oplus TS_2} s_2$. Stated in terms of equivalence classes, $TS_1 \sim TS_2$ if and only if

$$\forall C \in (S_1 \uplus S_2)/\sim_{TS_1 \oplus TS_2} . \; I_1 \cap C \neq \varnothing \quad \text{iff} \quad I_2 \cap C \neq \varnothing \quad .$$

Here, $(S_1 \uplus S_2)/\sim_{TS_1 \oplus TS_2}$denotes the quotient space with respect to $\sim_{TS_1 \oplus TS_2}$, i.e., the set of all bisimulation equivalence classes in $S_1 \uplus S_2$. The latter observation is based on the fact that $\sim_{TS_1 \oplus TS_2}$ is an equivalence relation, see the first part of the following lemma.

Lemma 7.8. Coarsest Bisimulation

For transition system $TS = (S, Act, \rightarrow, I, AP, L)$ it holds that:

1. *$\sim_{TS}$ is an equivalence relation on S.*
2. *$\sim_{TS}$ is a bisimulation for TS.*
3. *$\sim_{TS}$ is the coarsest bisimulation for TS.*

Proof: The first claim follows directly from the characterization of $\sim_{TS}$ in terms of $\sim$, and Lemma 7.4 on page 453. The last claim states that each bisimulation $\mathcal{R}$ for TS is

finer than $\sim_{TS}$; this immediately follows from the definition of $\sim_{TS}$. It remains to prove that $\sim_{TS}$ is a bisimulation for TS. We show that $\sim_{TS}$ satisfies conditions (1) and (2) of Definition 7.7. Condition (3) follows by symmetry. Let $s_1 \sim_{TS} s_2$. Then, there exists a bisimulation $\mathcal{R}$ that contains (s_1, s_2). From condition (1), it follows that $L(s_1) = L(s_2)$. Condition (2) yields that for any transition $s_1 \to s_1'$ there is a transition $s_2 \to s_2'$ with $(s_1', s_2') \in \mathcal{R}$. Hence, $s_1' \sim_{TS} s_2'$. ■

Stated differently, the relation $\sim_{TS}$ is the coarsest equivalence on the state space of TS such that equivalent states are equally labeled and can simulate each other as shown in Figure 7.5.

$$\begin{array}{ccc} s_1 & \sim_{TS} & s_2 \\ \downarrow & & \\ s_1' & & \end{array} \qquad \text{can be complemented to} \qquad \begin{array}{ccc} s_1 & \sim_{TS} & s_2 \\ \downarrow & & \downarrow \\ s_1' & \sim_{TS} & s_2' \end{array}$$

Figure 7.5: Condition (2) for bisimulation equivalence $\sim_{TS}$ on states.

Remark 7.9. Union of Bisimulations

For finite index set I and $(\mathcal{R}_i)_{i \in I}$ a family of bisimulation relations for TS, $\bigcup_{i \in I} \mathcal{R}_i$ is a bisimulation for TS too (see Exercise 7.2). Since $\sim_{TS}$ is the coarsest bisimulation for TS, $\sim_{TS}$ coincides with the union of all bisimulation relations for TS. ■

As stated before, bisimilar transition systems satisfy the same linear-time properties. Such properties—and, as we will see later, all temporal formulae that can be expressed in CTL*—can thus be checked on the quotient system instead of on the original (and possibly much larger) transition system. Before providing the definition of quotient transition systems with respect to $\sim_{TS}$, let us first fix some notations.

Notation 7.10. Equivalence Classes, Quotient Space

Let S be a set and $\mathcal{R}$ an equivalence on S. For $s \in S$, $[s]_{\mathcal{R}}$ denotes the equivalence class of state s under $\mathcal{R}$, i.e., $[s]_{\mathcal{R}} = \{ s' \in S \mid (s, s') \in \mathcal{R} \}$. Note that for $s' \in [s]_{\mathcal{R}}$ we have $[s']_{\mathcal{R}} = [s]_{\mathcal{R}}$. The set $[s]_{\mathcal{R}}$ is often referred to as the $\mathcal{R}$-equivalence class of s. The quotient space of S under $\mathcal{R}$, denoted by $S/\mathcal{R} = \{ [s]_{\mathcal{R}} \mid s \in S \}$, is the set consisting of all $\mathcal{R}$-equivalence classes. ■

Definition 7.11. Bisimulation Quotient

For transition system $TS = (S, Act, \rightarrow, I, AP, L)$ and bisimulation $\sim_{TS}$, the *quotient transition system* $TS/\sim_{TS}$ is defined by

$$TS/\sim_{TS} = (S/\sim_{TS}, \{\tau\}, \rightarrow', I', AP, L')$$

where:

- $I' = \{[s]_\sim \mid s \in I\}$,
- $\rightarrow'$ is defined by
$$\frac{s \xrightarrow{\alpha} s'}{[s]_\sim \xrightarrow{\tau} [s']_\sim},$$
- $L'([s]_\sim) = L(s)$.

■

In the sequel, $TS/\sim_{TS}$ is referred to as the *bisimulation quotient* of TS. For the sake of simplicity, we write $TS/\sim$ rather than $TS/\sim_{TS}$.

The state space of $TS/\sim$ is the quotient of S under $\sim$. The initial states in $TS/\sim$ are the $\sim$-equivalence classes of the initial states in TS. Each transition $s \rightarrow s'$ in TS induces a transition $[s]_\sim \rightarrow' [s']_\sim$. As the action label is irrelevant, these labels are omitted from now on; this is reflected in the definition by replacing any action $\alpha \in Act$ by an arbitrary action, τ say. The state-labeling function L' is well-defined, since all states in $[s]_\sim$ are equally labeled (see the definition of bisimulation). According to the definition of $\rightarrow'$ it follows for any $B, C \in S/\sim$:

$$B \rightarrow' C \quad \text{if and only if} \quad \exists s \in B.\, \exists s' \in C.\, s \rightarrow s'$$

By condition (2) of Definition 7.7, this is equivalent to

$$B \rightarrow' C \quad \text{if and only if} \quad \forall s \in B.\, \exists s' \in C.\, s \rightarrow s'.$$

The following two examples show that enormous state-space reductions may be obtained by considering the bisimulation quotient. In some cases, it even allows obtaining a finite quotient transition system for infinite transition systems.

Example 7.12. Many Printers

Consider a system consisting of n printers, each represented as extremely simplified by two states, *ready* and *print*. The initial state is *ready*, and once started, each printer alternates between being *ready* and *print*ing. The entire system is given by

$$TS_n \;=\; \underbrace{Printer \,|||\, \ldots \,|||\, Printer}_{n \text{ times}}.$$

Assume the states of TS_n are labeled with atomic propositions from the set $AP = \{\,0, 1, \ldots, n\,\}$. Intuitively, $L(s)$ denotes the number of printers available in state s, i.e., which are in the local state *ready*. The number of states of TS_n is exponential in n (it is 2^n); for n=3, TS_n is depicted in Figure 7.6, where r denotes ready, and p denotes print. The quotient transition system $TS_n/\sim$, however, only contains n+1 states. For n=3, $TS_n/\sim$ is depicted in Figure 7.7.

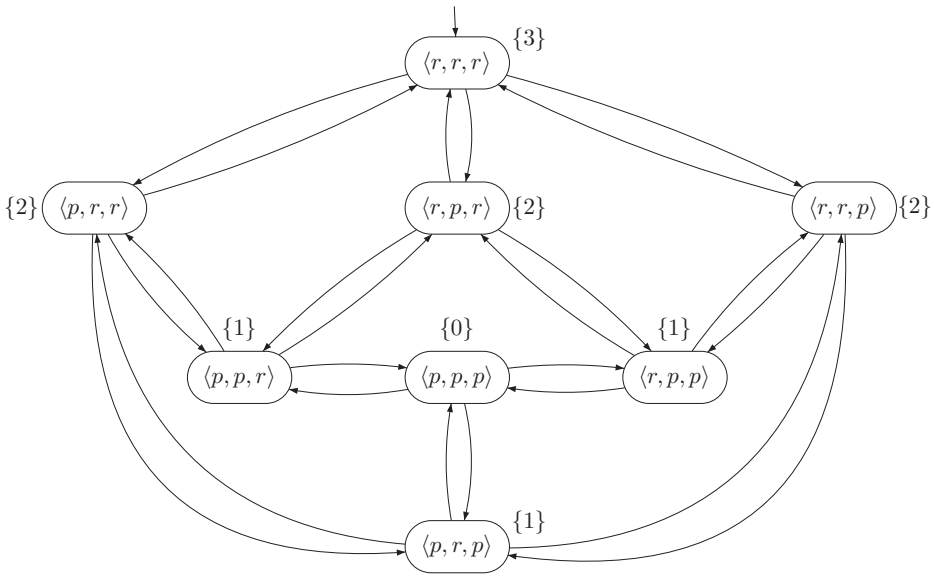


Figure 7.6: Transition system TS_3 for three independent printers

Figure 7.7: Bisimulation quotient $TS_3/\!\sim$.

■

Example 7.13. The Bakery Algorithm

We consider a mutual exclusion algorithm, originally proposed by Lamport, which is known as the *Bakery* algorithm. Although the principle of the Bakery algorithm allows guaranteeing mutual exclusion for an arbitrary number of processes, we consider the simpler setting with two processes. Let P_1 and P_2 be two processes, and x_1 and x_2 be two shared variables that both initially equal zero, see the following program text:

Process 1:		Process 2:	
	.....		.....
	while true {		**while** true {
	.....		.....
n_1 :	$x_1 := x_2 + 1;$	n_2 :	$x_2 := x_1 + 1;$
w_1 :	**wait until** $(x_2 = 0 \,\|\|\, x_1 < x_2)$ {	w_2 :	**wait until** $(x_1 = 0 \,\|\|\, x_2 < x_1)$ {
c_1 :	... critical section ...}	c_2 :	... critical section ...}
	$x_1 := 0;$		$x_2 := 0;$
	.....		.....
	}		}

These variables are used to resolve a conflict if both processes want to enter the critical section. (One might consider the value of a variable as a ticket, i.e., a number that one typically gets upon entering a bakery. The holder of the lowest number is the next customer to be served.) If process P_i is waiting, and $x_i < x_j$ or x_j=0, then it may enter the critical section. We have $x_i > 0$ whenever process P_i is either waiting to acquire access to the critical section, or is in the critical section. On requesting access to the critical section, process P_i sets x_i to x_j+1, where $i \neq j$. Intuitively, process P_i gives priority to its opponent, process P_j.

As the value of x_1 and x_2 may grow unboundedly, the underlying transition system of the parallel composition of P_1 and P_2 is infinite; a fragment of the transition system is depicted in Figure 7.8. An example of a path fragment that visits infinitely many different states is:

process P_1	process P_2	x_1	x_2	effect
$noncrit_1$	$noncrit_2$	0	0	P_1 requests access to critical section
$wait_1$	$noncrit_2$	1	0	P_2 requests access to critical section
$wait_1$	$wait_2$	1	2	P_1 enters the critical section
$crit_1$	$wait_2$	1	2	P_1 leaves the critical section
$noncrit_1$	$wait_2$	0	2	P_1 requests access to critical section
$wait_1$	$wait_2$	3	2	P_2 enters the critical section
$wait_1$	$crit_2$	3	2	P_2 leaves the critical section
$wait_1$	$noncrit_2$	3	0	P_2 requests access to critical section
$wait_1$	$wait_2$	3	4	P_2 enters the critical section
...	...	..	..	...

Although algorithmic analysis, such as LTL model checking, is impossible due to the infinity of the transition system, it is not difficult to check that the Bakery algorithm suffers neither from deadlock nor starvation:

- A deadlock only occurs whenever none of the processes can enter the critical section, i.e., if $x_1{=}x_2 > 0$. It is easy to see, however, that apart from the initial situation, we always have $x_1 \neq x_2$.
- Starvation only occurs if a process that wants to enter the critical section is never able to do so. Such situation, however, can never occur: in case both processes want to enter the critical section, it is impossible that a process acquires access to the critical section twice in a row.

Alternatively, the correctness of the Bakery algorithm can also be established algorithmically. This is done by considering an abstraction of the infinite transition system such that, instead of the concrete values of x_1 and x_2, it is only recorded whether

$$x_1 > x_2 > 0 \quad \text{or} \quad x_2 > x_1 > 0 \quad \text{or} \quad x_1 = 0 \quad \text{or} \quad x_2 = 0$$

Note that this information is sufficient to determine which process may acquire access to the critical section. By means of this *data abstraction*, we obtain a finite transition system $TS_{Bak}^{abstract}$, e.g., the infinite set of states for which x_1 and x_2 exceed 0 is mapped onto the single abstract state $\langle wait_1, wait_2, x_1 > x_2 > 0 \rangle$. When considering the atomic propositions

$$\{\, noncrit_i, wait_i, crit_i \mid i = 1, 2 \,\} \;\cup\; \{\, x_1 > x_2 > 0, x_2 > x_1 > 0, x_1 = 0, x_2 = 0 \,\}$$

the finite transition system $TS_{Bak}^{abstract}$ (with the obvious state labeling) is trace-equivalent to the original infinite transition system TS_{Bak}. Due to the fact that trace-equivalent transition systems satisfy the same LT properties, it follows that each LT property that is

shown to hold for the finite (abstract) transition system also holds for the original infinite transition system! The following LT properties, expressed as LTL formulae, indeed hold for $TS_{Bak}^{abstract}$:

$$\Box(\neg crit_1 \ \vee \ \neg crit_2) \quad \text{and} \quad (\Box \Diamond wait_1 \ \Rightarrow \ \Box \Diamond crit_1) \ \wedge \ (\Box \Diamond wait_2 \ \Rightarrow \ \Box \Diamond crit_2) \, .$$

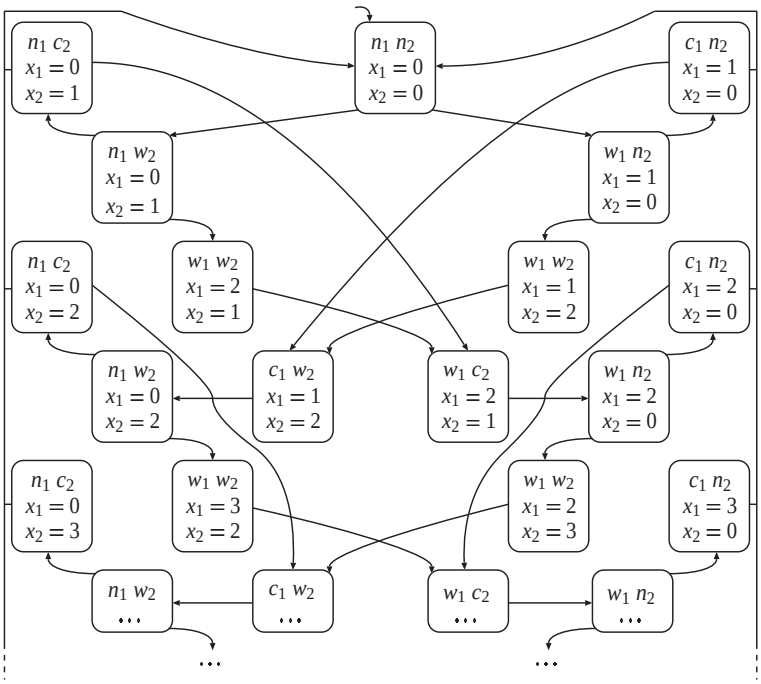


Figure 7.8: Fragment of the infinite transition system of the Bakery algorithm.

Transition systems TS_{Bak} and $TS_{Bak}^{abstract}$ are bisimilar. This can be easily proven by indicating a bisimulation relation. The data abstraction described above is formalized by means of the function $f : S \to S'$ where S and S' denote the set of reachable states of TS_{Bak} and $TS_{Bak}^{abstract}$, respectively. The function f associates to any reachable state s of TS_{Bak}, an (abstract) state $f(s)$ of $TS_{Bak}^{abstract}$. Let $s = \langle \ell_1, \ell_2, x_1 = b_1, x_2 = b_2 \rangle$ be a state of TS_{Bak} with $\ell_i \in \{ noncrit_i, wait_i, crit_i \}$ and $b_i \in \mathbb{N}$ for $i = 1, 2$. Then, we define

$$f(s) \ = \ \begin{cases} \langle \ell_1, \ell_2, x_1 = 0, x_2 = 0 \rangle & \text{if } b_1 = b_2 = 0 \\ \langle \ell_1, \ell_2, x_1 = 0, x_2 > 0 \rangle & \text{if } b_1 = 0 \text{ and } b_2 > 0 \\ \langle \ell_1, \ell_2, x_1 > 0, x_2 = 0 \rangle & \text{if } b_1 > 0 \text{ and } b_2 = 0 \\ \langle \ell_1, \ell_2, x_1 > x_2 > 0 \rangle & \text{if } b_1 > b_2 > 0 \\ \langle \ell_1, \ell_2, x_2 > x_1 > 0 \rangle & \text{if } b_2 > b_1 > 0 \quad . \end{cases}$$

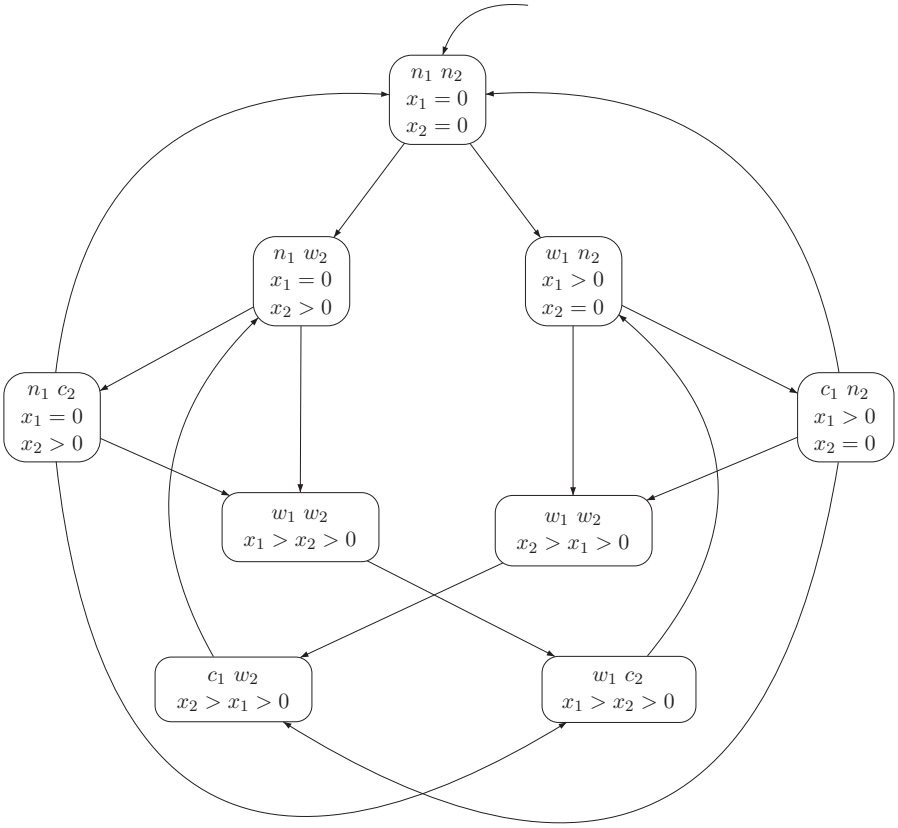


Figure 7.9: Bisimulation quotient transition system of the Bakery algorithm

It follows by straightforward reasoning that $\mathcal{R} = \{ (s, f(s)) \mid s \in S \}$ is a bisimulation for $(TS_{Bak}, TS_{Bak}^{abstract})$ for any subset of $AP = \{ noncrit_i, wait_i, crit_i \mid i = 1, 2 \}$. The transition system $TS_{Bak}^{abstract}$ in Figure 7.9 above is the bisimulation quotient system

$$TS_{Bak}^{abstract} \quad = \quad TS_{Bak}/\!\sim \quad \text{for} \quad AP = \{ crit_1, crit_2 \}.$$

Whereas the original transition system is *infinite*, its bisimulation quotient is *finite*. ■

Theorem 7.14. Bisimulation Equivalence of TS ***and*** $TS/\!\sim$

For any transition system TS it holds that $TS \sim TS/\!\sim$.

Proof: Follows immediately from the fact that $\{ (s, s') \mid s' \in [s]_\sim, s \in S \}$ is a bisimulation for $(TS, TS/\mathcal{R})$ in the sense of Definition 7.1 (page 451). ■

In general, the quotient transition system $TS/\mathcal{R}$ with respect to a bisimulation $\mathcal{R}$ contains more states than $TS/\!\sim$ since $\sim$ is the coarsest bisimulation relation. It is often simple to manually indicate a (meaningful) bisimulation, while the computation of the quotient space $S/\!\sim$ requires an algorithmic analysis of the complete transition system (see Section 7.3 on page 476 and further). Therefore, it may be advantageous to generate the quotient system $TS/\mathcal{R}$ instead of $TS/\!\sim$.

7.1.2 Action-Based Bisimulation

As the prime interest of this monograph is model checking, the notions of bisimulation are all focused on the state labelings; labels of transitions are simply ignored. In other contexts, most notably process algebras, analogous notions of bisimulation are considered that do the reverse—these notions ignore state labelings and are focused on transitions labels, i.e. actions. The aim of this subsection is to relate these notions.

Definition 7.15. Action-Based Bisimulation Equivalence

Let $TS_i = (S_i, Act, \rightarrow_i, I_i, AP_i, L_i)$, i=1, 2, be transition systems over the set Act of actions. An *action-based bisimulation* for (TS_1, TS_2) is a binary relation $\mathcal{R} \subseteq S_1 \times S_2$ such that

(A) $\forall s_1 \in I_1\, \exists s_2 \in I_2.\ (s_1, s_2) \in \mathcal{R}$ and $\forall s_2 \in I_2\, \exists s_1 \in I_1.\ (s_1, s_2) \in \mathcal{R}$

(B) for any $(s_1, s_2) \in \mathcal{R}$ it holds that

(2') if $s_1 \xrightarrow{\alpha}_1 s_1'$, then $s_2 \xrightarrow{\alpha}_2 s_2'$ with $(s_1', s_2') \in \mathcal{R}$ for some $s_2' \in S_2$

(3') if $s_2 \xrightarrow{\alpha}_2 s_2'$, then $s_1 \xrightarrow{\alpha}_1 s_1'$ with $(s_1', s_2') \in \mathcal{R}$, for some $s_1' \in S_1'$.

TS_1 and TS_2 are *action-based bisimulation equivalent* (or action-based bisimilar), denoted $TS_1 \sim^{Act} TS_2$, if there exists an action-based bisimulation $\mathcal{R}$ for (TS_1, TS_2). ■

Action-based bisimulation differs from the variant for state labels (see Definition 7.1) in the following way: the state-labeling condition (B.1) is reformulated by means of the transition labels, and thus is encoded by conditions (B.2') and (B.3'). All results and concepts presented for $\sim$ can be adapted for $\sim^{Act}$ in a straightforward manner. For instance, $\sim^{Act}$ is an equivalence and can be adapted to an equivalence $\sim_{TS}^{Act}$ for the states of a single transition system TS in a similar way as before. The action-based bisimulation quotient system $TS/\sim^{Act}$ is defined as $TS/\sim$ except that we deal with an empty set of atomic propositions and lift any transition $s \xrightarrow{\alpha} s'$ to an equally labeled transition $B \xrightarrow{\alpha} B'$

where B and B' denote the action-based bisimulation equivalence classes of states s and s', respectively.

In the context of process calculi, an important aspect of bisimulation is whether it enjoys a substitutivity property with respect to syntactic operators in the process calculus, such as parallel composition. The following result states that action-based bisimulation is a congruence for the parallel composition $\|_H$ with synchronization over handshaking actions , see Definition 2.26 on page 48.

Lemma 7.16. Congruence w.r.t. Handshaking

For transition systems TS_1, TS_1' over Act_1, TS_2, TS_2' over Act_2, and $H \subseteq Act_1 \cap Act_2$ it holds that

$$TS_1 \sim^{Act} TS_1' \quad \text{and} \quad TS_2 \sim^{Act} TS_2' \quad \text{implies } TS_1 \|_H TS_2 \sim^{Act} TS_1' \|_H TS_2'.$$

Proof: Let $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP, L_i)$ and $TS_i' = (S_i', Act_i, \rightarrow_i', I_i', AP, L_i')$ and let $\mathcal{R}_i \subseteq S_i \times S_i'$ be an action-based bisimulation for (TS_i, TS_i'), i=1, 2. Then, the relation:

$$\mathcal{R} = \{ (\langle s_1, s_2 \rangle, \langle s_1', s_2' \rangle) \mid (s_1, s_1') \in \mathcal{R}_1 \ \wedge \ (s_2, s_2') \in \mathcal{R}_2 \}$$

is an action-based bisimulation for $(TS_1 \|_H TS_2, TS_1' \|_H TS_2')$. This can be seen as follows. The fulfillment of condition (A) is obvious. To check condition (B.2'), assume that (1) there is a transition $\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle t_1, t_2 \rangle$ in $TS_1 \|_H TS_2$, and (2) $(\langle s_1, s_2 \rangle, \langle t_1, t_2 \rangle)$ in $\mathcal{R}$. Distinguish two cases.

1. $\alpha \in Act \setminus H$. Then $\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle t_1, t_2 \rangle$ arises by an individual move of either TS_1 or TS_2. By symmetry, we may assume that $s_1 \xrightarrow{\alpha}_1 t_1$ and $s_2 = t_2$. Since (s_1, s_1') belongs to bisimulation $\mathcal{R}_1$, there exists a transition $s_1' \xrightarrow{\alpha}_1' t_1'$ in TS_1' with $(t_1, t_1') \in \mathcal{R}_1$. Thus, $\langle s_1', s_2' \rangle \xrightarrow{\alpha} \langle t_1', s_2' \rangle$ is a transition in $TS_1' \|_H TS_2'$ and $(\langle t_1, s_2 \rangle, \langle t_1', s_2' \rangle) \in \mathcal{R}$.
2. $\alpha \in H$. Then $\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle t_1, t_2 \rangle$ arises by synchronizing transitions in TS_1 and TS_2. That is, $s_1 \xrightarrow{\alpha}_1 t_1$ is a transition in TS_1 and $s_2 \xrightarrow{\alpha}_2 t_2$ a transition in TS_2. Since $(s_i, s_i') \in \mathcal{R}_i$, there exists a transition $s_i' \xrightarrow{\alpha}_i' t_i'$ in TS_i' (for $i = 1, 2$) with $(t_i, t_i') \in \mathcal{R}_i$. Thus, $\langle s_1', s_2' \rangle \xrightarrow{\alpha} \langle t_1', t_2' \rangle$ is a transition in $TS_1' \|_H TS_2'$ and $(\langle t_1, t_2 \rangle, \langle t_1', t_2' \rangle) \in \mathcal{R}$.

The fulfillment of condition (B.3') follows by a symmetric argument. ■

Let us now consider the relation between state-based and action-based bisimulation in more detail. We first discuss how an action-based bisimulation can be obtained from

a state-based bisimulation, and the reverse. This is done for transition system $TS = (S, Act, \rightarrow, I, AP, L)$.

Consider the bisimulation $\sim_{TS}$ on S. The intention is to define a transition system

$$TS_{act} = (S_{act}, Act, \rightarrow_{act}, I_{act}, AP_{act}, L_{act})$$

such that $\sim_{TS}$ and the action-based bisimulation $\sim_{TS}^{Act}$ coincide. As our interest is in action-based bisimulation on TS_{act}, AP_{act} and L_{act} are irrelevant, and can be taken as AP and L, respectively. Let $S_{act} = S \cup \{ t \}$ where t is a new state (i.e., $t \notin S$). TS_{act} has the same initial states as TS, i.e., $I_{act} = I$, and is equipped with the action set $Act = 2^{AP} \cup \{ \tau \}$. The transition relation $\rightarrow_{act}$ is given by the rules:

$$\frac{s \longrightarrow s'}{s \xrightarrow{L(s)}_{act} s'} \qquad \text{and} \qquad \frac{s \text{ is a terminal state in } TS}{s \xrightarrow{\tau}_{act} t}$$

Thus, the new state t serves to characterize the terminal states in TS. For bisimulation $\mathcal{R}$ for TS, $\mathcal{R}_{act} = \mathcal{R} \cup \{ (t,t) \}$ is an action-based bisimulation for TS_{act}. Vice versa, for action-based bisimulation $\mathcal{R}_{act}$ for TS_{act}, $\mathcal{R}_{act} \cap (S \times S)$ is a bisimulation for TS. Thus, for all states $s_1, s_2 \in S$:

$$s_1 \sim_{TS} s_2 \quad \text{if and only if} \quad s_1 \sim_{TS_{act}}^{Act} s_2.$$

Let us now consider the reverse direction. Consider the action-based bisimulation $\sim_{TS}^{Act}$ on S. The intention is to define a transition system

$$TS_{state} = (S_{state}, Act_{state}, \rightarrow_{state}, I_{state}, AP_{state}, L_{state})$$

such that $\sim_{TS}^{Act}$ and the bisimulation $\sim_{TS_{state}}$ coincide. As our interest is in a state-based bisimulation, the action-set Act_{state} is not of importance. Let $S_{state} = S \cup (S \times Act)$ where it is assumed that $S \cap (S \times Act) = \varnothing$. (Such construction is also used to compare action-based vs. state-based fairness on page 263) Take $I_{state} = I$. The actions of TS serve as atomic propositions in TS_{state}, i.e., $AP_{state} = Act$. The labeling function of L_{state} is defined by $L(s) = \varnothing$ and $L(\langle s, \alpha \rangle) = \{ \alpha \}$. The transition relation $\rightarrow_{state}$ is defined by the rules:

$$\frac{s \xrightarrow{\alpha} s'}{s \longrightarrow_{state} \langle s', \alpha \rangle} \qquad \text{and} \qquad \frac{s \xrightarrow{\alpha} s', \quad \beta \in Act}{\langle s, \beta \rangle \longrightarrow_{state} \langle s', \alpha \rangle}$$

That is, state $\langle s, \alpha \rangle$ in TS_{state} serves to simulate state s in TS. In fact, the second component α indicates the action via which s is entered. It now follows that for all states $s_1, s_2 \in S$ (see Exercise 7.5):

$$s_1 \sim_{TS}^{Act} s_2 \quad \text{if and only if} \quad s_1 \sim_{TS_{state}} s_2.$$

7.2 Bisimulation and CTL* Equivalence

This section considers the equivalence relations induced by the temporal logics CTL and CTL* and discusses their connection to bisimulation equivalence. As in the chapters on temporal logic, this section is restricted to transition systems that do not have any terminal states. These transition systems thus only have infinite paths.

States in a transition system are equivalent with respect to a logic whenever these states cannot be distinguished by the truth value of any of such formulae. Stated differently, whenever there is a formula in the logic that holds in one state, but not in the other, these states are not equivalent.

Definition 7.17. CTL* Equivalence

Let TS, TS_1, and TS_2 be transition systems over AP without terminal states.

1. States s_1, s_2 in TS are *CTL*-equivalent*, denoted $s_1 \equiv_{CTL^*} s_2$, if

$$s_1 \models \Phi \quad \text{iff} \quad s_2 \models \Phi \qquad \text{for all CTL}^* \text{ state formulae over } AP.$$

2. TS_1 and TS_2 are CTL*-equivalent, denoted $TS_1 \equiv_{CTL^*} TS_2$, if

$$TS_1 \models \Phi \quad \text{iff} \quad TS_2 \models \Phi \qquad \text{for all CTL}^* \text{ state formulae over } AP.$$

■

States s_1 and s_2 are CTL* equivalent if there does not exist a CTL* state formula that holds in s_1 and not in s_2, or, vice versa, holds in s_2, but not in s_1. This definition can easily be adapted to any subset of CTL*, e.g., s_1 and s_2 are CTL equivalent, denoted $s_1 \equiv_{CTL} s_2$, if for all CTL formulae Φ over AP, $\{ s_1, s_2 \} \subseteq Sat(\Phi)$ or $\{ s_1, s_2 \} \cap Sat(\Phi) = \varnothing$. Similarly, s_1 and s_2 are LTL-equivalent, denoted $s_1 \equiv_{LTL} s_2$, if s_1 and s_2 satisfy the same LTL formula over AP.

The fact that trace-equivalent transition systems satisfy the same LTL formulae (see Theorem 3.15 on page 104) can now be stated as follows:

Theorem 7.18. Trace Equivalence is Finer Than LTL Equivalence

$\equiv_{trace} \subseteq \equiv_{LTL}$.

Remark 7.19. Distinguishing Nonequivalent States by Formulae

Let TS be a transition system without terminal states and s_1, s_2 states in TS. If $s_1 \not\equiv_{CTL} s_2$, then there exists a CTL state formula Φ with $s_1 \models \Phi$ and $s_2 \not\models \Phi$. This follows from the definition of CTL equivalence which yields the existence of a formula Φ with (i) $s_1 \models \Phi$ and $s_2 \not\models \Phi$ or (ii) $s_1 \not\models \Phi$ and $s_2 \models \Phi$. In case (ii), one may switch from Φ to $\neg\Phi$ and obtain $s_1 \models \neg\Phi$ and $s_2 \not\models \neg\Phi$.

A corresponding result holds for CTL*, but *not* for LTL. This can be seen as follows. Assume that $s_1 \not\equiv_{LTL} s_2$ and $\mathit{Traces}(s_1)$ is a proper subset of $\mathit{Traces}(s_2)$. Then, all LTL formulae that hold for s_2 also hold for s_1. However, since there are traces in $\mathit{Traces}(s_2)$ that are not in $\mathit{Traces}(s_1)$, there exists an LTL formula φ (e.g., characterizing such trace) with $s_2 \models \varphi$, but $s_1 \not\models \varphi$. ■

The main result of this section is stated in Theorem 7.20 (see below). It asserts that CTL equivalence, CTL* equivalence, and bisimulation equivalence coincide. This theorem has various important consequences. First, and for all, it relates the notion of bisimulation equivalence to logical equivalences. As a result, bisimulation equivalence preserves all formulae that can be formulated in CTL* or CTL. In principle this result allows performing model checking on the bisimulation quotient transition system—assuming that we can obtain this in an algorithmic manner—while preserving both affirmative and negative outcomes of the model checking. If a CTL* formula holds for the quotient, it also holds for the original transition system. Moreover, if the formula is refuted by the quotient, the original transition system refutes it too. On the other hand, it indicates that a single CTL* formula that holds for one but not for the other state suffices to show the nonbisimilarity of the states.

Secondly, the fact that CTL and CTL* equivalence coincide is perhaps surprising as CTL* is strictly more expressive than CTL as CTL* subsumes LTL, but the expressivenesses of CTL and LTL are incomparable, see Theorem 6.21 on page 337. So, although the expressiveness of CTL* is strictly larger than that of CTL, their logical equivalence is the same. In particular, to show CTL* equivalence it suffices to show CTL equivalence!

Theorem 7.20. CTL*/CTL and Bisimulation Equivalence

For finite transition system TS without terminal states:

$$\sim_{TS} \quad = \quad \equiv_{CTL} \quad = \quad \equiv_{CTL^*} .$$

Proof: The proof of this central result is divided into three steps. First, it is shown that CTL equivalence is finer than bisimulation equivalence, i.e., $\equiv_{CTL} \subseteq \sim_{TS}$; see Lemma

7.21 below. Subsequently, it is proven that bisimulation equivalence is finer than CTL* equivalence, i.e., $\sim_{TS} \subseteq \equiv_{CTL^*}$; see Lemma 7.26 on page 473. The obvious fact that CTL* equivalence is finer than CTL equivalence (since CTL* subsumes CTL) completes the proof. Summarizing, we have:

$$\underbrace{\sim_{TS} \subseteq \equiv_{CTL^*}}_{\text{Lemma 7.26}} \quad \text{and} \quad \equiv_{CTL^*} \subseteq \equiv_{CTL} \quad \text{and} \quad \underbrace{\equiv_{CTL} \subseteq \sim_{TS}}_{\text{Lemma 7.21}}$$

■

Lemma 7.21. CTL Equivalence is Finer Than Bisimulation

For finite transition system TS without terminal states, and s_1, s_2 states in TS:

$$s_1 \equiv_{CTL} s_2 \quad \textit{implies} \quad s_1 \sim_{TS} s_2 \ .$$

Proof: It suffices to show that the relation:

$$\mathcal{R} \ = \ \{ (s_1, s_2) \in S \times S \mid \ s_1 \equiv_{CTL} s_2 \}$$

is a bisimulation for *TS*. This is proven by checking the conditions of being a bisimulation (see Definition 7.7 on page 456). As $\mathcal{R}$ is, by definition, an equivalence relation, it suffices to consider the first two conditions. Let $(s_1, s_2) \in \mathcal{R}$, i.e., $s_1 \equiv_{CTL} s_2$.

1. Consider the following CTL state formula Φ over AP:

   $$\Phi \ = \ \bigwedge_{a \in L(s_1)} a \ \wedge \ \bigwedge_{a \in AP \setminus L(s_1)} \neg a.$$

   Clearly, $s_1 \models \Phi$. Since $s_1 \equiv_{CTL} s_2$, it follows that $s_2 \models \Phi$. As Φ characterizes the labeling of state s_1, we immediately get $L(s_1) = L(s_2)$. Thus, condition (1) of Definition 7.7 holds.

2. For any equivalence class $C \in S/\mathcal{R}$, let CTL formula Φ_C be such that

   $(*) \qquad Sat(\Phi_C) \ = \ C.$

   Φ_C is obtained as follows. For every pair (C, D) of equivalence classes $C, D \in S/\mathcal{R}$ with $C \neq D$, let CTL formula $\Phi_{C,D}$ be such that $Sat(\Phi_{C,D}) \supseteq C$ and $Sat(\Phi_{C,D}) \cap D = \varnothing$. Since *TS* is finite, there are only finitely many equivalence classes under $\mathcal{R}$. Hence, Φ_C can be defined as the conjunction of all formulae $\Phi_{C,D}$:

$$\Phi_C \;=\; \bigwedge_{\substack{D \in S/\mathcal{R} \\ D \neq C}} \Phi_{C,D} \;.$$

Clearly, condition $(*)$ is satisfied.

Let $B \in S/\mathcal{R}$ and $s_1,\ s_2 \in B$, i.e., $(s_1, s_2) \in \mathcal{R}$ and B is the equivalence class of s_1, s_2 with respect to $\mathcal{R}$. Further, let $s_1' \in \mathit{Post}(s_1)$ and C be the equivalence class of s_1' with respect to $\mathcal{R}$, i.e., $C = [s_1']_{\mathcal{R}}$. Now we show the existence of a transition $s_2 \to s_2'$ with $(s_1', s_2') \in \mathcal{R}$.

Due to $s_1' \in \mathit{Post}(s_1) \cap C$ and $(*)$, we have $s_1 \;\models\; \exists \bigcirc\, \Phi_C$. Since $s_1 \equiv_{CTL} s_2$, we get $s_2 \;\models\; \exists \bigcirc\, \Phi_C$. Thus, there is a state $s_2' \in \mathit{Post}(s_2)$ with $s_2' \models \Phi_C$. But then equation $(*)$ yields $s_2' \in C$. Since $C = [s_1']_{\mathcal{R}}$, we obtain $(s_1', s_2') \in \mathcal{R}$.

■

Remark 7.22. Master Formulae

The proof of Lemma 7.21 relies on establishing so-called *master formulae* Φ_C for equivalence class C such that $\mathit{Sat}(\Phi_C) = C$. To indicate how to obtain such master formulae, consider the bisimulation quotient of the Bakery algorithm (see Figure 7.9 on page 464), and let $AP = \{\, \mathit{crit}_1, \mathit{crit}_2 \,\}$. Instead of considering the conjunction of formulae $\Phi_{C,D}$, master formulae can be obtained by inspecting the transition system. The equivalence class $C = \langle \mathit{wait}_1, \mathit{wait}_2, x_1 > x_2 > 0 \rangle$ is, given the atomic propositions in AP, uniquely characterized by the fact that none of the processes is currently in the critical section and the second process acquires access to the critical section immediately. A master formula Φ_C for C thus is

$$\Phi_C \;=\; \neg \mathit{crit}_1 \;\wedge\; \neg \mathit{crit}_2 \;\wedge\; \forall \bigcirc\, \mathit{crit}_2 \;.$$

A master formula for the equivalence class $D = \langle \mathit{wait}_1, \mathit{crit}_2, -- \rangle$ is

$$\Phi_D \;=\; \mathit{crit}_2 \;\wedge\; \forall \bigcirc\, \neg \mathit{crit}_2 \;.$$

It is easy to check that none of the other equivalence classes under bisimulation satisfies Φ_D. ■

Remark 7.23. Logical Characterization by a Sublogic of CTL

We stress that the proof of Lemma 7.21 only exploits the propositional fragment of CTL, i.e., atomic propositions, conjunction, and negation, and the modal operator $\exists\bigcirc$. It does not, however, rely on the presence of the until operator. Thus, bisimulation equivalence (and CTL* equivalence) agrees with the equivalence induced by a simple logic that only contains atomic propositions, negation, conjunction and $\exists\bigcirc$. In particular, any two

nonbisimilar states can be distinguished by a CTL formula that does contain neither the until operator U nor one of the derived operators $\Diamond$, $\Box$, W or R. ■

Remark 7.24. Infinite Transition Systems

Lemma 7.21 is restricted to *finite* transition systems. This restriction is reflected in the proof as follows. The master formula for equivalence class C, formula Φ_C is defined as the conjunction of CTL formulae for the form $\Phi_{C,D}$ with $C \neq D$. As CTL only allows for finite conjunctions, there should only be finitely many equivalence classes $D \neq C$. This is guaranteed, of course, if the transition system is finite. An extension of CTL with infinite conjunctions would be needed to obtain a similar result for infinite transition systems.

An example of an infinite transition system TS where CTL equivalence is not finer than bisimulation equivalence is as follows. Assume the set AP of atomic propositions is infinite. The states in TS are s_1, s_2 and the states t_A where A is a subset of AP. States s_1, s_2 are labelled with $\varnothing$, while the label of state t_A is A. State s_1 has a transition to all states t_A where $A \neq AP$, while state s_2 has transitions to all states t_A, including $A = AP$. Thus, $Post(s_1) = \{t_A \mid A \subseteq AP, A \neq AP\}$ and $Post(s_2) = \{t_A \mid A \subseteq AP\}$. States t_A have a single outgoing transition which leads to s_1. Clearly, s_1 and s_2 are not bisimulation-equivalent as s_2 has a transition to state t_{AP} but there is no successor of s_1 with the label $L(t_{AP}) = AP$. On the other hand, s_1 and s_2 are CTL equivalent with respect to AP. This follows from the fact that (1) the set of atomic propositions that appear in a given CTL formula over AP is finite and (2) s_1, s_2 are bisimulation-equivalent when shrinking the set AP' of atomic propositions to any finite subset of AP. Note that then $\mathcal{R} = \{(s_1, s_2)\} \cup \{(t_A, t_B) : A \cap AP' = B \cap AP'\}$ is a bisimulation. But then Lemma 7.26 yields that s_1, s_2 satisfy the same CTL formulae over AP'.

However, the result stated in Lemma 7.21 can also be established for (possibly infinite) transition systems that are finitely branching. This is shown in the following lemma. ■

Recall that a transition system is finitely branching if (i) the set of initial states is finite and (ii) for each state s the set of successors (i.e., the set $Post(s)$) is finite.

Lemma 7.25. CTL Equivalence is Finer Than Bisimulation (Revisited)

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a finitely branching *transition system without terminal states, and s_1, s_2 states in TS. Then $s_1 \equiv_{CTL} s_2$ implies $s_1 \sim_{TS} s_2$.*

Proof: As in the proof of Lemma 7.25 we show that

$$\mathcal{R} \;=\; \{\, (s_1, s_2) \in S \times S \mid s_1 \equiv_{CTL} s_2 \,\}$$

is a bisimulation for *TS*. Let $(s_1, s_2) \in \mathcal{R}$. These states are equally labeled, since for any atomic proposition a we have $s_1 \models a$ iff $s_2 \models a$, i.e., $a \in L(s_1)$ if and only if $a \in L(s_2)$. Condition (2) is proven by contraposition. Let $(s_1, s_2) \in \mathcal{R}$ and $s_1' \in \mathit{Post}(s_1)$. Assume there is no $s_2' \in \mathit{Post}(s_2)$ with $(s_1', s_2') \in \mathcal{R}$. Since *TS* is finitely branching, the set $\mathit{Post}(s_2)$ is finite, say it agrees with $\{ t_1, \ldots, t_k \}$. The assumption $(s_1', t_j) \notin \mathcal{R}$ and the definition of $\mathcal{R}$ yields the existence of a CTL formula Ψ_j such that

$$s_1' \models \Psi_j \quad \text{and} \quad t_j \not\models \Psi_j \quad \text{for } 0 < j \leqslant k.$$

Now consider the formula

$$\Phi \;=\; \exists \bigcirc (\Psi_1 \wedge \ldots \wedge \Psi_k).$$

Clearly, we have $s_1' \models \Psi_1 \wedge \ldots \wedge \Psi_k$. Hence, $s_1 \models \Phi$. On the other hand, $t_j \not\models \Psi_1 \wedge \ldots \wedge \Psi_k$ for $0 < j \leqslant k$. This yields $s_2 \not\models \Phi$. But then $(s_1, s_2) \notin \mathcal{R}$. Contradiction. ■

The following lemma asserts that bisimilar states are CTL* equivalent, and is not restricted to finite transition systems. Let $\pi_1 \sim_{TS} \pi_2$ denote that the path fragments π_1 and π_2 are statewise bisimilar, i.e., for $\pi_1 = s_{0,1}\, s_{1,1}\, s_{2,1} \ldots$ and $\pi_2 = s_{0,2}\, s_{1,2}\, s_{2,2} \ldots$, we have $s_{j,1} \sim_{TS} s_{j,2}$ for all $j \geqslant 0$.

Lemma 7.26. Bisimulation is Finer Than CTL* Equivalence

Let TS be a transition system TS (over AP) without terminal states, s_1, s_2 be states in TS, and π_1, π_2 infinite path fragments in TS. Then:

(a) If $s_1 \sim_{TS} s_2$, then for any CTL formula Φ: $s_1 \models \Phi$ if and only if $s_2 \models \Phi$.*

(b) If $\pi_1 \sim_{TS} \pi_2$, then for any CTL path formula φ: $\pi_1 \models \varphi$ if and only if $\pi_2 \models \varphi$.*

Proof: We simultaneously prove (a) and (b) by means of induction over the structure of the formula.

Induction basis. Let $s_1 \sim_{TS} s_2$. For $\Phi = \text{true}$, statement (a) obviously holds. Since $L(s_1) = L(s_2)$, it follows that for $\Phi = a \in AP$:

$$s_1 \models a \;\text{ iff }\; a \in L(s_1) \;\text{ iff }\; a \in L(s_2) \;\text{ iff }\; s_2 \models a\ .$$

Induction step

(a) Assume Φ_1, Φ_2, Ψ are CTL* state formulae for which proposition (a) holds, and φ to be a CTL* path formula for which proposition (b) holds. Let $s_1 \sim_{TS} s_2$. The proof is by structural induction on Φ:

Case 1: $\Phi = \Phi_1 \wedge \Phi_2$**.** The induction hypothesis for Φ_1 and Φ_2 provides

$$\begin{array}{rll} s_1 \models \Phi_1 \wedge \Phi_2 & \text{iff} & s_1 \models \Phi_1 \text{ and } s_1 \models \Phi_2 \\ & \text{iff} & s_2 \models \Phi_1 \text{ and } s_2 \models \Phi_2 \\ & \text{iff} & s_2 \models \Phi_1 \wedge \Phi_2. \end{array}$$

Case 2: $\Phi = \neg\Psi$**.** By applying the induction hypothesis for Ψ, one obtains

$$\begin{array}{rll} s_1 \models \neg\Psi & \text{iff} & s_1 \not\models \Psi \\ & \text{iff} & s_2 \not\models \Psi \\ & \text{iff} & s_2 \models \neg\Psi. \end{array}$$

Case 3: $\Phi = \exists\varphi$**.** For symmetry reasons, it suffices to show

$$s_1 \models \exists\varphi \quad \Longrightarrow \quad s_2 \models \exists\varphi.$$

Assume $s_1 \models \exists\varphi$. Then there exists $\pi_1 = s_{0,1}\, s_{1,1}\, s_{2,1} \ldots \in \mathit{Paths}(s_1)$ starting in $s_1 = s_{0,1}$, with $\pi_1 \models \varphi$. According to the path-lifting lemma (Lemma 7.5 on page 454) there exists a path $\pi_2 = s_{0,2}\, s_{1,2}\, s_{2,2} \ldots \in \mathit{Paths}(s_2)$ starting in $s_2 = s_{0,2}$, such that $\pi_1 \sim_{TS} \pi_2$. From the induction hypothesis (on part (b)), it follows that $\pi_2 \models \varphi$. Thus, $s_2 \models \exists\varphi$.

(b) Assume statement (a) holds for CTL* state formula Φ, and (b) holds for CTL* path formulae φ_1, φ_2, and ψ. Let $\pi_1 \sim_{TS} \pi_2$. Recall that $\pi_i[j \ldots]$ denotes the suffix $s_{j,i}\, s_{j+1,i}\, s_{j+2,i} \ldots$ of π_i. The proof proceeds by structural induction on φ:

Case 1: $\varphi = \Phi$**.** It follows from the CTL* semantics and the induction hypothesis on Φ that

$$\pi_1 \models \varphi \;\text{ iff }\; s_{0,1} \models \Phi \;\text{ iff }\; s_{0,2} \models \Phi \;\text{ iff }\; \pi_2 \models \varphi.$$

Case 2: $\varphi = \varphi_1 \wedge \varphi_2$**.** By applying the induction hypothesis to φ_1 and φ_2 we obtain

$$\begin{array}{rll} \pi_1 \models \varphi_1 \wedge \varphi_2 & \text{iff} & \pi_1 \models \varphi_1 \text{ and } \pi_1 \models \varphi_2 \\ & \text{iff} & \pi_2 \models \varphi_1 \text{ and } \pi_2 \models \varphi_2 \\ & \text{iff} & \pi_2 \models \varphi_1 \wedge \varphi_2. \end{array}$$

Case 3: $\varphi = \neg\psi$**.** We apply the induction hypothesis to ψ and obtain

$$\begin{array}{rll} \pi_1 \models \neg\psi & \text{iff} & \pi_1 \not\models \psi \\ & \text{iff} & \pi_2 \not\models \psi \\ & \text{iff} & \pi_2 \models \neg\psi. \end{array}$$

Case 4: $\varphi = \bigcirc\psi$**.** The induction hypothesis for ψ and the path fragments $\pi_l[1 \ldots]$, $l = 1, 2$, provides

$$\begin{array}{rll} \pi_1 \models \bigcirc\psi & \text{iff} & \pi_1[1 \ldots] \models \psi \\ & \text{iff} & \pi_2[1 \ldots] \models \psi \\ & \text{iff} & \pi_2 \models \bigcirc\psi. \end{array}$$

Case 5: $\varphi = \varphi_1 \,\mathsf{U}\, \varphi_2$**.** The induction hypothesis for φ_1 and φ_2 and the path fragments $\pi_l[i\ldots]$, $l = 1, 2$, $i = 0, 1, 2, \ldots$, provides

$$\begin{aligned}
\pi_1 \models \varphi_1 \,\mathsf{U}\, \varphi_2 \quad \text{iff} \quad & \text{there exists an index } j \in \mathbb{N} \text{ with} \\
& \pi_1[j\ldots] \models \varphi_2 \text{ and} \\
& \pi_1[i\ldots] \models \varphi_1,\ i = 0, 1, \ldots, j{-}1 \\
\\
\text{iff} \quad & \text{there exists an index } j \in \mathbb{N} \text{ with} \\
& \pi_2[j\ldots] \models \varphi_2 \text{ and} \\
& \pi_2[i\ldots] \models \varphi_1,\ i = 0, 1, \ldots, j{-}1 \\
\\
\text{iff} \quad & \pi_2 \models \varphi_1 \,\mathsf{U}\, \varphi_2.
\end{aligned}$$

■

Corollary 7.27. CTL*/CTL- vs. Bisimulation Equivalence

For finite transition systems TS_1, TS_2 (over AP) without terminal states, the following three statements are equivalent:

(a) $TS_1 \sim TS_2$

(b) $TS_1 \equiv_{CTL} TS_2$*, i.e., TS_1 and TS_2 satisfy the same CTL formulae.*

(c) $TS_1 \equiv_{CTL} TS_2$*, i.e., TS_1 and TS_2 satisfy the same CTL* formulae.*

Thus, bisimilar transition systems are equivalent with respect to all formulae that can be expressed in CTL*. The fact that CTL equivalence is finer than bisimulation equivalence proves that bisimulation equivalence is the *coarsest* equivalence which preserves all CTL formulae. Stated differently, a relation which is strictly coarser than $\sim$ may yield a smaller quotient transition system, but cannot guarantee the preservation of all CTL formulae. The fact that CTL equivalence is finer than $\sim$ is useful to prove that two finite transition systems are *not* bisimilar—it suffices to indicate a single CTL formula which holds for one transition system, but not for the other.

Example 7.28. Distinguishing Nonbisimilar Systems by CTL Formulae

Consider the beverage vending machines TS_1 and TS_2 in Figure 7.3 on page 453. For $AP = \{\, pay, beer, soda \,\}$, we have $TS_1 \not\sim_{TS} TS_2$. In fact, they can be distinguished by the CTL formula

$$\Phi \;=\; \exists \bigcirc (\exists \bigcirc \; beer \;\wedge\; \exists \bigcirc \; soda)$$

as $TS_1 \models \Phi$, but $TS_2 \not\models \Phi$. ■

7.3 Bisimulation-Quotienting Algorithms

This section presents algorithms for obtaining the bisimulation quotient for a finite transition system TS. Such an algorithm serves two purposes. First, it can be used to verify the bisimilarity of two finite transition systems TS_1 and TS_2 by considering the quotient of $TS_1 \oplus TS_2$ (see page 457). Since bisimulation is finer than trace equivalence, algorithms that verify whether $TS_1 \sim TS_2$ can also be used as a sound, though incomplete, proof technique for proving the trace equivalence of TS_1 and TS_2. Secondly, such algorithms can be used to obtain the abstract (and thus smaller) transition system $TS/\!\sim$ in a fully automated manner. As $TS \sim TS/\!\sim$ (see Theorem 7.14 on page 464), it follows from Corollary 7.27 on page 475 that any verification result for $TS/\!\sim$—either being negative or positive—carries over to TS. This applies to any formula expressed in either LTL, CTL, or CTL*.

This section treats two algorithms for computing the bisimulation quotient $TS/\!\sim$. Both algorithms rely on a *partition refinement* technique. Roughly speaking, the finite state space S is partitioned in *blocks*, i.e., pairwise disjoint sets of states. Starting from a straightforward initial partition where, e.g., all equally labeled states form a partition, the algorithm successively refines these partitions such that ultimately partitions only contain bisimilar states. This strategy strongly resembles minimization algorithms for deterministic finite automata (DFA).

In the sequel, let $TS = (S, Act, \to, I, AP, L)$ be a finite transition system with $S \neq \varnothing$.

Definition 7.29. Partition, Block, Superblock

A *partition* for S is a set $\Pi = \{B_1, \ldots, B_k\}$ such that $B_i \neq \varnothing$ (for $0 < i \leqslant k$), $B_i \cap B_j = \varnothing$ (for $0 < i, j \leqslant k$ and $i \neq j$), and $S = \bigcup_{0<i\leqslant k} B_i$.

$B_i \in \Pi$ is called a *block*. $C \subseteq S$ is a *superblock* of Π if $C = B_{i_1} \cup \ldots \cup B_{i_\ell}$ for some $B_{i_1}, \ldots, B_{i_\ell} \in \Pi$. ■

Let $[s]_\Pi$ denote the unique block of partition Π containing s. For partitions Π_1 and Π_2 of S, Π_1 is called *finer* than Π_2, or Π_2 is called *coarser* than Π_1, if:

$$\forall B_1 \in \Pi_1 \; \exists B_2 \in \Pi_2.\; B_1 \subseteq B_2.$$

In this case, every block of Π_2 can be written as a disjoint union of blocks in Π_1. Π_1 is *strictly finer* than Π_2 (and Π_2 is strictly coarser than Π_1) if Π_1 is finer than Π_2 and $\Pi_1 \neq \Pi_2$.

Remark 7.30. Partitions and Equivalences

There is a close connection between equivalence relations and partitions. For equivalence relation $\mathcal{R}$ on S, the quotient space $S/\mathcal{R}$ is a partition for S. Vice versa, partition Π for S induces the equivalence relation:

$$\begin{aligned} \mathcal{R}_\Pi &= \{\, (s_1, s_2) \mid \exists B \in \Pi.\ s_1 \in B \ \wedge\ s_2 \in B \,\} \\ &= \{\, (s_1, s_2) \mid [s_1]_\Pi = [s_2]_\Pi \,\} \end{aligned}$$

such that $S/\mathcal{R}_\Pi$ corresponds to Π. For equivalence relation $\mathcal{R}$, the equivalence relation $\mathcal{R}_\Pi$ induced by $\Pi = S/\mathcal{R}$ corresponds to $\mathcal{R}$. ■

The partition refinement algorithm works with a representation of equivalences by means of the induced partition (i.e., the quotient space). In the initial partition $\Pi_0 = \Pi_{AP}$, each group of equally labeled states forms a block. Subsequently, the current partition Π_i is successively replaced with a finer partition Π_{i+1}, (for details, see Section 7.3.2). This iterative refinement procedure is halted as soon as Π_i cannot be further refined, i.e., when $\Pi_i = \Pi_{i+1}$. Such situation is guaranteed to occur as S is finite. The resulting partition Π_i is the bisimulation quotient space. The main steps are outlined in Algorithm 28.

Algorithm 28 Partition refinement algorithm (basic idea)

Input: finite transition system TS with state space S
Output: bisimulation quotient space $S/\!\sim$

```
Π_0 := Π_AP;                                   (* see Section 7.3.1 *)
i := 0;
                  (* loop invariant: Π_i is coarser than S/∼ and finer than Π_AP *)
repeat
   Π_{i+1} := Refine(Π_i);
   i := i + 1;
until Π_i = Π_{i−1}                            (* no further refinement possible *)
return Π_i
```

The termination of the partition refinement procedure is clear, as the partition Π_{i+1} is strictly finer than Π_i. Thus, for the induced equivalence relations $\mathcal{R}_{\Pi_i}$ we have

$$S \times S \supseteq \mathcal{R}_{\Pi_0} \supsetneq \mathcal{R}_{\Pi_1} \supsetneq \mathcal{R}_{\Pi_2} \supsetneq \ldots \supsetneq \mathcal{R}_{\Pi_i} = \sim_{TS}.$$

Due to the fact that S is finite, a partition Π_i with $\Pi_i = \Pi_{i-1}$ is reached after at most $|S|$ iterations. After $|S|$ proper refinements, any block in Π_i is a singleton, and a further refinement is thus impossible.

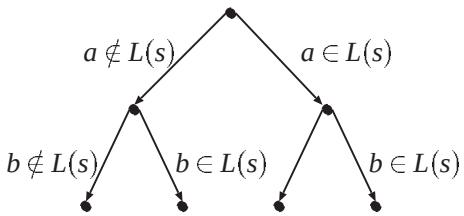
7.3.1 Determining the Initial Partition

Since bisimilar states are equally labeled, it is rather natural to use this in determining the initial partition $\Pi_0 = \Pi_{AP}$.

Definition 7.31. The AP Partition

The AP partition of TS, denoted Π_{AP}, is the quotient space $S/\mathcal{R}_{AP}$ induced by $\mathcal{R}_{AP} = \{ (s_1, s_2) \in S \times S \mid L(s_1) = L(s_2) \}$. ■

The initial partition $\Pi_0 = \Pi_{AP}$ can be computed as follows. The basic idea is to generate a decision tree for $a \in AP$. For $AP = \{ a_1, \ldots, a_k \}$, the height of the decision tree for AP is k. The vertices at depth $i < k$ represent the decision "is $a_{i+1} \in L(s)$?". The left branch of vertex v at depth $i < k$ represents the case "$a_{i+1} \notin L(s)$", while the right branch represents "$a_{i+1} \in L(s)$". The full decision tree for, e.g., $k = 2$, $a_1 = a$, and $a_2 = b$ is fo the form:



Leaf v represents a set of states. More precisely, $states(v)$ contains the states $s \in S$ for which $L(s)$ is represented by the path from the root to v. The decision tree for AP is constructed successively by considering all states in S separately. The initial decision tree consists only of the root v_0. On considering state s, the tree is traversed in a top-down manner, and new vertices are inserted when necessary, i.e., when s is the first encountered state with labeling $L(s)$. Once the tree traversal for state s reaches leaf w, $states(w)$ is extended with s. The essential steps are outlined in Algorithm 29.

Example 7.32. Determining the Initial Partition

Consider the transition system TS over $AP = \{ a, b \}$ in Figure 7.10. We assume that $L(s_0) = L(s_2) = L(s_5) = L(s_6) = \{ a \}$, $L(s_1) = \{ a, b \}$, and $L(s_3) = L(s_4) = \varnothing$. The first three steps for generating the initial partition and the resulting decision tree are indicated in Figure 7.11. The resulting partition Π_{AP} consists of three blocks. Block $B_1 = \{s_0, s_2, s_5, s_6\}$ represents all states labeled with $\{ a \}$. The two other blocks stand for the states labeled with $\{ a, b \}$ or $\varnothing$. ■

Algorithm 29 Computing the initial partition Π_{AP}

Input: $TS = (S, Act, \rightarrow, I, AP, L)$ over $AP = \{a_1, \ldots, a_k\}$ and finite S
Output: partition Π_{AP}

```
new(v_0);                                              (* create a root vertex *)
for all s ∈ S do
  v := v_0;                                            (* start a top-down traversal *)
  for i = 1,...,k−1 do
    if a_i ∈ L(s) then
      if right(v) = nil then new(right(v));            (* create a right child of v *)
      v := right(v);
    else
      if left(v) = nil then new(left(v));              (* create a left child of v *)
      v := left(v);
    fi
  od
                                                       (* v is a vertex at depth k *)
  if a_k ∈ L(s) then
    if right(v) = nil then new(right(v));              (* create a right child of v *)
    states(right(v)) := states(right(v)) ∪ { s };
  else
    if left(v) = nil then new(left(v));                (* create a left child of v *)
    states(left(v)) := states(left(v)) ∪ { s };
  fi
od
return {states(w) | w is a leaf}
```

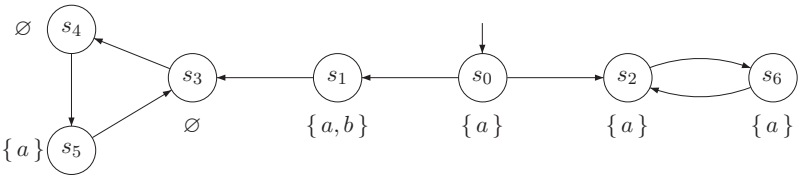


Figure 7.10: Transition system TS over $AP = \{a, b\}$.

We conclude this section by considering the time complexity of computing the initial partition. For each state s in S, the decision tree has to be traversed from root to leaf. This takes $\Theta(|AP|)$ time. The overall time complexity is now given as follows.

Lemma 7.33. Time Complexity of Computing the Initial Partition

The initial partition Π_{AP} can be computed in time $\Theta(|S| \cdot |AP|)$.

7.3.2 Refining Partitions

Since any partition Π_i $(i \geqslant 0)$ is finer than the initial partition Π_{AP}, each block in Π_i is guaranteed to contain only equally labeled states. The initial partition, however, does not consider the one-step successors of states. This is taken care of in the successive refinement steps.

Lemma 7.34. Coarsest Partition

The bisimulation quotient space $S/\!\sim$ is the coarsest partition Π for S such that

(i) Π *is finer than* Π_{AP}.

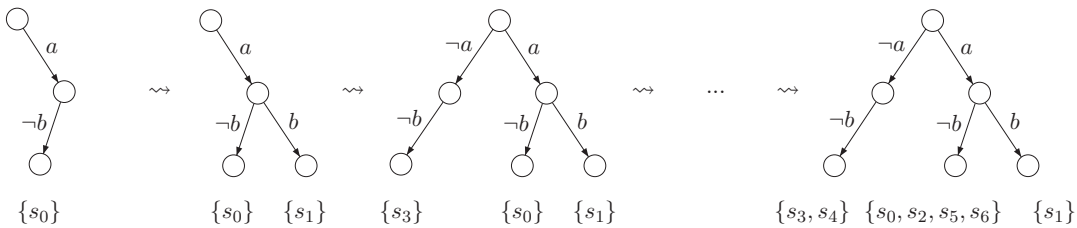


Figure 7.11: Generating the Partition Π_{AP}.

(ii) for all $B, C \in \Pi$*:* $B \cap Pre(C) = \varnothing$ *or* $B \subseteq Pre(C)$*.*

Moreover, if partition Π *fulfills (ii), then* $B \cap Pre(C) = \varnothing$ *or* $B \subseteq Pre(C)$ *for all* $B \in \Pi$ *and all superblocks* C *of* Π*.*

Recall that $Pre(C) = \{\, s \in S \mid Post(s) \cap C \neq \varnothing \,\}$ denotes the set of states in S, which have at least one successor in C. Thus, for every block B, we have that $B \cap Pre(C) = \varnothing$ if and only if *no* state of B has an immediate successor in C, and $B \subseteq Pre(C)$ if and only if *all* states in B have an immediate successor in C.

Proof: Let Π be a partition of S and $\mathcal{R}_\Pi$ the equivalence relation on S induced by Π. The proof is carried out in two steps. We first show that $\mathcal{R}_\Pi$ is a bisimulation if and only if the conditions (i) and (ii) are satisfied. Then, we show that $S/\sim$ is the coarsest partition satisfying (i) and (ii).

$\Leftarrow$: Assume that Π satisfies (i) and (ii). We prove that $\mathcal{R}_\Pi$ induced by Π is a bisimulation. Let $(s_1, s_2) \in \mathcal{R}_\Pi$ and $B = [s_1]_\Pi = [s_2]_\Pi$.

1. Since Π is finer than Π_{AP} (condition (i)), there exists a block B' of Π_{AP} containing B. Thus, $s_1, s_2 \in B \subseteq B' \in \Pi_{AP}$, and, therefore, $L(s_1) = L(s_2)$.
2. Let $s_1' \in Post(s_1)$ and $C = [s_1']_\Pi$. Then, $s_1 \in B \cap Pre(C)$. By condition (ii), we obtain $B \subseteq Pre(C)$. Hence, $s_2 \in Pre(C)$. So, there exists a state $s_2' \in Post(s_2) \cap C$. Since $s_2' \in C = [s_1']_\Pi$, it follows that $(s_1', s_2') \in \mathcal{R}_\Pi$.

$\Rightarrow$: Assume $\mathcal{R}_\Pi$ is a bisimulation. The proof obligation is to show that the conditions (i) and (ii) are satisfied.

(i) By contraposition. Assume that Π is not finer than Π_{AP}. Then, there exist a block $B \in \Pi$ and states $s_1, s_2 \in B$ with $[s_1]_{\Pi_{AP}} \neq [s_2]_{\Pi_{AP}}$. Then, $L(s_1) \neq L(s_2)$. Hence, $\mathcal{R}_\Pi$ is not a bisimulation. Contradiction.

(ii) Let B, C be blocks of Π. We assume that $B \cap Pre(C) \neq \varnothing$ and show that $B \subseteq Pre(C)$. As $B \cap Pre(C) \neq \varnothing$, there exists a state $s_1 \in B$ with $Post(s_1) \cap C \neq \varnothing$. Let $s_1' \in Post(s_1) \cap C$ and s_2 be an arbitrary state of B. We demonstrate that $s_2 \in Pre(C)$. Since $s_1, s_2 \in B$, we get that $(s_1, s_2) \in \mathcal{R}_\Pi$. Due to $s_1 \to s_1'$, there exists a transition $s_2 \to s_2'$ with $(s_1', s_2') \in \mathcal{R}_\Pi$. But then $s_1' \in C$ yields $s_2' \in C$. Hence, $s_2' \in Post(s_2) \cap C$. Thus, $s_2 \in Pre(C)$.

Assume Π satisfies (ii). We show that $B \cap Pre(C) = \varnothing$ or $B \subseteq Pre(C)$ for any $B \in \Pi$ and superblocks C of Π. The proof is by contraposition. Let $B \in \Pi$ and C a superblock, i.e., C is of the form $C = C_1 \cup \ldots \cup C_\ell$ for blocks $C_1, \ldots, C_\ell$ of Π. Assume that $B \cap Pre(C) \neq \varnothing$ and $B \not\subseteq Pre(C)$. Then, there exists an index $i \in \{1, \ldots, \ell\}$ with $B \cap Pre(C_i) \neq \varnothing$. Further, it is clear that $B \not\subseteq Pre(C_i)$, since otherwise $B \subseteq Pre(C_i) \subseteq Pre(C)$. Thus, condition (ii) is not satisfied for block $C_i \in \Pi$. Contradiction.

It remains to show that the bisimulation partition $\Pi = S/\!\sim$ is the coarsest partition of S satisfying the conditions (i) and (ii). This immediately follows from the fact that $\sim_{TS}$ is the coarsest bisimulation (Lemma 7.8 on page 457). ■

Let us now consider how partitions are successively refined. Every refinement aims at satisfying condition (ii). To that end, a superblock C of Π is considered and every block B of the current partition Π_i is decomposed ("splitted") into the subblocks $B \cap Pre(C)$ and $B \setminus Pre(C)$, provided that these subblocks are nonempty. If one of the subblocks is empty, then B satisfies condition (ii) and no decomposition takes place.

Definition 7.35. The Refinement Operator

Let Π be a partition for S and C be a superblock of Π. Then:

$$Refine(\Pi, C) \;=\; \bigcup_{B \in \Pi} Refine(B, C)$$

where $Refine(B, C) \;=\; \{B \cap Pre(C),\; B \setminus Pre(C)\} \setminus \{\varnothing\}$. ■

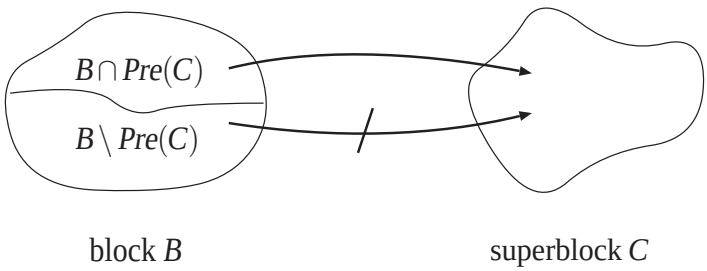


Figure 7.12: Refinement operator.

The basic scheme of the refinement operator is depicted in Figure 7.12. $Refine(B, C) = \{B\}$, i.e., B is not decomposed with respect to C, if all states in B have a direct C-successor or if no state in B has a direct C-successor. In case some states in B have a direct C-successor, while others have not, B is refined accordingly.

The following result shows that successive refinements, starting with partition Π_{AP}, yield a series of partitions $\Pi_0 = \Pi_{AP}, \Pi_1, \Pi_2, \Pi_3, \ldots$, which become increasingly finer and which all are coarser than $S/\!\sim$. This property is essential for establishing the correctness of the iterative approach for as outlined in Algorithm 28 (see page 477).

Lemma 7.36. Correctness of the Refinement Operator

Let Π be a partition of S, which is finer than Π_{AP} and coarser than $S/\!\sim$. Further, let C be a superblock for Π. Then:

(a) Refine(Π, C) is finer than Π.

(b) Refine(Π, C) is coarser than $S/\!\sim$.

Proof:

(a) This follows directly from the definition of *Refine*, since every block $B \in \Pi$ is either contained in *Refine*(Π, C) or is decomposed into $B \cap Pre(C)$ and $B \setminus Pre(C)$.

(b) Let $B \in S/\!\sim$. We show that B is contained in a block of *Refine*(Π, C). Since Π is coarser than $S/\!\sim$, there exists a block $B' \in \Pi$ with $B \subseteq B'$. B' is of the form $B' = B \cup D$ where D is a (possibly empty) superblock of $S/\!\sim$. If $B' \in$ *Refine*(Π, C), then $B \subseteq B' \in$ *Refine*(Π, C). Otherwise, i.e., if $B' \notin$ *Refine*(Π, C), then B' is decomposed into the subblocks $B' \cap Pre(C)$ and $B' \setminus Pre(C)$. We now show that B is contained in one of these two new subblocks. Condition (ii) of Lemma 7.34 implies that either $B \cap Pre(C) = \varnothing$ (thus, $B \setminus Pre(C) = B$) or $B \setminus Pre(C) = \varnothing$ (thus, $B \cap Pre(C) = B$). Since $B' = B \cup D$, B is either contained in block

- $B' \setminus Pre(C) = (B \setminus Pre(C)) \cup (D \setminus Pre(C))$
- or in $B' \cap Pre(C) = (B \cap Pre(C)) \cup (D \cap Pre(C))$.

■

Definition 7.37. Splitter, Stability

Let Π be a partition for S and C a superblock of Π.

1. C is a *splitter* for Π if there exists a block $B \in \Pi$ with $B \cap Pre(C) \neq \varnothing$ and $B \setminus Pre(C) \neq \varnothing$.

2. Block B is *stable* with respect to C if $B \cap Pre(C) = \varnothing$ or $B \setminus Pre(C) = \varnothing$.

3. Π is *stable* with respect to C if each block $B \in \Pi$ is stable wrt. C.

■

C is thus a splitter for Π if and only if $\Pi \neq Refine(\Pi, C)$, that is, if and only if Π is not stable for C. B is stable with respect to C whenever $\{ B \} = Refine(B, C)$. Note that $S/\!\sim$ is the coarsest stable partition that is finer than Π_{AP}.

Algorithm 30 Algorithm for computing the bisimulation quotient space

Input: finite transition system TS over AP with state space S
Output: bisimulation quotient space $S/\!\sim$

```
Π := Π_AP;
while there exists a splitter for Π do
  choose a splitter C for Π;
  Π := Refine(Π, C);          (* Refine(Π, C) is strictly finer than Π *)
od
return Π
```

The refine operator and the concept of a splitter can be effectively exploited in computing $S/\!\sim$, see Algorithm 30. Its correctness follows from Lemmas 7.34 and 7.36, Termination follows from the following result.

Lemma 7.38. Termination Criterion of Algorithm 30

Let Π be a partition for S, which is finer than Π_{AP} and coarser than $S/\!\sim$. Then, Π is strictly coarser than $S/\!\sim$ if and only if there exists a splitter for Π.

Proof: Follows immediately from Lemma 7.34 on page 480. ■

Example 7.39. Partition Refinement Algorithm

Figure 7.13 illustrates the principle of the partition refinement algorithm on a small example. The initial partition Π_{AP} identifies all equally labeled states:

$$\Pi_0 \;=\; \Pi_{AP} \;=\; \{\{ s_1, s_2, s_3 \}, \{ t_1, t_2, t_3 \}, \{ u_1, u_2 \}, \{ v_1, v_2 \}\}$$

In the first step, consider $C_1 = \{ v_1, v_2 \}$. This leaves the blocks $\{ s_1, s_2, s_3 \}$, $\{ u_1, u_2 \}$ and $\{ v_1, v_2 \}$ unaffected, but splits block of the t-states into $\{ t_1 \} = \{ t_1, t_2, t_3 \} \setminus Pre(C_1)$ and

$\{ t_2, t_3 \} = \{ t_1, t_2, t_3 \} \cap \mathit{Pre}(C_1)$. Thus, we obtain the partition

$$\Pi_1 = \{\{ s_1, s_2, s_3 \}, \{ t_1 \}, \{ t_2, t_3 \}, \{ u_1, u_2 \}, \{ v_1, v_2 \}\}.$$

In the second refinement step, consider $C_2 = \{ t_1 \}$. This splitter divides the s-block into

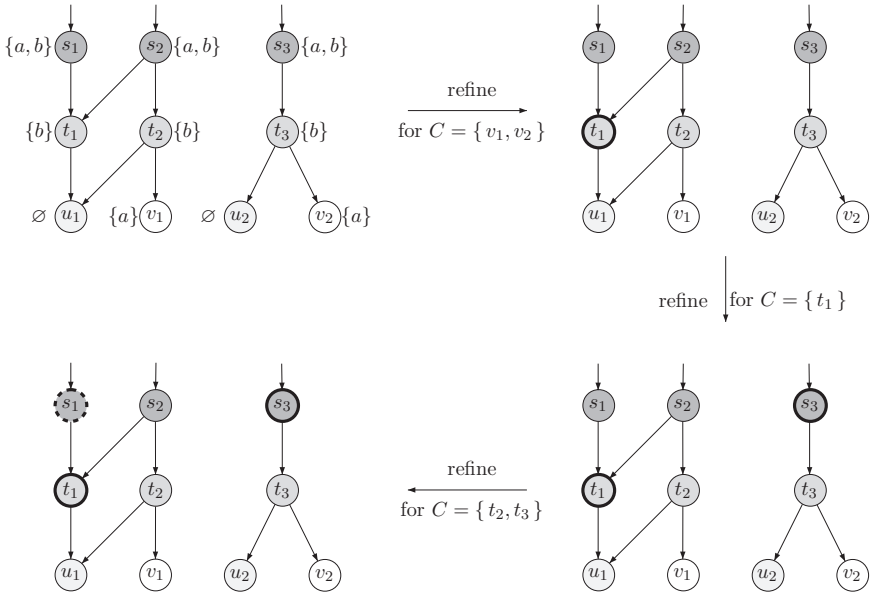


Figure 7.13: Execution of the partition refinement algorithm for a small example.

$\{ s_1, s_2 \} = \{ s_1, s_2, s_3 \} \cap \mathit{Pre}(C_2)$ and $\{ s_3 \} = \{ s_1, s_2, s_3 \} \setminus \mathit{Pre}(C_2)$, yielding

$$\Pi_2 = \{\{ s_1, s_2 \}, \{ s_3 \}, \{ t_1 \}, \{ t_2, t_3 \}, \{ u_1, u_2 \}, \{ v_1, v_2 \}\}.$$

In the third refinement step, consider $C_3 = \{ t_2, t_3 \}$. This splitter distinguishes states s_1 and s_2—s_1 has no transition to C_3, while s_2 does—and yields the partition

$$\Pi_3 = \{\{ s_1 \}, \{ s_2 \}, \{ s_3 \}, \{ t_1 \}, \{ t_2, t_3 \}, \{ u_1, u_2 \}, \{ v_1, v_2 \}\}.$$

No further refinements are possible, and Π_3 thus agrees with the bisimulation quotient of the original transition system. ■

A possible implementation of the operator $\mathit{Refine}(\Pi, C)$ is based on a list representation of the sets of predecessors $\mathit{Pre}(s')$, $s' \in S$. For any state $s' \in C$, the list of predecessors $\mathit{Pre}(s')$ is traversed, and $s \in \mathit{Pre}(s')$ is "moved" from the (data structure representing the) current block $B = [s]_\Pi$ to the block

$$[s]_{\mathit{Refine}(\Pi, C)} \;=\; B \cap \mathit{Pre}(C).$$

The states remaining in B form the subblock $B \setminus Pre(C)$. In case all states $s \in B$ are moved from B to $B \cap Pre(C)$, the set of states in the data structure representing block B is empty and we have $B \cap Pre(C) = B$ and $B \setminus Pre(C) = \varnothing$. In case no state is moved from B to $B \cap Pre(C)$, we have, $B = B \setminus Pre(C)$ and $B \cap Pre(C) = \varnothing$.

By means of the above-described technique, every state $s' \in C$ causes the costs $\mathcal{O}(|Pre(s')|+1)$, provided appropriate data structures are used to represent TS (or its state graph), and the partition Π. (The summand 1 in the complexity bound reflects the case $Pre(s') = \varnothing$; although these summands can be omitted when considering the complexity per state, they might be relevant when considering the complexity over *all* states.) Examples of such data structures are, e.g., an adjacency list representation for $Pre(\cdot)$, and a bit-vector representation of the satisfaction sets $Sat(a) = \{ s \in S \mid a \in L(s) \}$ to represent the labeling function. Furthermore, list representations of partition Π and of the blocks $B \in \Pi$ can be used. Using the fact that

$$\mathcal{O}\left(\sum_{s' \in C} (|Pre(s')| + 1)\right) \;=\; \mathcal{O}\left(|Pre(C)| + |C|\right)$$

we obtain for the time complexity of the refinement operator:

Lemma 7.40. Time Complexity of the Refinement Operator
Refine(Π, C) *can be computed in time* $\mathcal{O}\left(|Pre(C)| + |C|\right)$.

7.3.3 A First Partition Refinement Algorithm

So far we presented the partition refinement algorithm without specifying any search strategy for the splitters. Such search strategy prescribes how to determine a splitter C for a given partition Π_{i+1}. A simple strategy that will be pursued in this section, is to use the blocks of the previous partition Π_i (where $\Pi_{-1} = \{ S \}$) as splitter candidates for Π_{i+1}, see Algorithm 31. In every outermost iteration, the refine operator causes for a state $s \in S$ (as part of Π_{old}), the cost $\mathcal{O}(|Pre(s)| + 1)$, see page 486. The outermost iteration is traversed maximally $|S|$ times; this occurs when in every iteration exactly one state is separated, i.e., constitutes a separate (singleton) block. This yields for the total costs of the iteration:

$$\mathcal{O}\left(|S| \cdot \sum_{s' \in S} (|Pre(s')| + 1)\right) \quad = \quad \mathcal{O}\left(|S| \cdot (M + |S|)\right)$$

where

$$M \;=\; \sum_{s' \in S} |Pre(s')|$$

Algorithm 31 A first partition refinement algorithm

Input: finite transition system TS with the state space S
Output: bisimulation quotient space $S/\!\sim$

```
Π := Π_AP;                                   (* see Algorithm 29, page 479 *)
Π_old := { S };                              (* Π_old is the previous partition *)
                (* loop invariant: Π is coarser than S/∼ and finer than Π_AP and Π_old *)
repeat
  Π_old := Π;
  for all C ∈ Π_old do
    Π := Refine(Π, C);
  od
until Π = Π_old
return Π
```

denotes the number of edges in the state graph $G(TS)$. Assuming $M \geqslant |S|$, this can be simplified to $\mathcal{O}(|S|{\cdot}M)$. To obtain the total time complexity of Algorithm 31, it remains to consider the cost of computing Π_{AP}, which is $\Theta(|S|{\cdot}|AP|)$, as stated in Lemma 7.33 on page 480. This yields:

Theorem 7.41. Time Complexity of Algorithm 31

The bisimulation quotient space of TS can be computed by Algorithm 31 with time complexity $\mathcal{O}(|S| \cdot (|AP| + M))$, under the assumption that $M \geqslant |S|$, where M is the number of edges in the state graph $G(TS)$.

Computing the bisimulation quotient is thus linear in the number of states of TS. In the next section, a strategy is presented that improves this such that the time complexity is logarithmic in $|S|$.

7.3.4 An Efficiency Improvement

The crucial observation that allows for an improvement of the time complexity is that it is not necessary to refine with respect to *all* blocks $C \in \Pi_{old}$ as in Algorithm 31. Instead, it suffices to only consider roughly half of them, viz. the smaller subblocks of a previous refinement. More precisely, if a block C' of the current partition Π is decomposed into subblocks $C_1 = C' \cap Pre(D)$ and $C_2 = C' \setminus Pre(D)$, only the *smaller subblock* is used as a splitter candidate in the following iteration. Now, let $C \in \{ C_1, C_2 \}$ such that

$$|C| \;\leqslant\; |C'|/2, \quad \text{thus} \quad |C| \leqslant |C' \setminus C|.$$

The decomposition of the blocks $B \in \mathit{Refine}(\Pi, D)$ with respect to C and $C' \setminus C$ is now slightly modified by combining the refinement steps with respect to C and $C' \setminus C$. To make such "simultaneous" refinement possible, the algorithm exploits a ternary (instead of the previous binary) refinement operator:

$$\mathit{Refine}(\Pi, C, C' \setminus C) \;=\; \mathit{Refine}(\; \mathit{Refine}(\Pi, C),\; C' \setminus C \;)$$

where it is assumed that $|C| \leqslant |C' \setminus C|$. This modification of the refinement operator is necessary to ensure that the decomposed blocks are stable with respect to C and $C' \setminus C$. As before, the ternary refinement operator decomposes each block into subblocks:

$$\mathit{Refine}(\Pi, C, C' \setminus C) \;=\; \bigcup_{B \in \Pi} \mathit{Refine}(B, C, C' \setminus C)$$

The effect of the refinement operator $\mathit{Refine}(B, C, C' \setminus C)$ is schematically depicted for $B \subseteq \mathit{Pre}(C')$ in Figure 7.14. Thus, every block $B \in \Pi$ with $B \subseteq \mathit{Pre}(C')$ is decomposed

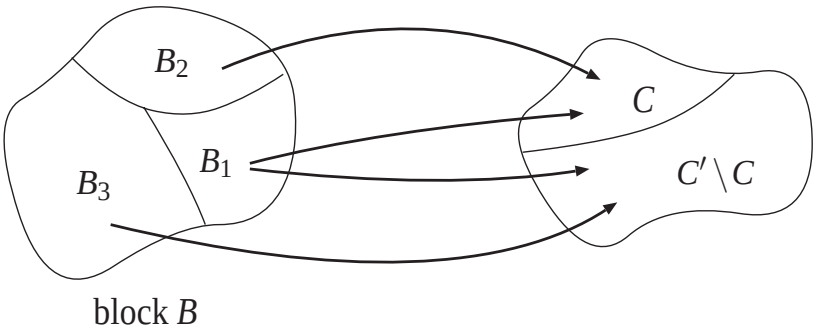


Figure 7.14: Ternary refinement operator.

into maximally *three* subblocks: states in B that only have direct successors in C, those that only have such direct successors in C', and the rest, i.e., states that have direct successors in both C and C'. Formally:

$$\mathit{Refine}(B, C, C' \setminus C) \;=\; \{\, B_1, B_2, B_3 \,\} \setminus \{\, \varnothing \,\},$$

where

$$\begin{array}{rcl} B_1 & = & B \cap \mathit{Pre}(C) \cap \mathit{Pre}(C' \setminus C), \\ B_2 & = & (B \cap \mathit{Pre}(C)) \setminus \mathit{Pre}(C' \setminus C), \\ B_3 & = & (B \cap \mathit{Pre}(C' \setminus C)) \setminus \mathit{Pre}(C). \end{array}$$

Note that the blocks B_1, B_2, B_3 are stable with respect to C and $C' \setminus C$.

If $B \cap \mathit{Pre}(C') = \varnothing$, then B is stable with respect to C and $C' \setminus C$, in which case $\mathit{Refine}(B, C, C' \setminus C) = \{\, B \,\}$. This suggests as loop invariant of the algorithm:

each block $B \in \Pi$ is stable with respect to all blocks in Π_{old}

From this invariant it follows that only the two cases $B \cap Pre(C') = \varnothing$ and $B \subseteq Pre(C')$ are possible.

Algorithm 32 outlines the main steps of the improved partition refinement algorithm. To establish the aforementioned loop invariant, the initial partition is $Refine(\Pi_{AP}, S)$, i.e., each block only contains equally-labeled states that are all either terminal or not. This is based on the observation that $Pre(S) = \{\, s \in S \mid s \text{ is nonterminal} \,\}$. Instead of first computing Π_{AP} and then applying the refinement operator, $Refine(\Pi_{AP}, S)$ can be obtained by executing Algorithm 29 with AP extended with a special symbol which identifies nonterminal states. The initial partition can thus (as before) be determined in time $\Theta(|S| \cdot |AP|)$.

Algorithm 32 An improved partition refinement algorithm

Input: finite transition system TS with state space S
Output: bisimulation quotient space $S/\!\sim$

$\Pi_{old} := \{\, S \,\}$;
$\Pi := Refine(\Pi_{AP}, S)$; (* similar to Algorithm 29, page 479 *)

(* loop invariant: Π is coarser than $S/\!\sim$ and finer than Π_{AP} and Π_{old}, *)
(* and Π is stable with respect to any block in Π_{old} *)

repeat
  $\Pi_{old} := \Pi$;
  choose block $C' \in \Pi_{old} \setminus \Pi$ and block $C \in \Pi$ with $C \subseteq C'$ and $|C| \leqslant \frac{|C'|}{2}$;
  $\Pi := Refine(\Pi, C, C' \setminus C)$;
until $\Pi = \Pi_{old}$
return Π

Example 7.42. Abstract Example of Algorithm 32

Consider the quotienting of the transition system in Figure 7.15. Since all black states are terminal, and all white states are not, we have $\Pi_{AP} = Refine(\Pi_{AP}, S)$.

In the first iteration, one may split with respect to the white or the black states. As $|\{\, u_1, \ldots, u_8, w_1, w_2, w_3 \,\}| > \frac{|S|}{2}$, the set of white states is not a suitable splitter. In fact, the only splitter candidate is block $C = \{\, v_1, v_2 \,\}$.

$$Refine(\Pi_{AP}, \underbrace{\{\, v_1, v_2 \,\}}_{C}, \underbrace{\{\, u_1, u_2, \ldots, u_8, w_1, w_2, w_3 \,\}}_{C' \setminus C})$$

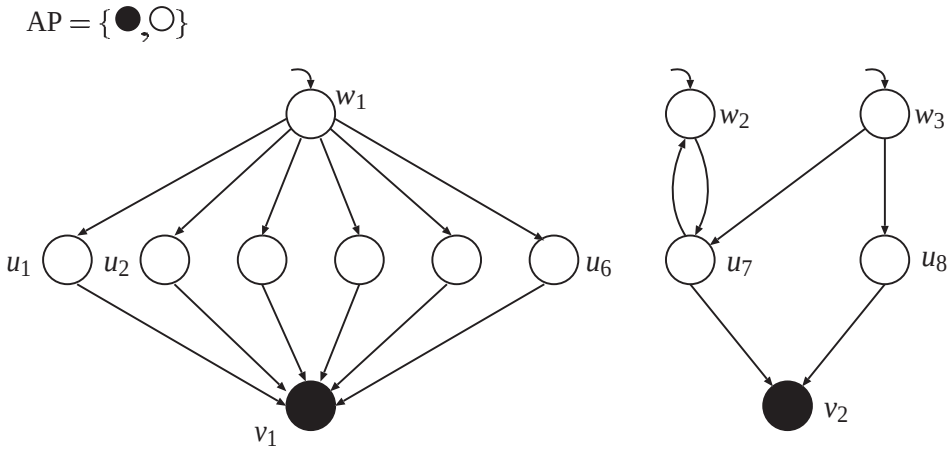


Figure 7.15: Example of a transition system.

(with $C' = S$), decomposes block $B = \{ u_1, u_2, \ldots, u_6, u_7, u_8, w_1, w_2, w_3 \}$ into

$$\begin{array}{lcll} B_1 & = & B \cap \mathit{Pre}(C) \cap \mathit{Pre}(C' \setminus C) & = \{ u_7 \}, \\ B_2 & = & (B \cap \mathit{Pre}(C)) \setminus \mathit{Pre}(C' \setminus C) & = \{ u_1, u_2, \ldots, u_6 \} \cup \{ u_8 \}, \\ B_3 & = & (B \cap \mathit{Pre}(C' \setminus C)) \setminus \mathit{Pre}(C) & = \{ w_1, w_2, w_3 \}. \end{array}$$

This yields $\Pi_{old} = \{ C, C' \setminus C \}$ and $\Pi = \{ C, B_1, B_2, B_3 \}$.

In the second iteration, the blocks $B_1 = \{ u_7 \}$ and $B_3 = \{ w_1, w_2, w_3 \}$ are potential splitters. C is not considered as $C \notin \Pi_{old} \setminus \Pi$. Block B_2 is not considered as splitter, since it is too large with respect to its superblock in Π_{old}: $|B_2| = 7 > 11/2 = \frac{|C' \setminus C|}{2}$. Suppose B_1 (called D) is selected as splitter; its superblock in Π_{old} equals $D' = C' \setminus C \in \Pi_{old}$. Figure 7.16 illustrates the partition obtained by

$$\mathit{Refine}(\Pi, \underbrace{\{ u_7 \}}_{=D}, \underbrace{\{ u_1, \ldots, u_6, u_8, w_1, w_2, w_3 \}}_{=D' \setminus D}).$$

The blocks $C = \{ v_1, v_2 \}$ and B_2 remain unchanged, since they do not have any direct successor in $D' \setminus D$. Block B_3 is refined as follows:

$$\mathit{Refine}(\{ w_1, w_2, w_3 \}, \{ u_7 \}, \{ u_1, \ldots, u_6, u_8, w_1, w_2, w_3 \}) \; = \; \{ \{ w_1 \}, \{ w_2 \}, \{ w_3 \} \}$$

since w_1 has only direct successors in $D' \setminus D$, w_2 can only move to D, and w_3 can move

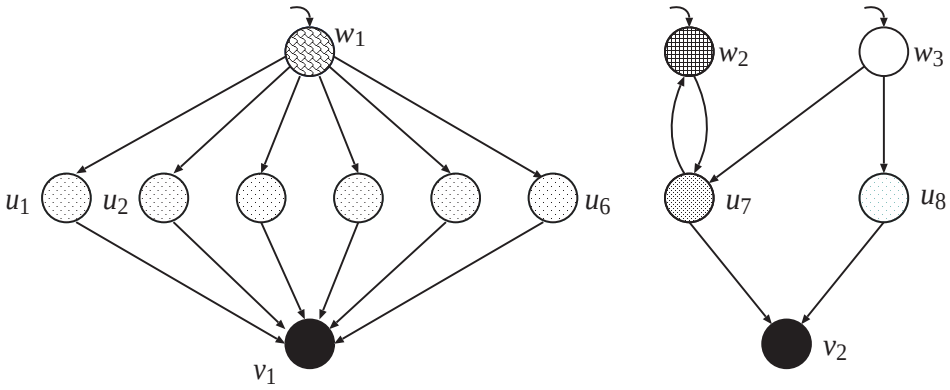


Figure 7.16: Example for Algorithm 32.

to D as well as to $D' \setminus D$. Thus, at the end of the second iteration:

$$\Pi \quad = \quad \{\{v_1, v_2\}, \{u_7\}, \{u_1, \ldots, u_6, u_8\}, \{w_1\}, \{w_2\}, \{w_3\}\},$$

$$\Pi_{old} \quad = \quad \{\underbrace{\{v_1, v_2\}}_{=C}, \underbrace{\{u_7\}}_{=D}, \underbrace{\{u_1, \ldots, u_6, u_8, w_1, w_2, w_3\}}_{=D' \setminus D}\}.$$

The next refinement does not impose any further splittings, and $\Pi = S/\!\sim$. ■

Refine$(\Pi, C, C' \setminus C)$ can be realized with time complexity $\mathcal{O}(|Pre(C)| + |C|)$ provided every state $s' \in Pre(C)$ causes the costs $\mathcal{O}(|Pre(s')| + 1)$. This can be established as follows. For every pair (s, C') with $s \in Pre(C')$, $C' \in \Pi_{old}$, a counter $\delta(s, C')$ is introduced that keeps track of the number of direct successors of s in C':

$$\delta(s, C') \quad = \quad |Post(s) \cap C'| \ .$$

During the execution of *Refine*$(\Pi, C, C' \setminus C)$, the values $\delta(s, C)$ and $\delta(s, C' \setminus C)$ are computed for any $s \in Pre(C)$ as follows:

```
for all s' ∈ C do
  for all s ∈ Pre(s') do
    δ(s, C) := δ(s, C) + 1;
  od
od
for all s ∈ Pre(C) do
  δ(s, C' \ C) := δ(s, C') − δ(s, C);
od
```

where it is assumed that initially $\delta(s, C) = 0$.

Let $B \in \Pi$ be a block with $B \subseteq Pre(C')$, which should be decomposed into B_1, B_2, B_3 by means of $Refine(B, C, C' \setminus C)$. Then B_1, B_2, and B_3 are obtained as follows:

$$\begin{array}{lcll} B_1 & = & B \cap Pre(C) \cap Pre(C' \setminus C) & = & \{\, s \in B \mid \delta(s,C) > 0, \delta(s, C' \setminus C) > 0 \,\} \\ B_2 & = & B \cap Pre(C) \setminus Pre(C' \setminus C) & = & \{\, s \in B \mid \delta(s,C) > 0, \delta(s, C' \setminus C) = 0 \,\} \\ B_3 & = & B \cap Pre(C' \setminus C) \setminus Pre(C) & = & \{\, s \in B \mid \delta(s,C) = 0, \delta(s, C' \setminus C) > 0 \,\} \end{array}$$

The decomposition of block $B \in \Pi$ is realized by "moving" the states $s \in Pre(C)$ from block B to block B_1 or B_2. The states remaining in B represent block B_3.

The initial values $\delta(s, C)$ and $\delta(s, C' \setminus C)$ need only be determined for the states in $Pre(C)$. The initial values of counters for $s \in Pre(C') \setminus Pre(C)$ are derived by $\delta(s, C' \setminus C) = \delta(s, C')$, and $\delta(s, C) = 0$. For these states, the variable $\delta(s, C)$ is not needed. Variable $\delta(s, C')$ for state $s \in Pre(C') \setminus Pre(C)$ and block C' can be identified with variable $\delta(s, C' \setminus C)$ for the new block $C' \setminus C$. These considerations lead to the following lemma:

Lemma 7.43. Time Complexity of the Ternary Refinement Operator

The time complexity of $Refine(\Pi, C, C' \setminus C)$ *is in* $\mathcal{O}(|Pre(C)| + |C|)$.

As asserted by the following theorem, the time complexity of the improved quotienting algorithm (see Algorithm 32) is logarithmic in the number of states.

Theorem 7.44. Time Complexity of Algorithm 32

The bisimulation quotient of a finite transition system $TS = (S, Act, \rightarrow, I, AP, L)$ *can be computed with Algorithm 32 in* $\mathcal{O}\left(|S| \cdot |AP| \;+\; M \cdot \log |S|\right)$.

Proof: The time complexity of Algorithm 32 is bounded above by

$$\mathcal{O}\Big(\;\; |S| \cdot |AP| \;+\; \sum_{s \in S} K(s) \cdot (|Pre(s)| + 1) \;\; \Big)$$

where $K(s')$ denotes the number of blocks C containing s' for which $Refine(\Pi, C \ldots)$ is invoked.

The first summand represents the asymptotic time needed to compute the initial partition $Refine(\Pi_{AP}, S)$. This partition can be computed in the same way as Π_{AP} by extending the set of atomic propositions with a new label that indicates the terminal states. Recall from Lemma 7.33 (page 480) that Π_{AP} can be computed in $\Theta(|S| \cdot |AP|)$. The fact that an additional atomic proposition is needed is not relevant.

The second summand stands for the cost of the refinement steps; recall that state $s \in C$ induces the costs $\mathcal{O}(|Pre(s)|+1)$ on executing $Refine(\Pi, C, C' \setminus C)$, see Lemma 7.43 (page 492). This summand can be bounded from above as follows. We first observe that

$$K(s) \;\leqslant\; \log|S| + 1 \quad \text{for any state } s.$$

This is proven as follows. Let $s \in S$ and C_i be the ith block with $s \in C_i$, for which $Refine(\Pi, C_i, \ldots)$ is executed. Then:

$$|C_{i+1}| \leqslant \frac{|C_i|}{2} \quad \text{and} \quad |C_1| \leqslant |S|.$$

Let $K(s) = k$. Then:

$$1 \;\leqslant |C_k| \leqslant \frac{|C_{k-1}|}{2} \leqslant \frac{|C_{k-2}|}{4} \leqslant \ldots \leqslant \frac{|C_{k-i}|}{2^i} \leqslant \ldots \leqslant \frac{|C_1|}{2^{k-1}} \leqslant \frac{|S|}{2^{k-1}}$$

From this, we conclude $2^{k-1} \;\leqslant\; |S|$, or equivalently $k-1 \leqslant \log|S|$. This yields $K(s) \;=\; k \;\leqslant\; \log|S|+1$.

Let $M = \sum_{s\in S} |Pre(s)|$ and assume that a representation of the sets $Pre(\cdot)$ can be obtained in time $\mathcal{O}(M)$. Thus:

$$\begin{aligned}
\sum_{s'\in S} K(s')\cdot(|Pre(s')|+1) &\leqslant (\log|S|+1)\sum_{s'\in S}(|Pre(s')|+1) \\
&= (\log|S|+1)\cdot(M+|S|) \\
&\leqslant 2\cdot(\log|S|+1)\cdot M \;=\; \mathcal{O}(M\cdot\log|S|)
\end{aligned}$$

■

7.3.5 Equivalence Checking of Transition Systems

The partition refinement algorithms can be used to verify the bisimilarity of the finite transition systems TS_1 and TS_2. To that end, the bisimulation quotient space of the composite transition system $TS = TS_1 \oplus TS_2$ (see page 457) is computed. Subsequently, it is checked whether

$$C \cap I_1 = \varnothing \quad \text{if and only if} \quad C \cap I_2 = \varnothing$$

for each bisimulation equivalence class C of the composite system TS. Here, I_i denotes the set of initial states of TS_i, $i = 1, 2$.

Corollary 7.45. Complexity of Checking Bisimulation Equivalence

Checking whether $TS_1 \sim TS_2$ for finite transition systems TS_1 and TS_2 over AP with the state spaces S_1 and S_2, respectively, can be performed in time

$$\mathcal{O}\left((|S_1|+|S_2|)\cdot|AP| \;+\; (M_1+M_2)\cdot\log(|S_1|+|S_2|)\right).$$

Here, M_i denotes the number of edges in $G(TS_i)$ and is assumed to be at least $|S_i|$ $(i = 1, 2)$.

Checking whether two transition systems are trace-equivalent is much harder. In the following theorem, the problem which asks whether two finite transition systems are trace equivalent is shown to be complete for the complexity class PSPACE. This means that the trace equivalence problem belongs to PSPACE, i.e., is solvable by a polynomial space-bounded algorithm, and is PSPACE-hard.

Theorem 7.46. Lower Bound on Checking Trace Equivalence

Let TS_1 and TS_2 be finite transition systems over AP. Then:

(a) The problem whether $Traces_{fin}(TS_1) = Traces_{fin}(TS_2)$ is PSPACE*-complete.*

(b) The problem whether $Traces(TS_1) = Traces(TS_2)$ is PSPACE*-complete.*

Proof: We first prove membership of PSPACE. We prove that problems (a) and (b) are in PSPACE by providing a polynomial reduction from these problems to the equivalence problem for nondeterministic finite automata (NFA). As the latter problem is in PSPACE, the existence of this reduction implies that problems (a) and (b) are in PSPACE. The equivalence problem for NFA $\mathcal{A}_1$ and $\mathcal{A}_2$ is whether $\mathcal{L}(\mathcal{A}_1) = \mathcal{L}(\mathcal{A}_2)$.

The main proof idea is to define NFA $\mathcal{A}_{TS}$ for transition system TS such that $\mathcal{L}(\mathcal{A}_{TS})$ can be obtained from $Traces(TS)$, and vice versa. For finite transition system $TS = (S, Act, \rightarrow, I, AP, L)$, let NFA $\mathcal{A}_{TS} = (Q, \Sigma, \delta, Q_0, F)$ be defined as follows:

- $Q = S \cup \{\, t \,\}$ with $t \notin S$ (state t identifies terminal states in TS)
- $\Sigma = 2^{AP} \cup \{\, \tau \,\}$ with $\tau \notin 2^{AP}$

- The transition relation δ in $\mathcal{A}_{TS}$ is given by

$$\delta(s,A) \;=\; \begin{cases} \mathit{Post}(s) & \text{if } s \in S,\ \mathit{Post}(s) \neq \varnothing \text{ and } L(s) = A \\ \{\, t \,\} & \text{if } (s \in S \wedge \mathit{Post}(s) = \varnothing \wedge L(s) = A) \\ & \quad \vee\ (s = t \wedge A = \tau) \\ \varnothing & \text{otherwise} \end{cases}$$

- $Q_0 = I$
- $F = Q$

It now follows that $\mathit{Traces}(TS)$ is the set consisting of all finite, nonempty words $A_0\, A_1 \ldots A_n$ where $A_0\, A_1 \ldots A_n \tau \in \mathcal{L}(\mathcal{A}_{TS})$ and all infinite words $A_0\, A_1\, A_2 \ldots \in (2^{AP})^\omega$ where all prefixes $A_0 \ldots A_n$ belong to $\mathcal{L}(\mathcal{A}_{TS})$. Hence, $\mathit{Traces}(TS)$ can be derived from $\mathcal{L}(\mathcal{A}_{TS})$, and vice versa.

For finite transition systems TS_1 and TS_2 we have

$$\mathit{Traces}(TS_1) = \mathit{Traces}(TS_2) \quad \text{ if and only if } \quad \mathcal{L}(\mathcal{A}_{TS_1}) = \mathcal{L}(\mathcal{A}_{TS_2}).$$

As the NFA $\mathcal{A}_{TS}$ are obtained in polynomial time, this yields a polynomial reduction from problem (b) to the equivalence problem for NFA.

A polynomial reduction from (a) to the equivalence problem for NFA is obtained by considering a similar transformation $TS \mapsto \mathcal{A}'_{TS}$ where $\mathcal{A}'_{TS}$ agrees with $\mathcal{A}_{TS}$, except that the self-loop at state t is omitted. State t is thus the only terminal in $\mathcal{A}'_{TS}$. Then, $\mathit{Traces}_{\mathit{fin}}(TS)$ agrees with the set of all nonempty words accepted by $\mathcal{A}'_{TS}$, i.e., $\mathit{Traces}_{\mathit{fin}}(TS) = \mathcal{L}(\mathcal{A}'_{TS}) \setminus \{\, \varepsilon \,\}$. Thus:

$$\mathit{Traces}_{\mathit{fin}}(TS_1) = \mathit{Traces}_{\mathit{fin}}(TS_2) \quad \text{ if and only if } \quad \mathcal{L}(\mathcal{A}'_{TS_1}) = \mathcal{L}(\mathcal{A}'_{TS_2}).$$

It remains to show the PSPACE-hardness. This is shown by a polynomial reduction from the language-equivalence problem for NFA, a problem that is PSPACE-complete. PSPACE-hardness of this problem follows from the fact that the (universality) problem whether $\mathcal{L}(\mathsf{E}) = \Sigma^*$ for regular expression E over Σ is PSPACE-hard; see e.g. [383]. Since there are polynomial transformations from regular expressions to NFA and there is a single-state NFA for Σ^*, the universality problem is polynomial reducible to the equivalence problem for NFA.

The main idea is to map an NFA $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ to a finite transition system $TS_{\mathcal{A}} = (S, \{\, \tau \,\}, \to, I, AP, L)$ such that $\mathit{Traces}(TS_{\mathcal{A}})$ encodes $\mathcal{L}(\mathcal{A})$. As a preprocessing step we delete all states q in $\mathcal{A}$ with $\delta^*(q, w) \cap F = \varnothing$ for any $w \in \Sigma^*$, as from these states it is impossible to reach a final state. $TS_{\mathcal{A}}$ is defined as

- $S = Q_0 \cup (Q \times \Sigma) \cup \{ \mathit{accept} \}$ with $Q_0 \cap (Q \times \Sigma) = \varnothing$ and $\mathit{accept} \notin Q \cup (Q \times \Sigma)$
- $\rightarrow$ is defined by the following rules:

$$\frac{q_0 \in Q_0 \ \wedge \ p \in \delta(q_0, A)}{q_0 \xrightarrow{\tau} \langle p, A \rangle} \qquad \frac{q \in Q \ \wedge \ B \in \Sigma \ \wedge \ p \in \delta(q, A)}{\langle q, B \rangle \xrightarrow{\tau} \langle p, A \rangle}$$

  and

$$\frac{q \in F \ \wedge \ B \in \Sigma}{\langle q, B \rangle \xrightarrow{\tau} \mathit{accept}} \qquad \frac{}{\mathit{accept} \xrightarrow{\tau} \mathit{accept}}$$

- $I = Q_0$
- $AP = \Sigma \cup \{ \mathit{accept} \}$ with $\mathit{accept} \notin \Sigma$
- $L(q_0) = \varnothing$ for any $q_0 \in Q_0$, $L(\langle q, A \rangle) = \{ A \}$, and $L(\mathit{accept}) = \{ \mathit{accept} \}$

It is not difficult to establish that $\mathit{Traces}_{\mathit{fin}}(TS_{\mathcal{A}})$ consists of all prefixes of words of the form $\varnothing\{ A_1 \} \ldots \{ A_n \} \{ \mathit{accept} \}^m$ with $m \geqslant 0$ and $A_1 \ldots A_n \in \mathcal{L}(\mathcal{A})$. $\mathcal{L}(\mathcal{A})$ can be derived from $\mathit{Traces}(TS_{\mathcal{A}})$ as follows:

$$A_1 \ldots A_n \in \mathcal{L}(\mathcal{A}) \quad \text{iff} \quad \varnothing\{ A_1 \} \ldots \{ A_n \} \{ \mathit{accept} \} \in \mathit{Traces}_{\mathit{fin}}(TS_{\mathcal{A}})$$

Thus, for NFA $\mathcal{A}_1$ and $\mathcal{A}_2$ over the alphabet Σ:

$$\begin{aligned} \mathcal{L}(\mathcal{A}_1) = \mathcal{L}(\mathcal{A}_2) \quad &\text{iff} \quad \mathit{Traces}_{\mathit{fin}}(TS_{\mathcal{A}_1}) = \mathit{Traces}_{\mathit{fin}}(TS_{\mathcal{A}_2}) \\ &\text{iff} \quad \mathit{Traces}(TS_{\mathcal{A}_1}) = \mathit{Traces}(TS_{\mathcal{A}_2}) \end{aligned}$$

The equivalence of finite trace inclusion and trace inclusion follows from Theorem 3.30 on page 117 as the transition systems $TS_{\mathcal{A}_i}$ are finite and do not have terminal states. Since $TS_{\mathcal{A}}$ is obtained in time $\mathcal{O}(|\mathcal{A}|)$ from $\mathcal{A}$, the above yields a polynomial reduction from the equivalence problem for NFA to problems (a) and (b).

■

7.4 Simulation Relations

Bisimulation relations are equivalences requiring two bisimilar states to exhibit identical stepwise behavior. On the contrary, *simulation* relations are preorders on the state space requiring that whenever $s \preceq s'$ (s' simulates s), state s' can mimic all stepwise behavior

of s; the reverse, i.e., $s' \preceq s$ is not guaranteed, so state s' may perform transitions that cannot be matched by state s. Thus, if s' simulates s then every successor of s has a corresponding, i.e., related successor of s', but the reverse does not necessarily hold. Simulation can be lifted to entire transition systems by comparing (according to $\preceq$) their initial states. Simulation relations are often used for verification purposes to show that one system correctly implements another, more abstract system. One of the interesting aspects of simulation relations is that they allow a verification by "local" reasoning. The transitivity of $\preceq$ allows a stepwise verification in which the correctness is established via several intermediate systems. Simulation relations are therefore used as a basis for abstraction techniques where the rough idea is to replace the model to be verified by a smaller abstract model and to verify the latter instead of the original one.

The simulation order $\preceq$ induces an equivalence which is coarser than bisimulation equivalence, and hence yields a better abstraction (i.e., a smaller quotient space), while still preserving a wide range of logical formulae in LTL and CTL. As bisimulation equivalence is the coarsest equivalence that preserves CTL and CTL*, the simulation order $\preceq$ preserves a fragment of these logics. The use of simulation thus relies on the preservation of certain classes of formulae, not of all formulae. For instance, if $s \preceq s'$, then for any safety property $\forall\varphi$ it follows that $s' \models \forall\varphi$ implies $s \models \forall\varphi$, since any path starting in s is mimicked by a similar path that starts in s'. The reverse, $s' \not\models \forall\varphi$, cannot be used to deduce that $\forall\varphi$ does not hold in the simulated state s; the paths starting in s' that violate φ might be behaviors that s cannot perform at all.

As for bisimulation relations, the formal definition of the simulation order relies on a coinductive approach which defines the simulation order as the greatest relation satisfying certain conditions:

Definition 7.47. Simulation Order

Let $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP, L_i)$, $i = 1, 2$, be transition systems over AP. A *simulation* for (TS_1, TS_2) is a binary relation $\mathcal{R} \subseteq S_1 \times S_2$ such that

(A) $\forall s_1 \in I_1 \,.\, (\exists s_2 \in I_2.\ (s_1, s_2) \in \mathcal{R})$

(B) for all $(s_1, s_2) \in \mathcal{R}$ it holds that:

(1) $L_1(s_1) \;=\; L_2(s_2)$

(2) if $s_1' \in \mathit{Post}(s_1)$, then there exists $s_2' \in \mathit{Post}(s_2)$ with $(s_1', s_2') \in \mathcal{R}$.

TS_1 is simulated by TS_2 (or, equivalently, TS_2 simulates TS_1), denoted $TS_1 \preceq TS_2$, if there exists a simulation $\mathcal{R}$ for (TS_1, TS_2). ■

Condition (A) requires that all initial states in TS_1 are related to an initial state of TS_2. As the reverse is not required, there might be initial states of TS_2 that are not matched by an initial state of TS_1. Conditions (B.1) and (B.2) are as for bisimulations; note that the symmetric counterpart of (B.2)—as for bisimulation—is not required.

Example 7.48. Two Beverage Vending Machines

Consider the transition systems in Figure 7.17. Let $AP = \{ pay, beer, soda \}$ with the obvious labeling functions. The relation

$$\mathcal{R} \;=\; \{ (s_0, t_0),\ (s_1, t_1),\ (s_2, t_1),\ (s_3, t_2),\ (s_4, t_3) \}$$

is a simulation for (TS_1, TS_2). Since $\mathcal{R}$ contains the pair of initial states (s_0, t_0), $TS_1 \preceq TS_2$. The reverse does not hold, i.e., $TS_2 \not\preceq TS_1$, since there is no state in TS_1 that can mimic state t_1. This is due to the fact that the options "*beer*" and "*soda*" are possible in state t_1, but in no state in TS_1.

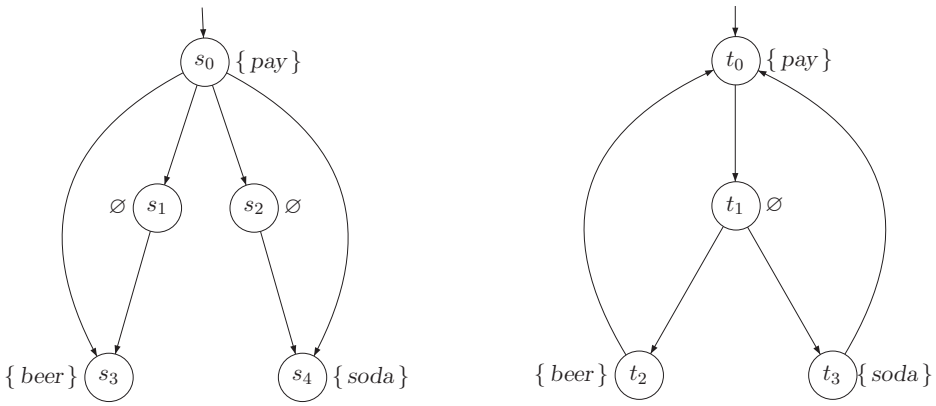


Figure 7.17: The vending machine on the right simulates the one on the left.

Consider now $AP = \{ pay, drink \}$, and assume states s_3, s_4, t_2 and t_3 equals $\{ drink \}$. Relation $\mathcal{R}$ is again a simulation for (TS_1, TS_2), and thus $TS_1 \preceq TS_2$. Its inverse,

$$\mathcal{R}^{-1} \;=\; \{ (t_0, s_0),\ (t_1, s_1),\ (t_1, s_2),\ (t_2, s_3),\ (t_3, s_4) \},$$

is a simulation for (TS_2, TS_1). We thus also obtain $TS_2 \preceq TS_1$. ■

The simulation relation $\preceq$ is a preorder, i.e., it is reflexive and transitive. As the conditions of simulation relations are not symmetric, the simulation preorder is not an equivalence.

Lemma 7.49. Reflexivity and Transitivity of $\preceq$

For a fixed set AP of atomic propositions, the relation $\preceq$ is reflexive and transitive.

Proof: Similar to the proof of reflexivity and transitivity of $\sim$; see Lemma 7.4 (page 453). ■

$TS_1 \preceq TS_2$ holds if TS_1 originates from TS_2 by deleting transitions from TS_2, e.g., in case of a nondeterministic choice in TS_1, only one alternative is retained. In such cases, TS_1 can be understood as a *refinement* of TS_2, since TS_1 resolves some nondeterminism in TS_2. $TS_1 \preceq TS_2$ also holds when TS_2 arises from TS_1 by means of an *abstraction*. In fact, abstraction is a fundamental concept that permits the analysis of large or even infinite transition systems. Abstraction is identified by a set of abstract states $\widehat{S}$; an abstraction function f, which associates to each (concrete) state s of the transition system TS the abstract state $f(s)$ which represents it; and a set AP of atomic propositions which label the concrete and abstract states. Abstractions differ in the choice of the set $\widehat{S}$ of abstract states, the abstraction function f, and the relevant propositions AP. The concept of abstraction functions has, in fact, already been used in several examples in this monograph. For instance, an abstraction function was used for the Bakery algorithm in order to prove that the infinite transition system TS_{Bak} has a finite bisimulation quotient, see Example 7.13 (page 461). Due to the special structure of the transition system, here a bisimulation-equivalent transition system results. Typically, however, an abstract transition system results that simulates TS.

We briefly outline the essential ideas of abstractions that are obtained by aggregating disjoint sets of concrete states into single abstract states. Abstraction functions map concrete states onto abstract ones, such that abstract states are associated with equally labeled concrete states only.

Definition 7.50. Abstraction Function

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system, and $\widehat{S}$ be a set of (abstract) states. $f : S \rightarrow \widehat{S}$ is an *abstraction function* if for any $s, s' \in S$: $f(s) = f(s')$ implies $L(s) = L(s')$. ■

The abstract transition system TS_f originates from TS by identifying all states that are represented by the same abstract state under abstraction function f. An abstract state is initial whenever it represents an initial concrete state. Similarly, there is a transition from abstract state $f(s)$ to state $f(s')$ if there is a transition from s to s'.

Definition 7.51. Abstract Transition System

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a (concrete) transition system, $\widehat{S}$ a set of (abstract) states, and $f : S \to \widehat{S}$ an abstraction function. The *abstract transition system* $TS_f = (\widehat{S}, Act, \rightarrow_f, I_f, AP, L_f)$ induced by f on TS is defined by:

- $\rightarrow_f$ is defined by the rule: $\dfrac{s \xrightarrow{\alpha} s'}{f(s) \xrightarrow{\alpha}_f f(s')}$
- $I_f = \{ f(s) \mid s \in I \}$
- $L_f(f(s)) = L(s)$ for all states $s \in S$

■

Lemma 7.52.

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a (concrete) transition system, $\widehat{S}$ a set of (abstract) states, and $f : S \to \widehat{S}$ an abstraction function. Then $TS \preceq TS_f$.

Proof: Follows from the fact that $\mathcal{R} = \{ (s, f(s)) \mid s \in S \}$ is a simulation for (TS, TS_f). ■

Example 7.53. An Automatic Door Opener

Consider an automatic door opener as modeled by the (concrete) transition system in Figure 7.18; for simplicity, the action labels are omitted from the transitions. Let $AP = \{ \mathit{alarm}, \mathit{open} \}$. The door opener requires a three-digit code $d_1\, d_2\, d_3$ as input with $d_i \in \{ 0, \ldots, 9 \}$. It allows an erroneous digit to be entered, but this may happen at most twice. The variable *error* keeps track of the number of wrong digits that have been provided, and is initially zero. In case *error* exceeds two, the door opener issues an alarm signal. On a successful input of the door code, the door is opened. Once locked again, it returns to its initial state. Location ℓ_i (for $i = 0, 1, 2$) indicates that the first i digits of the code have been correctly entered; the second component of a state indicates the value of the variable *error* (if applicable).

Consider the following two abstractions. In the first abstraction, the concrete states are aggregated as indicated by the dashed ovals in Figure 7.18. In fact, this is a data abstraction in which the domain of the variable *error* is restricted to $\{ \leqslant 1, 2 \}$, i.e., the values 0 and 1 are not distinguished in the abstract transition system. The corresponding abstraction

function f is defined by

$$f(\langle \ell, error = k \rangle) = \begin{cases} \langle \ell, error \leqslant 1 \rangle & \text{if } k \in \{0, 1\} \\ \langle \ell, error = 2 \rangle & \text{if } k = 2 \end{cases}$$

For all other concrete states, f is the identity function. It immediately follows that f is indeed an abstraction function, as only equally labeled states are mapped onto the same abstract state. The abstract transition system TS_f is depicted in Figure 7.19; by construction we have $TS \preceq TS_f$.

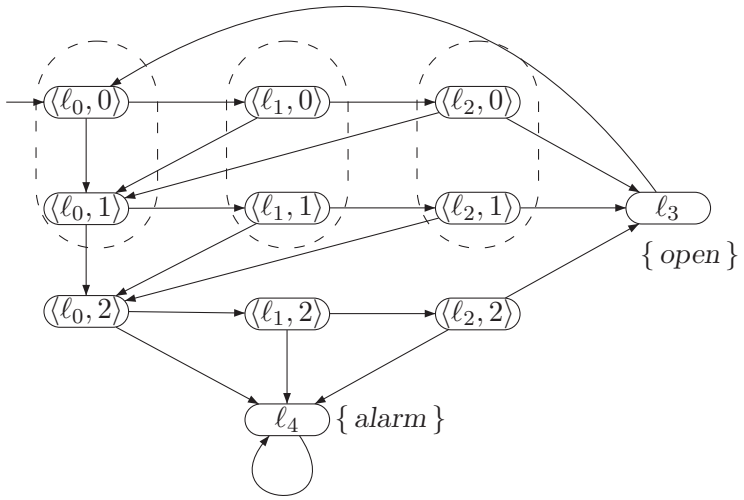


Figure 7.18: Transition system of the door opener

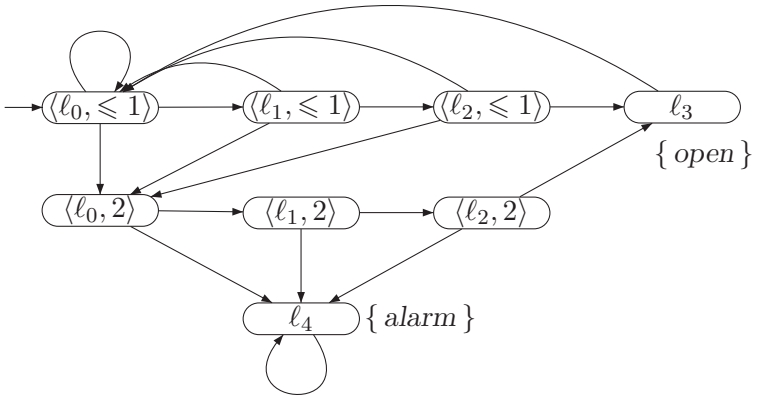


Figure 7.19: Abstract transition system of the door opener (under abstraction function f)

Now consider an abstraction in which we completely abstract from the value of the variable *error*. The concrete states that are aggregated are indicated by the dashed ovals in Figure

7.20. The corresponding abstraction function g is defined by $g(\langle \ell, error = k \rangle) = \ell$, for any location $\ell \in \{ \ell_0, \ell_1, \ell_2 \}$; g is the identity function otherwise. This yields the transition system TS_g in Figure 7.21. As g is indeed an abstraction function, it follows that $TS \preceq TS_g$. ■

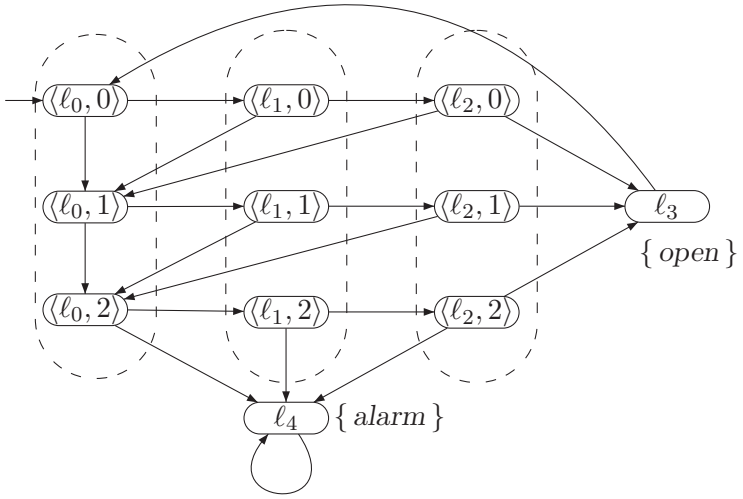


Figure 7.20: Alternative aggregation of states for the door opener.

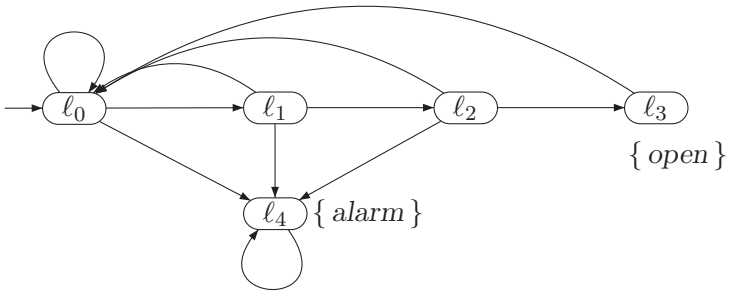


Figure 7.21: Abstract transition system of the door opener (under abstraction function g).

The abstract transition systems in the previous example have been obtained by choosing for the variable *error* in the system an abstract domain that is smaller than its original domain $\{ 0, 1, 2 \}$. Let us illustrate what such a type of abstraction (also known as *data abstraction*) means for program graphs. The essential idea of abstraction in program graphs is to abstract from concrete values of certain program variables or locations, i.e., program counters. Rather than keeping track of the concrete and precise value of variables, only abstract values are considered. For instance, a possible abstraction of an integer x could be a three-valued variable which is -1 whenever $x < 0$, 0 whenever $x{=}0$, and 1 if $x > 1$. Such abstraction is useful when only the sign of x is of importance.

Example 7.54. Data Abstraction applied to a Simple Sequential Program

Consider the following program fragment on the nonnegative integer variables x and y:

ℓ_0 **while** $x > 0$ **do**
ℓ_1 $\quad x := x - 1;$
ℓ_2 $\quad y := y + 1;$
od;
ℓ_3 **if** $even(y)$ **then** return "1" **else** return "0" **fi**;
ℓ_4 $\vdots$

Let PG be the program graph of this program, and TS its underlying transition system $TS(PG)$. Each state of TS is of the form $s = \langle \ell, x = n, y = m \rangle$, where ℓ is a program location and m and n are natural numbers. By means of abstraction one of the variables may be completely ignored or the variable domains are restricted. We exemplify the second kind of abstraction and restrict x and y to the domains

$$dom_{abstract}(x) = \{gzero, zero\} \quad \text{and} \quad dom_{abstract}(y) = \{even, odd\}.$$

Stated in words, we only keep track of whether $x > 0$ or x=0, and whether y is even or odd. The precise values of x and y are not explicitly administered.

The abstraction function f which maps a concrete state $\langle \ell, x = n, y = m \rangle$ (with $m, n \in \mathbb{N}$) onto an abstract state $\langle \ell, x = V, y = W \rangle$ (with $V \in \{gzero, zero\}$ and $W \in \{even, odd\}$) is defined as follows:

$$f(\langle \ell, x = v, y = w \rangle) \;=\; \begin{cases} \langle \ell, x = gzero, y = even \rangle & \text{if } x > 0 \wedge y \text{ is even} \\ \langle \ell, x = gzero, y = odd \rangle & \text{if } x > 0 \wedge y \text{ is odd} \\ \langle \ell, x = zero,\ y = even \rangle & \text{if } x = 0 \wedge y \text{ is even} \\ \langle \ell, x = zero,\ y = odd \rangle & \text{if } x = 0 \wedge y \text{ is odd} \end{cases}$$

To obtain an abstract transition system TS_f with $TS \preceq TS_f$, the operations in TS (e.g., incrementing y) have to be replaced with corresponding abstract operations that yield values from the abstract domains. Accordingly, the statement $y := y + 1$ is replaced with the abstract operation:

$$y \mapsto \begin{cases} even & \text{if } y = odd \\ odd & \text{if } y = even \end{cases}$$

The outcome of the statement $x := x - 1$ depends on the value of x. In the abstract setting, the precise value of x is, however, not known. Thus, $x := x - 1$ corresponds to the abstract *nondeterministic* statement:

$$x := gzero \quad \textbf{or} \quad x := zero$$

(If the guard of the while loop would be $x > 1$, then there would be no reason for nondeterminism in the abstract program.)

The abstract transition system TS_f results from the following piece of program:

ℓ_0 **while** $(x = \mathit{gzero})$ **do**
ℓ_1 $\quad x := \mathit{gzero}$ **or** $x := \mathit{zero}$;
ℓ_2 $\quad$ **if** $y = \mathit{even}$ **then** $y := \mathit{odd}$ **else** $y := \mathit{even}$ **fi**;
od;
ℓ_3 **if** $y = \mathit{even}$ **then** return "1" **else** return "0" **fi**;
ℓ_4 $\vdots$

Note that the abstract program originates from syntactic transformations which can be completely automated. The previous considerations show that

$$\mathcal{R} \;=\; \{\, (s, f(s)) \mid s \in \mathit{Loc} \times \mathit{Eval}(x \in \mathbb{N}, y \in \mathbb{N}) \,\}$$

is a simulation for (TS, TS_f), provided that the initial values of the abstract variables are chosen in accordance with the initial values of the concrete variables, and the set of atomic propositions is (a subset of)

$$AP \;=\; \{\, \ell_0, \ell_1, \ell_2, \ell_3, \ell_4, x > 0, x = 0, \mathit{even}(y), \mathit{odd}(y) \,\}$$

used with the obvious labeling function. ■

If $\mathcal{R}$ is a simulation and $(s_1, s_2) \in \mathcal{R}$ then each path fragment π_1 of s_1 can be lifted to a path fragment π_2 of s_2 such that π_1 and π_2 are statewise related via $\mathcal{R}$, see Figure 7.22.

Lemma 7.55. Simulation on Path Fragments

Let TS_1 and TS_2 be transition systems over AP, $\mathcal{R}$ a simulation for (TS_1, TS_2), and $(s_1, s_2) \in \mathcal{R}$ Then for each (finite or infinite) path fragment $\pi_1 = s_{0,1}\, s_{1,1}\, s_{2,1} \ldots$ starting in $s_{0,1} = s_1$ there exists a path fragment $\pi_2 = s_{0,2}\, s_{1,2}\, s_{2,2} \ldots$ from state $s_{0,2} = s_2$ of the same length such that $(s_{j,1}, s_{j,2}) \in \mathcal{R}$ for all j.

Proof: The proof is similar to Lemma 7.5 (page 454). ■

Hence, whenever $\mathcal{R}$ is a simulation that contains (s_1, s_2) and π_1 an infinite path from state s_1 then there exists an infinite path from s_2 such that π_1 and π_2 are statewise $\mathcal{R}$-related. The labeling condition ensures that the traces of π_1 and π_2 agree. This yields that all infinite traces of s_1 are at the same time infinite traces of s_2. If, however, π_1 is a finite

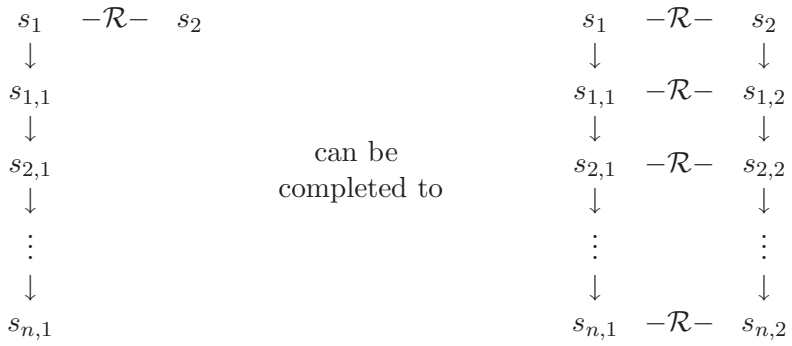


Figure 7.22: Path fragment lifting for simulations.

path from s_1 then π_2 is a finite path fragment from s_2, but possibly not maximal (i.e., the last state of π_2 might be nonterminal). Hence, state s_1 can have finite traces, which are trace fragments, but not traces of s_2.

7.4.1 Simulation Equivalence

The simulation relation $\preceq$ is transitive and reflexive, but not symmetric. An example where $TS_1 \preceq TS_2$, but $TS_2 \not\preceq TS_1$, was provided in Example 7.48 (page 498). However, as any preorder, $\preceq$ induces an equivalence relation, the so-called *kernel* of $\preceq$, which is defined by $\simeq \;=\; \preceq \;\cap\; \preceq^{-1}$. It consists of all pairs (TS_1, TS_2) of transition systems that can mutually simulate each other. The relation $\simeq$ is called simulation equivalence.

Definition 7.56. Simulation Equivalence

Transition systems TS_1 and TS_2 (over AP) are *simulation-equivalent*, denoted $TS_1 \simeq TS_2$, if $TS_1 \preceq TS_2$ and $TS_2 \preceq TS_1$. ■

Example 7.57. Simulation Equivalent Transition Systems

The transition systems TS_1 (left) and TS_2 (right) in Figure 7.23 are simulation-equivalent. This can be seen as follows. Since TS_1 is a subgraph of TS_2 (up to isomorphism, i.e., state identities), it is clear that $TS_1 \preceq TS_2$. Consider now the reverse direction. The transition $t_1 \to t_2$ in TS_2 is simulated by $s_1 \to s_2$. Formally,

$$\mathcal{R} = \{\, (t_1, s_1), (t_2, s_2), (t_3, s_2), (t_4, s_3) \,\}$$

is a simulation for (TS_2, TS_1). From this, we derive $TS_2 \preceq TS_1$ and obtain $TS_1 \simeq TS_2$.

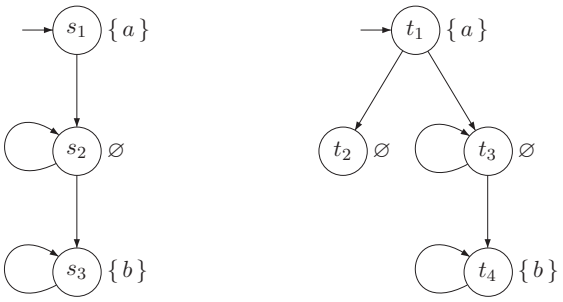


Figure 7.23: Simulation-equivalent transition systems.

The beverage vending machines described in Example 7.48 (page 498) are *not* simulation-equivalent if (as before) the set of propositions $AP = \{ pay, beer, soda \}$ is used. However, $TS_1 \simeq TS_2$ holds for $AP = \{ pay \}$ or for $AP = \{ pay, drink \}$. ■

The simulation preorder $\preceq$ and its induced equivalence may also be adopted to compare states of a single transition system.

Definition 7.58. Simulation Order as a Relation on States

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system. A *simulation* for TS is a binary relation $\mathcal{R} \subseteq S \times S$ such that for all $(s_1, s_2) \in \mathcal{R}$:

1. $L(s_1) = L(s_2)$.
2. If $s_1' \in Post(s_1)$, then there exists an $s_2' \in Post(s_2)$ with $(s_1', s_2') \in \mathcal{R}$.

State s_1 is simulated by s_2 (or s_2 simulates s_1), denoted $s_1 \preceq_{TS} s_2$, if there exists a simulation $\mathcal{R}$ for TS with $(s_1, s_2) \in \mathcal{R}$. States s_1 and s_2 of TS are *simulation-equivalent*, denoted $s_1 \simeq_{TS} s_2$, if $s_1 \preceq_{TS} s_2$ and $s_2 \preceq_{TS} s_1$. ■

For state s, let $Sim_{TS}(s)$ denote the *simulator set* of s, i.e., the set of states that simulate s. Formally:

$$Sim_{TS}(s) \;=\; \{\, s' \in S \mid s \preceq_{TS} s' \,\} \;.$$

For convenience, the branching-time relations introduced so far are summarized in Figure 7.24.

As stated on page 457 for bisimulations, the relation $\preceq_{TS}$ on $S \times S$ results from $\preceq$ via

$$s_1 \preceq_{TS} s_2 \quad \text{if and only if} \quad TS_{s_1} \preceq TS_{s_2}$$

simulation order
$s_1 \preceq_{TS} s_2 \quad :\Leftrightarrow \quad$ there exists a simulation $\mathcal{R}$ for TS with $(s_1, s_2) \in \mathcal{R}$

simulation equivalence
$s_1 \simeq_{TS} s_2 \quad :\Leftrightarrow \quad s_1 \preceq_{TS} s_2$ and $s_2 \preceq_{TS} s_1$

bisimulation equivalence
$s_1 \sim_{TS} s_2 \quad :\Leftrightarrow \quad$ there exists a bisimulation $\mathcal{R}$ for TS with $(s_1, s_2) \in \mathcal{R}$

Figure 7.24: Summary of the relations $\preceq_{TS}$, $\sim_{TS}$, and $\simeq_{TS}$.

where TS_s arises from TS by declaring s as the unique initial state. Vice versa, $TS_1 \preceq TS_2$ if each initial state s_1 of TS_1 is simulated by an initial state of TS_2 in the transition system $TS_1 \oplus TS_2$. Analogous observations hold for the simulation equivalence. These observations allow us to say that a state s_1 in a transition system TS_1 is simulated by a state s_2 of another transition system TS_2, provided that the pair (s_1, s_2) is contained in some simulation for (TS_1, TS_2).

In analogy to Lemma 7.8, $\preceq_{TS}$ is the coarsest simulation. Moreover, the simulation order on TS is a preorder (i.e., transitive and reflexive) on TS's state space and is the union of all simulations on TS.

Lemma 7.59. $\preceq_{TS}$ ***is a Preorder and the Coarsest Simulation***
For transition system $TS = (S, Act, \rightarrow, I, AP, L)$ *it holds that:*

1. $\preceq_{TS}$ *is a preorder on* S.
2. $\preceq_{TS}$ *is a simulation on* TS.
3. $\preceq_{TS}$ *is the coarsest simulation for* TS.

Proof: The first claim follows from Lemma 7.49 (page 453). To prove the second statement we show $\preceq_{TS}$ fulfills conditions (1) and (2) of simulations on TS (Definition 7.58). Let $s_1 \preceq_{TS} s_2$ for s_1, s_2 states in TS. Then, there exists a simulation $\mathcal{R}$ that contains (s_1, s_2). Since conditions (1) and (2) of Definition 7.58 hold for all pairs in $\mathcal{R}$, $L(s_1) = L(s_2)$ and for any transition $s_1 \rightarrow s_1'$, there exists a transition $s_2 \rightarrow s_2'$ with $(s_1', s_2') \in \mathcal{R}$. Hence, $s_1' \preceq_{TS} s_2'$. Thus, $\preceq_{TS}$ is a simulation on TS.

The third claim follows immediately from the second statement ($\preceq_{TS}$ is a simulation) and the fact that $s_1 \preceq_{TS} s_2$ if there exists some simulation containing (s_1, s_2). Hence, each simulation $\mathcal{R}$ is contained in $\preceq_{TS}$. ■

Let us now define the quotient transition system under simulation equivalence for transition system $TS = (S, Act, \rightarrow, I, AP, L)$. For simplicity, we omit the subscript TS and simply write $\preceq$ rather than $\preceq_{TS}$ and $\simeq$ rather than $\simeq_{TS}$.

Definition 7.60. Simulation Quotient System

For transition system $TS = (S, Act, \rightarrow, I, AP, L)$, the *simulation quotient* transition system $TS/\simeq$ is defined as follows:

$$TS/\simeq \;=\; (S/\simeq, \{\,\tau\,\}, \rightarrow_{\simeq}, I_{\simeq}, AP, L_{\simeq})$$

where $I_{\simeq} \;=\; \{\,[s]_{\simeq} \mid s \in I\,\}$, $L_{\simeq}([s]_{\simeq}) = L(s)$, and

$$\rightarrow_{\simeq} \text{ is defined by: } \quad \frac{s \xrightarrow{\alpha} s'}{[s]_{\simeq} \xrightarrow{\tau}_{\simeq} [s']_{\simeq}}$$

■

Note that the transition relation in Definition 7.60 is similar to that of the bisimulation quotient of TS, but considers equivalence classes under $\simeq$ rather than $\sim$. From the definition of $\rightarrow_{\simeq}$ it follows that

$$B \rightarrow_{\simeq} C \quad \text{if and only if} \quad \exists s \in B.\ \exists s' \in C.\ s \rightarrow s'.$$

In contrast to the bisimulation quotient, this is *not* equivalent to

$$B \rightarrow_{\simeq} C \quad \text{if and only if} \quad \forall s \in B.\ \exists s' \in C.\ s \rightarrow s'.$$

Stated in words, $B \rightarrow_{\simeq} C$ does not imply that $Post(s) \cap C \neq \varnothing$ for any $s \in B$. Thus, in general, we have $TS \not\sim TS/\simeq$. However, simulations can be established for TS and $TS/\simeq$ in both directions. That is to say, TS and $TS/\simeq$ are simulation equivalent:

***Theorem 7.61. Simulation equivalence of** TS **and** $TS/\simeq$*

For any transition system TS it holds that $TS \;\simeq\; TS/\simeq$.

Proof: $TS \preceq TS/\simeq$ follows directly from the fact that the relation $\mathcal{R} = \{\,(s, [s]_{\simeq}) \mid s \in S\,\}$ is a simulation for $(TS, TS/\simeq)$. We show that $TS/\simeq \;\preceq\; TS$ by establishing a simulation

$\mathcal{R}'$ for $(TS/\simeq, TS)$. First, observe that $\mathcal{R}' = \mathcal{R}^{-1} = \{\, ([s]_\simeq, s) \mid s \in S \,\}$ is not adequate, as it is possible that $s' \simeq s$, $s' \to t'$, but $[t']_\simeq \cap \mathit{Post}(s) = \varnothing$. Instead,

$$\mathcal{R}' = \{\, ([s]_\simeq, t) \mid s \preceq t \,\}$$

is a simulation for $(TS/\simeq, TS)$. This is proven as follows. Each initial state $[s_0]_\simeq$ in $TS/\simeq$ is simulated by an initial state in TS, since $[s_0]_\simeq \preceq s_0$ and $s_0 \in I$. In addition, $[s]_\simeq$ and t are equally labeled as $s \preceq t$ and all states in the simulator set $\mathit{Sim}_{TS}(s) = \{\, s' \in S \mid s \preceq_{TS} s' \,\}$ of s are equally labeled. It remains to check whether each transition of $[s]_\simeq$ is mimicked by a transition of t for $s \preceq t$. Let $B \in S/\simeq$, $(B, t) \in \mathcal{R}'$, and $B \to_\simeq C$ be a transition in $TS/\simeq$. By definition of $\to_\simeq$, there exists a state $s \in B$ and a transition $s \to s'$ in TS such that $C = [s']_\simeq$. As $(B, t) \in \mathcal{R}'$, t simulates some state $u \in B$, i.e., $u \preceq t$ for some $u \in B$. Since $s, u \in B$, and all states in B are simulation-equivalent, we get $u \preceq t$ and $s \preceq u$ (and $u \preceq s$). By transitivity of $\preceq$, we have $s \preceq t$. There is thus a transition $t \to t'$ in TS with $s' \preceq t'$. By definition of $\mathcal{R}'$, $(C, t') = ([s']_\simeq, t') \in \mathcal{R}'$. ∎

Remark 7.62. Alternative Definition of the Simulation Quotient System

Let us consider the following alternative definition of the simulation quotient system. Let TS be a transition system as before and

$$TS/\simeq' \;=\; (S/\simeq, \{\tau\}, \to'_\simeq, I_\simeq, AP, L_\simeq)$$

where $I_\simeq$ and $L_\simeq$ are as in Definition 7.60 and where the transitions in $TS/\simeq'$ arise by the rule

$$\frac{B, B' \in S/\simeq \;\;\wedge\;\; \forall s \in B\; \exists s' \in B'.\; s \to s'}{[s]_\simeq \to'_\simeq [s']_\simeq}$$

(where the action labels are omitted since they are not of importance here). Clearly, $TS/\simeq'$ is simulated by TS as $\mathcal{R} = \{([s]_\simeq, s) \mid s \in S\}$ is a simulation for $(TS/\simeq', TS)$. However, we cannot guarantee that the reverse holds, i.e., that TS is simulated by $TS/\simeq'$.

Let us illustrate this by means of an example. Assume that TS has two initial states s_1 and s_2 with $\mathit{Post}(s_1) = \{2i + 1 \mid i \geqslant 0\}$ and $\mathit{Post}(s_2) = \{2i \mid i \geqslant 0\}$. Moreover, assume that

$$1 \preceq 2 \preceq 3 \preceq \ldots$$

while state $n{+}1 \not\preceq n$ for all $n \in \mathbb{N}$. For instance, the state space of TS could be $\{s_1, s_2\} \cup \mathbb{N} \cup \{t_n \mid n \in \mathbb{N}\}$ with $s_j \to 2i{+}j$ for all $i \geqslant 0$, $j = 1, 2$, $n \to t_n$ and $t_{n+1} \to t_n$ for all $n \geqslant 0$, while state t_0 is terminal. Moreover, we deal with $AP = \{\, a, b \,\}$ and $L(s_1) = L(s_2) = \{\, a \,\}$, $L(n) = \{\, b \,\}$ and $L(t_n) = \varnothing$ for all $n \geqslant 0$. Then $t_i \preceq_{TS} t_j$ if and only if $i < j$ which yields

$$i \preceq_{TS} j \text{ if and only if } i < j$$

and $s_1 \simeq_{TS} s_2$. In the quotient system $TS/\simeq'$, states s_1 and s_2 are collapsed into their simulation equivalence class $B = \{ s_1, s_2 \}$. Since the states in $\mathbb{N}$ are pairwise not simulation-equivalent and since s_1 and s_2 do not have any common direct successor, B is a terminal state in $TS/\simeq'$. In particular, the reachable fragment of $TS/\simeq'$ just consists of the initial state B and does not have any transition. Thus, $TS/\simeq'$ and TS are not simulation equivalent.

In the above example, TS is infinite. If, however, we are given a finite transition system, then $TS/\simeq'$ simulates TS, which yields the simulation equivalence of TS and $TS/\simeq'$. To see why it suffices to establish a simulation for $(TS, TS/\simeq')$. Let

$$\mathcal{R}' = \{(s, [t]_\simeq) \mid s \preceq t\}$$

and show that $\mathcal{R}'$ is a simulation for $(TS, TS/\simeq')$. Conditions (A) and (B.1) are obvious. Let us check condition (B.2). We use the following claim:

Claim. If $B \in S/\simeq$, $t \in B$ and $t \longrightarrow t'$, then there exists a transition $t \longrightarrow t'_{max}$ such that $t' \preceq t'_{max}$ and $B \longrightarrow'_\simeq [t'_{max}]_\simeq$.

Proof of the claim. Let t'_{max} be such that t'_{max} is "maximal" in the sense that for any transition $t \longrightarrow v$ where $t'_{max} \preceq v$ we have $t'_{max} \simeq v$. (Such "maximal" elements exist since TS is finite.) For each state $u \in B$, we have $t \simeq u$. Since $t \preceq u$, there exists a transition $u \longrightarrow u'$ with $t'_{max} \preceq u'$. And vice versa, since $u \preceq t$, there exists a transition $t \longrightarrow v$ with $u' \preceq v$. But then $t'_{max} \preceq v$ by the transitivity of $\preceq$. The maximality of t'_{max} yields $t'_{max} \simeq v$. Since $t'_{max} \preceq u' \preceq v$ we get $t'_{max} \simeq u'$. We conclude that all states $u \in B$ have an outgoing transition leading to a state in $[t'_{max}]_\simeq$. This yields $B \longrightarrow'_\simeq [t'_{max}]_\simeq$.

We now use the above claim to show (B.2) for $\mathcal{R}'$ when TS is finite. Let us now suppose that $(s, B) \in \mathcal{R}'$ and that $s \longrightarrow s'$ is a transition in TS. We take an arbitrary representative $t \in B$. By definition of $\mathcal{R}'$, state t simulates s. Thus, there exists a transition $t \longrightarrow t'$ with $s' \preceq t'$. The above claim yields the existence of a transition $t \longrightarrow t'_{max}$ with $t' \preceq t'_{max}$ and $B \longrightarrow'_\simeq C$ where $C = [t'_{max}]_\simeq$. But then $s' \preceq t'_{max}$ and hence, $(s', C) \in \mathcal{R}'$. ∎

7.4.2 Bisimulation, Simulation, and Trace Equivalence

In this monograph, various equivalences and preorder relations have been defined on transition systems. This section compares bisimulation, simulation, and (infinite and finite) trace equivalence as well as the simulation preorder and trace inclusion. In the sequel of this section, let $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP, L_i)$, $i = 1, 2$, be transition systems over AP. We first observe that bisimulation equivalence implies simulation equivalence. This simply

follows by the symmetry of the conditions required for bisimulations. However, there are simulation equivalent transition systems which are not bisimulation equivalent. Note that bisimulation equivalence requires that whenever $s_1 \sim s_2$, then every transition $s_1 \to s_1'$ can be mimicked by a transition $s_2 \to s_2'$ such that $s_1' \sim s_2'$. Simulation equivalence, however, requires that whenever $s_1 \simeq s_2$, then $s_1 \to s_1'$ can be mimicked by $s_2 \to s_2'$ such that $s_1' \preceq s_2'$ (but not necessarily $s_1' \simeq s_2'$!). The fact that $TS_1 \simeq TS_2$ does not always imply $TS_1 \sim TS_2$ is illustrated by the following example.

Example 7.63. Similar but not Bisimilar Transition Systems

Consider the transition systems TS_1 (left) and TS_2 (right) in Figure 7.25. $TS_1 \not\sim TS_2$, as there is no bisimilar state in TS_2 that mimics state s_2; the only candidate would be t_2, but s_2 cannot mimic $t_2 \to t_4$. TS_1 and TS_2 are, however, simulation-equivalent. As TS_2 is a subgraph (up to isomorphism) of TS_1, we obtain $TS_2 \preceq TS_1$. In addition, $TS_1 \preceq TS_2$ as

$$\mathcal{R} = \{ (s_1, t_1), (s_2, t_2), (s_3, t_2), (s_4, t_3), (s_5, t_4) \}$$

is a simulation relation for (TS_1, TS_2). ■

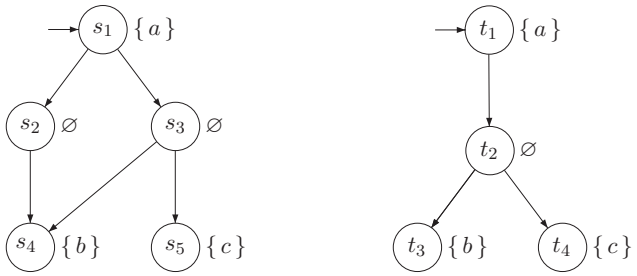


Figure 7.25: Simulation-, but not bisimulation-equivalent transition systems.

Theorem 7.64. Bisimulation is Strictly Finer than Simulation Equivalence

$TS_1 \sim TS_2$ implies $TS_1 \simeq TS_2$, but $TS_1 \not\sim TS_2$ and $TS_1 \simeq TS_2$ is possible.

Proof: Suppose $TS_1 \sim TS_2$. Then, there exists a bisimulation $\mathcal{R}$ for (TS_1, TS_2). It follows directly that $\mathcal{R}$ is a simulation for (TS_1, TS_2), and that $\mathcal{R}^{-1}$ is a simulation for (TS_2, TS_1). Hence, $TS_1 \simeq TS_2$. Example 7.63 provides transition systems TS_1 and TS_2 such that $TS_1 \simeq TS_2$, but $TS_1 \not\sim TS_2$. ■

Recall the notion of AP-determinism (see Definition 2.5 on page 24).

Definition 7.65. *AP*-**Deterministic Transition System**

Transition system $TS = (S, Act, \rightarrow, I, AP, L)$ is *AP-deterministic* if

1. for $A \subseteq AP$: $|I \cap \{ s \mid L(s) = A \}| \leqslant 1$, and
2. for $s \in S$: if $s \xrightarrow{\alpha} s'$ and $s \xrightarrow{\alpha} s''$ and $L(s') = L(s'')$, then $s' = s''$.

■

The difference between $\sim$ and $\simeq$ crucially relies on AP-determinism, e.g., transition system TS_1 in Figure 7.25 (left) is not AP-deterministic as its initial state has two distinct $\varnothing$-successors.

Theorem 7.66. *AP-Determinism Implies $\sim$ and $\simeq$ Coincide*

If TS_1 and TS_2 are AP-deterministic, then $TS_1 \sim TS_2$ if and only if $TS_1 \simeq TS_2$.

Proof: see Exercise 7.3. ■

This completes the comparison between bisimulation and simulation equivalence. To enable the comparison with trace equivalence, we first consider the relation between $\preceq$ and (finite and infinite) trace inclusion.

Theorem 7.67. *Simulation Order vs. Finite Trace Inclusion*

$TS_1 \preceq TS_2$ implies $Traces_{fin}(TS_1) \subseteq Traces_{fin}(TS_2)$.

Proof: Assume $TS_1 \preceq TS_2$. Let $\widehat{\pi_1}$ be $s_1 = s_{0,1}\, s_{1,1}\, s_{2,1} \ldots s_{n,1} \in Paths_{fin}(s_1)$ where $s_1 \in I_1$, i.e., s_1 is an initial state in TS_1. Since $TS_1 \preceq TS_2$, there exists $s_2 \in I_2$ such that $s_1 \preceq s_2$. As $\preceq$ can be lifted to finite path fragments (see Lemma 7.55 on page 504), it follows that there exists $\widehat{\pi_2} \in Paths_{fin}(s_2)$, say, of the form $s_2 = s_{0,2}\, s_{1,2}\, s_{2,2} \ldots s_{2,n}$ such that $\widehat{\pi_1} \preceq \widehat{\pi_2}$, i.e., $s_{j,1} \preceq s_{j,2}$ for $0 \leqslant j \leqslant n$. But then $L(s_{j,1}) = L(s_{j,2})$ for $0 \leqslant j \leqslant n$, that is, $trace(\pi_1) = trace(\pi_2)$. ■

Corollary 7.68. *Simulation Preserves Safety Properties*

Let P_{safe} be a safety LT property and TS_1 and TS_2 transition systems (all over AP) without terminal states. Then:

$$TS_1 \preceq TS_2 \quad \textit{and} \quad TS_2 \models P_{safe} \quad \textit{implies} \quad TS_1 \models P_{safe}.$$

Proof: Follows directly from Theorem 7.67 and the fact that $\mathit{Traces}_{\mathit{fin}}(TS_2) \supseteq \mathit{Traces}_{\mathit{fin}}(TS_1)$ implies that if $TS_2 \models P_{\mathit{safe}}$, then $TS_1 \models P_{\mathit{safe}}$ (see Corollary 3.15 on page 104). ■

Theorem 7.67 relates simulation preorder and finite trace inclusion, and cannot be obtained for trace inclusion. That is, in general, $TS_1 \preceq TS_2$ does not imply that $\mathit{Traces}(TS_1) \subseteq \mathit{Traces}(TS_2)$. This is illustrated by the following example.

Example 7.69. Simulation Preorder Does Not Imply Trace Inclusion

Consider the transition systems TS_1 (left) and TS_2 (right) depicted in Figure 7.26. We have $TS_1 \preceq TS_2$, but $\mathit{Traces}(TS_1) \not\subseteq \mathit{Traces}(TS_2)$ since $\{a\}\varnothing \in \mathit{Traces}(TS_1)$ but $\{a\}\varnothing \notin \mathit{Traces}(TS_1)$. This is due to the fact that $s_2 \preceq t_2$, but whereas s_2 is a terminal state, t_2 is

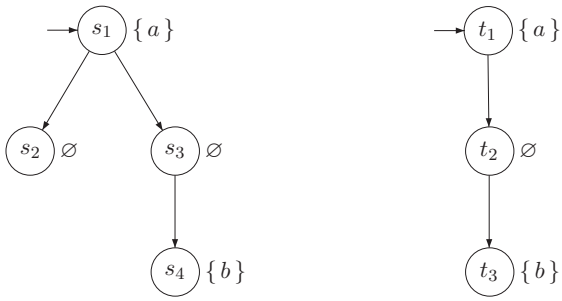


Figure 7.26: $TS_1 \preceq TS_2$, but $\mathit{Traces}(TS_1) \not\subseteq \mathit{Traces}(TS_2)$.

not. (Note that terminal states are simulated by any equally labeled state, be it terminal or not). As a result, a trace in TS_1 may end in s_2, but a trace in TS_2 cannot end in t_2. ■

The example indicates that terminal states are an essential factor of not preserving trace inclusion. Indeed, if terminal states are absent in TS_1 and $TS_1 \preceq TS_2$ then all paths in TS_1 are infinite and the path fragment lifting for simulations yields that each path of TS_1 can be lifted to an infinite path in TS_2. This yields:

Theorem 7.70. Simulation Order and Trace Inclusion

If $TS_1 \preceq TS_2$ and TS_1 does not have terminal states then $\mathit{Traces}(TS_1) \subseteq \mathit{Traces}(TS_2)$.

Thus, for transition systems without terminal states, the result stated in Corollary 7.68 holds for all LT properties and not just the safety properties.

For the class of AP-deterministic transition systems one obtains a similar result. Here, however, we have to deal with simulation equivalence and trace equivalence (rather than the simulation preorder and trace inclusion).

Theorem 7.71. Simulation in AP-Deterministic Systems

If $TS_1 \simeq TS_2$ and TS_1 and TS_2 are AP-deterministic then $Traces(TS_1) = Traces(TS_2)$.

Proof: See Exercise 7.3. ■

Corollary 7.72. Simulation and (Finite) Trace Equivalence

For transition systems TS_1 and TS_2 over AP:

(a) If $TS_1 \simeq TS_2$, then $Traces_{fin}(TS_1) = Traces_{fin}(TS_2)$.

(b) If TS_1 and TS_2 do not have terminal states and $TS_1 \simeq TS_2$, then $Traces(TS_1) = Traces(TS_2)$.

(c) If TS_1 and TS_2 are AP-deterministic, then $TS_1 \simeq TS_2$ if and only if $Traces(TS_1) = Traces(TS_2)$.

Proof: Claim (a) follows from Theorem 7.67, claim (b) from Theorem 7.70. Claim (c) follows from (and only if) Theorem 7.71 and (if) Exercise 7.3. ■

Simulation equivalent transition systems without terminal states satisfy the same LT properties, and hence, the same LTL formulae. Claim (c) together with Theorem 7.66 yield that for AP-deterministic transition systems, $\sim$, $\simeq$, and trace equivalence coincide. (Finite) Trace equivalence does not imply simulation equivalence. This is illustrated by the two beverage vending machines described in Example 7.48 on page 498.

Remark 7.73. Bisimulation and Trace Equivalence

Since $\sim$ is finer than $\simeq$, it follows from Theorem 7.67 (page 512) that $\sim$ is finer than finite trace equivalence:

$$TS_1 \sim TS_2 \text{ implies } Traces_{fin}(TS_1) \;=\; Traces_{fin}(TS_2).$$

However, bisimulation equivalent transition systems are even trace eqivalent. That is, $TS_1 \sim TS_2$ implies $Traces(TS_1) = Traces(TS_2)$. This applies to any transition system

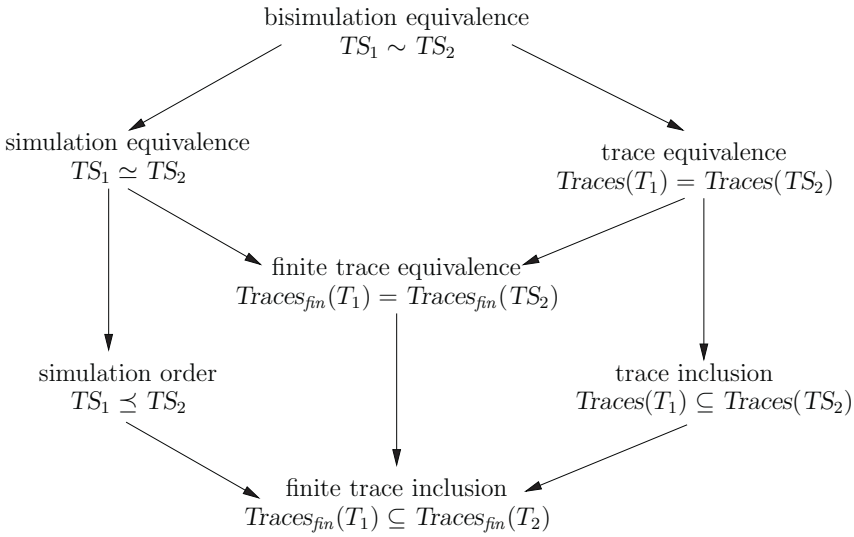


Figure 7.27: Relation between equivalences and preorders on transition systems.

TS_1, i.e., the absence of terminal states (as in Theorem 7.70 on page 513) is not required. This stems from the fact that a terminal state cannot be bisimilar to a nonterminal state, while terminal states are simulated by any state with the same label. ■

Figure 7.27 summarizes the relationship between bisimulation, simulation, and (finite and infinite) trace equivalence as well as simulation order and trace inclusion in the form of a Hasse diagram. The vertices of the diagram are relations, and an edge from $\mathcal{R}$ to $\mathcal{R}'$ means that $\mathcal{R}$ is strictly finer than $\mathcal{R}'$, that is to say, $\mathcal{R}$ is more distinctive than $\mathcal{R}'$. Recall that for AP-deterministic transition systems, $\sim$, $\simeq$ and trace equivalence coincide.

7.5 Simulation and ∀CTL* Equivalence

The aim of this section is to provide a logical characterization of the simulation order $\preceq$. This amounts establishing an analogous result to the theorem stating that bisimulation equivalence coincides with CTL* (and CTL) equivalence for finite transition systems without terminal states. That is to say, given two transition systems TS_1 and TS_2, the aim is to characterize the relation $\preceq$ in terms of the satisfaction relations on TS_1 and TS_2 for some temporal logic. In fact, from the results of the previous section, it can be inferred that for transition systems without terminal states, $TS_1 \preceq TS_2$ implies trace inclusion of TS_1 and TS_2, and thus whenever $TS_2 \models \varphi$ for a LTL formula φ, it follows $TS_1 \models \varphi$. In this section, it is shown that this not only applies to any LTL formula, but for a fragment

of CTL* that includes LTL.

Since $\preceq$ is not symmetric, it cannot be expected that $\preceq$ can be characterized by a sublogic of CTL* which allows negation of arbitrary state formulae. This can be seen as follows. Let $\mathbf{L}$ be a fragment of CTL* which is closed under negation, i.e., $\Phi \in \mathbf{L}$ implies $\neg\Phi \in \mathbf{L}$, such that for any transition system TS without terminal states and states s_1, s_2 of TS:

$$s_1 \preceq_{TS} s_2 \quad \text{iff} \quad \text{for all state formulae } \Phi \text{ of } \mathbf{L}\text{: } s_2 \models \Phi \implies s_1 \models \Phi.$$

Let $s_1 \preceq_{TS} s_2$. Note that $s_2 \models \neg\Phi$ implies $s_1 \models \neg\Phi$, i.e., $s_1 \not\models \neg\Phi$ implies $s_2 \not\models \neg\Phi$. Then, for any state formula Φ of $\mathbf{L}$:

$$s_1 \models \Phi \implies s_1 \not\models \neg\Phi \implies s_2 \not\models \neg\Phi \implies s_2 \models \Phi.$$

Hence, $s_2 \preceq_{TS} s_1$. Thus, a logic that characterizes $\preceq_{TS}$ cannot be closed under negation, as that would require $\preceq_{TS}$ to be symmetric which, however, is not the case.

The proof of the path-lifting lemma (Lemma 7.55 on page 504) informally explains why for every LTL formula φ such that $s_2 \models \varphi$ we have $s_1 \models \varphi$. This observation can be generalized for CTL* formulae of the form $\forall\varphi$, provided that the state formulae in φ contain neither negation nor existential quantification. These considerations motivate the definition of the *universal fragment* of CTL*, denoted $\forall$CTL*.

Definition 7.74. Universal Fragment of CTL*

The *universal fragment* of CTL*, denoted $\forall$CTL*, consists of the state formulae Φ and path formulae φ given, for $a \in AP$, by

$$\Phi \;::=\; \text{true} \mid \text{false} \mid a \mid \neg a \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid \forall\varphi$$

$$\varphi \;::=\; \Phi \mid \bigcirc\, \varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \,\mathsf{U}\, \varphi_2 \mid \varphi_1 \,\mathsf{R}\, \varphi_2.$$

■

State formulae in $\forall$CTL* are required to be in positive normal form, (negations may only occur adjacent to atomic propositions) and do not contain existential path quantifiers. Due to the positive normal form (PNF, for short), the release operator R is considered as a basic operator in the logic. The universal fragment of CTL, denoted $\forall$CTL, is obtained from the definition of $\forall$CTL* by restricting the path formulae to

$$\varphi \;::=\; \bigcirc\Phi \mid \Phi_1 \,\mathsf{U}\, \Phi_2 \mid \Phi_1 \,\mathsf{R}\, \Phi_2.$$

The eventually and always operators are obtained by the ∀CTL* path formulae $\Diamond\varphi = \text{true}\,\mathsf{U}\,\varphi$ and $\Box\varphi = \text{false}\,\mathsf{R}\,\varphi$. Similarly, in ∀CTL these modalities are obtained by $\forall\Diamond\Phi = \forall(\text{true}\,\mathsf{U}\,\Phi)$ and $\forall\Box\Phi = \forall(\text{false}\,\mathsf{R}\,\Phi)$.

∀CTL* covers all safety properties that can be expressed in CTL*. The following result shows that for any LTL formula there exists an equivalent ∀CTL* formula.

Lemma 7.75. ∀CTL* Includes LTL

For every LTL formula φ there exists an equivalent ∀CTL formula.*

Proof: It follows from Theorem 5.24 (page 257) that for any LTL formula φ there exists an equivalent LTL formula in PNF. From the definition of ∀CTL* and positive normal form LTL, it follows directly that any LTL formula in PNF can be regarded as a ∀CTL* formula. ■

Vice versa, the universal fragment of CTL* is more expressive than LTL, as, e.g., $\forall\Diamond\forall\Box a$ is a ∀CTL* formula that has no equivalent LTL formula. The following theorem indicates a temporal logic characterization of the simulation order by ∀CTL* and ∀CTL.

Theorem 7.76. Simulation Order and ∀CTL*/∀CTL

Let TS be a finite transition system without terminal states and s_1, s_2 be states of TS. The following statements are equivalent:

(a) $s_1 \preceq_{TS} s_2$.

(b) For all ∀CTL formulae* Φ: $s_2 \models \Phi$ *implies* $s_1 \models \Phi$.

(c) For all ∀CTL formulae Φ: $s_2 \models \Phi$ *implies* $s_1 \models \Phi$.

Proof: The proof technique is analogous to Theorem 7.20 (page 469).

(a) $\Longrightarrow$ (b) follows from the following claims that hold for arbitrary (i.e., not necessarily finite) transition system TS with state space S:

(i) if $s_1 \preceq_{TS} s_2$, then for all ∀CTL* state formulae Φ: $s_2 \models \Phi$ implies $s_1 \models \Phi$.

(ii) if $\pi_1 \preceq_{TS} \pi_2$, then for all ∀CTL* path formulae φ: $\pi_2 \models \varphi$ implies $\pi_1 \models \varphi$.

The proof of these claims is by structural induction, similar to Lemma 7.26 (page 473), and is omitted here. See also Exercise 7.14. For the treatment of formulas $\forall\varphi$ in the step induction for (i), the assumption that the given transition system does not have terminal states is crucial, since it permits path lifting rather than just path fragment lifting.

(b) $\Longrightarrow$ (c) is obvious since $\forall$CTL is a sublogic of $\forall$CTL*.

(c) $\Longrightarrow$ (a). Let S be the state space of the finite transition system TS. It suffices to show that

$$\mathcal{R} = \{\, (s_1, s_2) \in S \times S \mid \forall \Phi \in \forall\mathrm{CTL}^*.\, s_2 \models \Phi \;\Rightarrow\; s_1 \models \Phi \,\}$$

is a simulation for TS. This is proven by checking the conditions of a simulation relation. Let $(s_1, s_2) \in \mathcal{R}$.

1. If AP is finite, then we can argue by means of the formula

$$\Phi \;=\; \bigwedge_{a \in L(s_2)} a \;\wedge \bigwedge_{a \in AP \setminus L(s_2)} \neg a.$$

   Note that Φ is a propositional formula in PNF, and hence a $\forall$CTL formula. Since $s_2 \models \Phi$ and $(s_1, s_2) \in \mathcal{R}$, we obtain $s_1 \models \Phi$. As Φ uniquely characterizes the labeling of s_1, we have $L(s_1) = L(s_2)$.

   If AP is infinite we need a slightly different argument. Let $a \in AP$. If $a \in L(s_2)$, then $s_2 \models a$. Since the atomic proposition a is a $\forall$CTL formula, we get $s_1 \models a$ (by definition of $\mathcal{R}$), and therefore $a \in L(s_1)$. Similarly, if $a \notin L(s_2)$, then $s_2 \models \neg a$. Again, as $\neg a$ is a $\forall$CTL formula and by definition of $\mathcal{R}$, we get $s_1 \models \neg a$, and hence $a \notin L(s_1)$. This yields $L(s_1) = L(s_2)$.

2. The idea for establishing the condition (2) of simulation relations is to define a $\forall$CTL master formula Φ_u for the *downward closure* of state u with respect to $\mathcal{R}$, i.e.,

$$Sat(\Phi_u) \;=\; u{\downarrow} \;= \{\, t \in S \mid (t, u) \in \mathcal{R} \,\}.$$

   The definition of Φ_u is as follows. If u, t are states in TS with $t \notin u{\downarrow}$ then, according to the definition of $u{\downarrow}$ and $\mathcal{R}$, there exists a $\forall$CTL formula $\Phi_{u,t}$ such that $u \models \Phi_{u,t}$ and $t \not\models \Phi_{u,t}$. Let Φ_u be the conjunction of all formulae $\Phi_{u,t}$ where $t \notin u{\downarrow}$:

$$\Phi_u = \bigwedge_{\substack{t \in S \\ (t,u) \notin \mathcal{R}}} \Phi_{u,t}$$

   As TS is finite, Φ_u arises by a finite conjunction, and thus, it is a $\forall$CTL formula. In fact, Φ_u is a master formula for $u{\downarrow}$:

$$Sat(\Phi_u) = u{\downarrow}$$

Let us check this. If $v \notin u{\downarrow}$ then $(v,u) \notin \mathcal{R}$ and Φ_u has the form $\ldots \wedge \Phi_{u,v} \wedge \ldots$. Hence, $v \not\models \Phi_u$ (as $v \not\models \Phi_{u,v}$), and therefore $v \notin \mathit{Sat}(\Phi_u)$. Vice versa, if $v \in u{\downarrow}$ then $(v,u) \in \mathcal{R}$. As $u \models \Phi_u$ we get $v \models \Phi_u$ (by definition of $\mathcal{R}$). Hence, $v \in \mathit{Sat}(\Phi_u)$.

It remains to prove that whenever $(s_1,s_2) \in \mathcal{R}$ and $s_1 \to s_1'$, we have that $s_2 \to s_2'$ such that $(s_1',s_2') \in \mathcal{R}$. Since TS is finite, the set of direct successors of s_2 is finite, say $\mathit{Post}(s_2) = \{u_1,\ldots,u_k\}$. Let $\Phi_i = \Phi_{u_i}$ be the master formula for $u_i{\downarrow}$, for $0 < i \leqslant k$. From $u_i \in u_i{\downarrow}$, it follows $u_i \models \Phi_i$. Hence:

$$s_2 \models \forall\bigcirc \bigvee_{0<i\leqslant k} \Phi_i$$

Since $(s_1,s_2) \in \mathcal{R}$ and $\forall\bigcirc \bigvee_{0<i\leqslant k} \Phi_i$ is a ∀CTL formula, we have:

$$s_1 \models \forall\bigcirc \bigvee_{1\leqslant i\leqslant k} \Phi_i$$

Since $s_1' \in \mathit{Post}(s_1)$, $s_1' \in \mathit{Sat}(\Phi_i)$ for some $i \in \{1,\ldots,k\}$. Hence, $s_1' \in u_i{\downarrow}$, i.e., $(s_1',u_i) \in \mathcal{R}$. Taking $s_2' = u_i$, we obtain $s_2 \to s_2'$ and $(s_1',s_2') \in \mathcal{R}$.

■

Theorem 7.76 can be reformulated for the simulation order between transition systems. This yields

$$TS_1 \preceq TS_2 \quad \text{if and only if} \quad TS_2 \models \Phi \;\Rightarrow\; TS_1 \models \Phi$$

for any ∀CTL* (or ∀CTL) formula Φ. As the proof of Theorem 7.76 does not exploit the until operator, this result applies to the fragment of CTL* that consists of literals (i.e., atomic propositions and their negations), conjunction, disjunction, and the modality $\forall\bigcirc$.

Example 7.77. Distinguishing Nonsimilar Transition Systems

Consider the transition systems TS_1 (left) and TS_2 (right) in Figure 7.28. TS_1 is simulated by TS_2, but due to the self-loop in the initial state, $TS_2 \not\preceq TS_1$. This can also be established by providing a ∀CTL formula Φ such that, e.g., $TS_1 \models \Phi$ whereas $TS_2 \not\models \Phi$. Such a ∀CTL formula is

$$\Phi \;=\; \forall\bigcirc(\forall\bigcirc \neg a \;\vee\; \forall\bigcirc a).$$

■

An alternative logical characterization of $\preceq$ is obtained by replacing the universal quantifier ∀ by the existential quantifier ∃ using the CTL* duality rule: $\forall\varphi \;\equiv\; \neg\exists\neg\varphi$. Theorem 7.76 yields:

$$s_1 \;\preceq_{TS}\; s_2 \;\text{ iff }\quad \text{for all formulae } \exists\varphi\text{: } s_1 \models \exists\varphi \;\text{ implies }\; s_2 \models \exists\varphi.$$

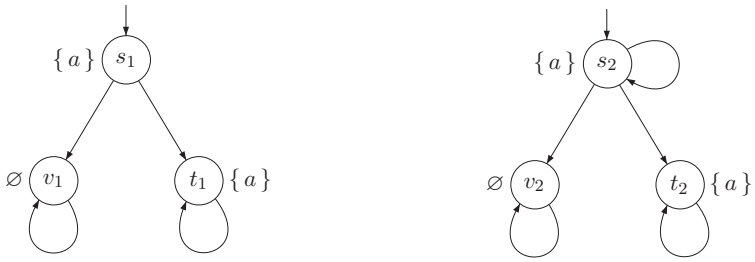


Figure 7.28: $TS_1 \preceq TS_2$, but $TS_2 \not\preceq TS_1$.

Here, φ is an arbitrary CTL* path formula that does not contain universal path quantifiers, and is in PNF. Such formulae are $\exists$CTL* formulae.

Definition 7.78. Existential Fragment of CTL*

The existential fragment of CTL*, denoted $\exists$CTL*, consists of the state formulae Φ and path formulae φ given, for $a \in AP$, by

$$\Phi \;::=\; \text{true} \mid \text{false} \mid a \mid \neg a \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid \exists\varphi$$

$$\varphi \;::=\; \Phi \mid \bigcirc\, \varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \,\mathsf{U}\, \varphi_2 \mid \varphi_1 \,\mathsf{R}\, \varphi_2.$$

■

The existential fragment of CTL, denoted $\exists$CTL, is obtained by restricting the path formulae in the above definition to

$$\varphi \;::=\; \bigcirc \Phi \mid \Phi_1 \,\mathsf{U}\, \Phi_2 \mid \Phi_1 \,\mathsf{R}\, \Phi_2.$$

The modalities eventually and always can be derived as for $\forall$CTL* and $\forall$CTL.

Theorem 7.79. Simulation Order and $\exists$CTL*/$\exists$CTL

Let TS be a finite transition system without terminal states, and s_1, s_2 states of TS. Then, the following statements are equivalent:

(a) $s_1 \preceq_{TS} s_2$.

(b) For all $\exists$CTL* formulae Φ: $s_1 \models \Phi$ *implies* $s_2 \models \Phi$.

(c) For all $\exists$CTL formulae Φ: $s_1 \models \Phi$ *implies* $s_2 \models \Phi$.

Proof: Directly from Theorem 7.76, as statements (b) and (c) here are the duals to statements (b) and (c) in Theorem 7.76. ■

Corollary 7.80. Characterizations of Simulation Equivalence

Let TS be a finite transition system without terminal states and s_1, s_2 be states in TS. Then the following five statements are equivalent:

(a) $s_1 \simeq_{TS} s_2$.

(b) s_1 *and* s_2 *satisfy the same* $\forall CTL^*$ *formulae.*

(c) s_1 *and* s_2 *satisfy the same* $\forall CTL$ *formulae.*

(d) s_1 *and* s_2 *satisfy the same* $\exists CTL^*$ *formulae.*

(e) s_1 *and* s_2 *satisfy the same* $\exists CTL$ *formulae.*

Proof: From Theorems 7.76 and 7.79. ■

In fact, this result can be strengthened as the listed equivalences coincide with the logical equivalence on the sublogic of CTL that just contains atomic propositions and their negations, conjunction, disjunction, and the next-step operator $\bigcirc$, with either a universal or existential path quantifier.

7.6 Simulation-Quotienting Algorithms

This section presents algorithms for obtaining the simulation quotient $TS/\simeq$ for finite transition system TS. Such algorithms serve two purposes. First, they can be used to verify whether $TS_1 \simeq TS_2$ by considering the simulation quotient of $TS_1 \oplus TS_2$ (see page 457). Secondly, such algorithms can be used to obtain the abstract (and thus smaller) transition system $TS/\simeq$ in a fully automated manner. As $TS \simeq TS/\simeq$, it follows that any verification result for $TS/\simeq$, either being negative or positive, carries over to TS. This applies to any formula expressed in either the universal or existential fragment of CTL or CTL*. Since simulation equivalence is coarser than bisimulation equivalence, the quotient $TS/\simeq$ is at most as large as—but maybe significantly smaller than—$TS/\sim$. The bisimulation quotient of a given infinite state transition system might be infinite whereas its simulation quotient is finite.

The goal of this section is to present an algorithm which takes as input a finite transition system $TS = (S, Act, \rightarrow, I, AP, L)$, possibly with terminal states, and computes the simulation order $\preceq_{TS}$. Clearly, this algorithm yields at the same time an automatic approach to check whether one finite transition system simulates another one and a sound, but incomplete technique for proving finite trace inclusion. The basic scheme for computing the simulation order is sketched in Algorithm 33.

Algorithm 33 Computation of the simulation order (basic idea)

Input: finite transition system TS over AP with state space S
Output: simulation order $\preceq_{TS}$

$\mathcal{R} := \{ (s_1, s_2) \mid L(s_1) = L(s_2) \}$;
while $\mathcal{R}$ is not a simulation **do**
  choose $(s_1, s_2) \in \mathcal{R}$ such that $s_1 \rightarrow s_1'$, but there is no s_2' such that $s_2 \rightarrow s_2'$ and $(s_1', s_2') \in \mathcal{R}$;
  $\mathcal{R} := \mathcal{R} \setminus \{ (s_1, s_2) \}$
od
return $\mathcal{R}$

The number of iterations is bounded above by $|S|^2$, since

$$S \times S \supseteq \mathcal{R}_0 \supsetneq \mathcal{R}_1 \supsetneq \mathcal{R}_2 \supsetneq \ldots \supsetneq \mathcal{R}_n = \preceq$$

where $\mathcal{R}_i$ denotes the relation $\mathcal{R}$ at the start of the $(i{+}1)$st iteration.

In the following, we discuss some details that permit an efficient implementation of this algorithm. Instead of explicitly representing $\mathcal{R}$, we use

$$Sim_{\mathcal{R}}(s_1) = \{ s_2 \in S \mid (s_1, s_2) \in \mathcal{R} \}.$$

Note that $Sim_{\mathcal{R}}(s_1)$ is a superset of $Sim_{TS}(s_1)$. This yields Algorithm 34.

A straightforward implementation of Algorithm 34 yields a time complexity $\mathcal{O}(M \cdot |S|^3)$, where M, the number of edges in the state graph $G(TS)$, is assumed to be at least $|S|$. Let us briefly sketch how this time complexity is established. In every iteration, it is checked for each transition $s_1 \rightarrow s_1'$, whether there exists a state $s_2 \in Sim(s_1)$ that *cannot* simulate this transition. Let counter $\delta(s_1', s_2)$ denote the number of successors of s_2 that belong to the current simulator set of s_1', i.e.,

$$\delta(s_1', s_2) = |Post(s_2) \cap Sim(s_1')|.$$

The initialization of these counters can be done in $\mathcal{O}(M \cdot |S|)$ by setting $\delta(s_1', s_2) = |Post(s_2)|$ for each $s_2 \in Sim(s_1)$ and $s_1' \in Post(s_1)$. Checking $Post(s_2) \cap Sim(s_1') = \varnothing$ reduces to check whether $\delta(s_1', s_2) = 0$. Given a matrix representation of δ, this takes $\mathcal{O}(1)$. Consider now the operations during an iteration. On removing state s_2 from $Sim(s_1)$, we set

Algorithm 34 Computation of the simulation order (first refinement)

Input: finite transition system TS over AP with state space S
Output: simulation order $\preceq_{TS}$

for all $s_1 \in S$ **do**
  $Sim(s_1) := \{ s_2 \in S \mid L(s_1) = L(s_2) \}$; (* initialization *)
od

while $\exists (s_1, s_2) \in S \times Sim(s_1). \exists s_1' \in Post(s_1)$ with $Post(s_2) \cap Sim(s_1') = \varnothing$ **do**
  choose such a pair of states (s_1, s_2); (* $s_1 \not\preceq_{TS} s_2$ *)
  $Sim(s_1) := Sim(s_1) \setminus \{ s_2 \}$;
od
(* $Sim(s) = Sim_{TS}(s)$ for any s *)
return $\{ (s_1, s_2) \mid s_2 \in Sim(s_1) \}$

$$\delta(s_1, v_2) := \delta(s_1, v_2) - 1 \text{ for all } v_2 \in Pre(s_2).$$

(The number of direct successors of v_2 that can simulate s_1 is reduced by one, viz. states s_2.) Assuming a list presentation of the sets $Pre(\cdot)$ and bit vector representations of the simulator sets $Sim(\cdot)$, the total time complexity of the body of the while-loop (when ranging over all iterations) is in $\mathcal{O}(M \cdot |S|)$. Note that each state s_2 is removed from $Sim(s_1)$ at most once. Hence, the total number of steps performed in the body of the while-loop is bounded by

$$\mathcal{O}\Big(\sum_{s_1 \in S} \underbrace{\sum_{s_2 \in S} |Pre(s_2)|}_{=M} \Big) \;=\; \mathcal{O}(M \cdot |S|).$$

In order to check whether $\mathcal{R}$ is a simulation, we may inspect all transitions $s_1 \to s_1'$ and all states s_2 and check whether the counter $\delta(s_1', s_2)$ is 0. If so and $(s_1, s_2) \in \mathcal{R}$, then $\mathcal{R}$ is not a simulation. Otherwise we have found a pair (s_1, s_2) to be removed from $\mathcal{R}$ in the body of the while-loop. As $|S|^2$ is an upper bound for the number of iterations of the while-loop and the costs required to check the condition of the while-loop are in $\mathcal{O}(M \cdot |S|)$, the total time complexity of Algorithm 34 is $\mathcal{O}(M \cdot |S|^3)$.

With a simple trick, the time complexity can be reduced to $\mathcal{O}(M \cdot |S|^2)$. The idea is to organize all pairs (s_1', s_2) where $\delta(s_1', s_2) = 0$ in a list and to pick one such pair (s_1', s_2) per iteration, rather than seeking for a pair (s_1, s_2) that violates the simulation condition. In each iteration, we run through the predecessor list of s_1' and check for each state $s_1 \in Pre(s_1')$ whether $s_2 \in Sim(s_1)$. If so, then we remove s_2 from $Sim(s_1)$ and decrement the counters $\delta(s_1, v_2)$ for all states $v_2 \in Pre(s_2)$. In case $\delta(s_1, v_2)$ becomes 0, the pair (s_1, v_2) is inserted in the list organizing all pairs where $\delta(\cdot)$ is 0.

An Efficiency Improvement We now discuss a further refinement of Algorithm 34 that leads to the worst-case time complexity $\mathcal{O}(M{\cdot}|S|)$. The crux of this more efficient realization is the following observation. Suppose $s_1 \to s_1'$ and $s_2 \to s_2'$, and we are about to remove s_2' from $\mathit{Sim}(s_1')$. If state s_2' is the *only* direct successor of s_2 that belongs to the current simulator set of s_1', i.e,

$$s_1 \in \mathit{Pre}(s_1'),\ s_2 \in \mathit{Sim}(s_1) \text{ and } \mathit{Sim}(s_1') \cap \mathit{Post}(s_2) = \{\, s_2' \,\},$$

then there does not exist a transition $s_2 \to s_2'$ which can simulate $s_1 \to s_1'$. Hence, s_2—in fact, any direct predecessor of s_2' for which this holds—can be safely removed from $\mathit{Sim}(s_1)$.

This observation can be generalized to sets of states. Let $\mathit{Sim}_{\mathit{old}}(s_1) \supseteq \mathit{Sim}(s_1)$ denote the simulator set that preceded the last removal of states from $\mathit{Sim}(s_1)$; initially $\mathit{Sim}_{\mathit{old}}(s_1) = S$. On the removal of s_2' from $\mathit{Sim}(s_1')$, we consider all direct predecessors of s_1', and remove all states in

$$\mathit{Remove}(s_1') \;=\; \mathit{Pre}\left(\mathit{Sim}_{\mathit{old}}(s_1')\right) \setminus \mathit{Pre}\left(\mathit{Sim}(s_1')\right)$$

from $\mathit{Sim}(s_1)$. This is justified as all the states in this set do not have a successor simulating s_1', and thus these states cannot simulate any of the predecessors of s_1'. (Note that $\mathit{Post}(s_2) \cap \mathit{Sim}(s_1') \neq \varnothing$ if and only if $s_2 \in \mathit{Pre}(\mathit{Sim}(s_1'))$.) This yields Algorithm 35, which also takes into account that a terminal state cannot be in the simulator set of a nonterminal state. Thus, when s_1' is considered the first time, then $\mathit{Remove}(s_1')$ consists of $\mathit{Pre}\,(S) \setminus \mathit{Pre}\,(\mathit{Sim}(s_1'))$ and all terminal states (that is, we define $\mathit{Remove}(s_1') = S \setminus \mathit{Pre}\,(\mathit{Sim}(s_1'))$).

The next observation is that there is no need to explicitly represent the sets $\mathit{Sim}_{\mathit{old}}(\cdot)$. The sets $\mathit{Remove}(\cdot)$ are dynamically adapted on modifying $\mathit{Sim}(s_1)$. The termination condition of the iteration can be replaced by $\mathit{Remove}(s_1') = \varnothing$ for all $s_1' \in S$. Intuitively speaking, this means that there are no states that need to be removed from the sets of simulators $\mathit{Sim}(s_1)$ for $s_1 \in \mathit{Pre}(s_1')$. Let s_1' be a state such that $\mathit{Remove}(s_1') \neq \varnothing$. Now consider all pairs $(s_1, s_2) \in S \times S$ such that

$$s_2 \in \mathit{Remove}(s_1') \;=\; \mathit{Pre}\left(\mathit{Sim}_{\mathit{old}}(s_1')\right) \setminus \mathit{Pre}\left(\mathit{Sim}(s_1')\right)$$

and $s_1 \in \mathit{Pre}(s_1')$. Then, $s_1 \to s_1'$, but there is no transition in $s_2 \to s_2' \in \mathit{Sim}(s_1')$. This yields $s_1 \not\preceq_{TS} s_2$. Therefore, s_2 can be removed from $\mathit{Sim}(s_1)$. Accordingly, the set $\mathit{Remove}(s_1)$ is extended with any state s such that

$$s \in \mathit{Pre}(s_2) \text{ and } \mathit{Post}(s) \cap \mathit{Sim}(s_1) = \varnothing.$$

For these states s, we have: if $u \to s_1$, then there is no matching transition $s \to t$ with $t \in \mathit{Sim}(s_1)$. Thus, $u \not\preceq_{TS} s$. Hence, if in a later iteration of the while-loop, state s_1 is

Algorithm 35 Computation of the simulation order (second refinement)

Input: finite transition system TS over AP with state space S
Output: simulation order $\preceq_{TS}$

for all $s_1 \in S$ **do**
  $Sim_{old}(s_1) :=$ undefined;
  $Sim(s_1) := \{ s_2 \in S \mid L(s_1) = L(s_2) \}$
od

while $\exists s \in S$ with $Sim_{old}(s) \neq Sim(s)$ **do**
  choose s'_1 such that $Sim_{old}(s'_1) \neq Sim(s'_1)$
  if $Sim_{old}(s'_1) =$ undefined **then**
    $Remove(s'_1) := S \setminus Pre(Sim(s'_1))$
  else
    $Remove(s'_1) := Pre(Sim_{old}(s'_1)) \setminus Pre(Sim(s'_1))$
  fi
  for all $s_1 \in Pre(s'_1)$ **do**
    $Sim(s_1) := Sim(s_1) \setminus Remove(s'_1)$
  od
  $Sim_{old}(s'_1) := Sim(s'_1)$
od
return $\{ (s_1, s_2) \mid s_2 \in Sim(s_1) \}$

chosen, then the simulator sets of the predecessors $u \in Pre(s_1)$ are regarded and state s is to be removed from $Sim(u)$. This yields Algorithm 36.

Algorithm 36 Computation of the simulation order

Input: finite transition system TS with state space S
Output: simulation order $\preceq_{TS}$

```
for all s1 ∈ S do
  Sim(s1) := { s2 ∈ S | L(s1) = L(s2) };
  Remove(s1) := S \ Pre(Sim(s1))
od
                                   (* loop invariant: Remove(s1') ⊆ S \ Pre(Sim(s1')) *)
while (∃ s1' ∈ S with Remove(s1') ≠ ∅) do
  choose s1' such that Remove(s1') ≠ ∅;
  for all s2 ∈ Remove(s1') do
    for all s1 ∈ Pre(s1') do
      if s2 ∈ Sim(s1) then
        Sim(s1) := Sim(s1) \ { s2 };                    (* s2 ∈ Sim_old(s1) \ Sim(s1) *)
        for all s ∈ Pre(s2) with Post(s) ∩ Sim(s1) = ∅ do
                                   (* s ∈ Pre(Sim_old(s1)) \ Pre(Sim(s1)) *)
          Remove(s1) := Remove(s1) ∪ { s }
        od
      fi
    od
  od
  Remove(s1') := ∅;                                     (* Sim_old(s1') := Sim(s1') *)
od
return { (s1, s2) | s2 ∈ Sim(s1) }
```

Theorem 7.81. Partial Correctness of Algorithm 36

On termination, Algorithm 36 returns $\preceq_{TS}$.

Proof: First, observe that the outermost loop (i.e., the while-loop) maintains the following loop invariant. For all states $s_1 \in S$:

(a) $Remove(s_1) \subseteq S \setminus Pre(Sim(s_1))$.

(b) $\{ s_2 \in S \mid s_1 \preceq_{TS} s_2 \} \subseteq Sim(s_1) \subseteq \{ s_2 \in S \mid L(s_1) = L(s_2) \}$.

(c) For all $s_2 \in Sim(s_1)$, one of the following two statements holds:

- $\exists s_1' \in \mathit{Post}(s_1)$ with $\mathit{Post}(s_2) \cap \mathit{Sim}(s_1') = \varnothing$ and $s_2 \in \mathit{Remove}(s_1')$,
- $\mathit{Post}(s_2) \cap \mathit{Sim}(s_1') \neq \varnothing$ for all $s_1' \in \mathit{Post}(s_1)$.

From (c), it follows that whenever $\mathit{Remove}(s_1') = \varnothing$ for all $s_1' \in S$, then:

$$\forall s_1 \in S. \forall s_2 \in \mathit{Sim}(s_1). \forall s_1' \in \mathit{Post}(s_1).\ \mathit{Post}(s_2) \cap \mathit{Sim}(s_1') \neq \varnothing.$$

Therefore, on termination, the relation $\mathcal{R} = \{ (s_1, s_2) \mid s_2 \in \mathit{Sim}(s_1) \}$ is a simulation for TS. Assertion (b) implies that $\mathcal{R}$ agrees with $\preceq_{TS}$, as $\preceq_{TS}$ is the coarsest simulation for TS. ■

Lemma 7.82. Termination of Algorithm 36

For each pair $(s_2, s_1') \in S \times S$, state s_2 is inserted in (and removed from) the set $\mathit{Remove}(s_1')$ at most once.

In particular, Algorithm 36 requires at most $\mathcal{O}(|S|^2)$ iterations.

Proof: Assume $s_2 \in \mathit{Remove}(s_1')$ and let s_1' be the state that is selected in the outermost iteration. Then $s_2 \notin \mathit{Pre}(\mathit{Sim}(s_1'))$ (by part (a) of the loop invariant established in the proof of Theorem 7.81). Since the simulator sets are decreasing, $s_2 \notin \mathit{Pre}(\mathit{Sim}(s_1'))$ in all further iterations: the only possibility to insert $s = s_2$ in $\mathit{Remove}(s_1')$ is when $s \in \mathit{Pre}(\bar{s}_2)$ for some state $\bar{s}_2 \in \mathit{Sim}(s_1')$ with $\mathit{Post}(s) \cap \mathit{Sim}(s_1') = \{ \bar{s}_2 \}$. But then $s_2 = s \in \mathit{Pre}(\mathit{Sim}(s_1'))$. Thus, s_2 will never be added to $\mathit{Remove}(s_1')$ once it has been deleted from $\mathit{Remove}(s_1')$. ■

This lemma is the starting point for deriving the time complexity of Algorithm 36. Let M be the number of edges in the state graph of TS and assume $M \geqslant |S|$. Assume that list representations for the sets of predecessors $\mathit{Pre}(\cdot)$ are available. As for the initial algorithm to compute $\preceq_{TS}$, we deal with counters

$$\delta(s_1, s) \ = \ |\mathit{Post}(s) \cap \mathit{Sim}(s_1)|.$$

Using Algorithm 29 (page 479), the initial simulator sets $\mathit{Sim}(s_1) = \{s_2 \in S \mid L(s_1) = L(s_2)\}$ can be obtained in time $\mathcal{O}(|S| \cdot |AP|)$. We then compute the sets $\mathit{Remove}(s_1)$ and the counters $\delta(s_1, s)$ in time $\mathcal{O}(M \cdot |S|)$. In each iteration the counters $\delta(s_1, s)$ need to be updated. This is done as follows. On removing s_2 from $\mathit{Sim}(s_1)$, the list $\mathit{Pre}(s_2)$ is traversed and we set

$$\delta(s_1, s) \ := \ \delta(s_1, s) - 1 \quad \text{for every } s \in \mathit{Pre}(s_1).$$

The check $\mathit{Post}(s) \cap \mathit{Sim}(s_1) = \varnothing$ boils down to $\delta(s_1, s) = 0$ and can be performed in constant time. The total costs of the outermost iteration can be estimated as follows. For

each state s_2 and each edge $s \to s_2$, the condition "$s_2 \in \mathit{Remove}(s_1') \ \wedge \ s_2 \in \mathit{Sim}(s_1)$" is satisfied at most once. Therefore, the code fragment

if $s_2 \in \mathit{Sim}(s_1)$ **then**
  $\mathit{Sim}(s_1) := \mathit{Sim}(s_1) \setminus \{ s_2 \}$;
  for all $s \in \mathit{Pre}(s_2)$ **do**
    if $\mathit{Post}(s) \cap \mathit{Sim}(s_1) = \varnothing$ **then**
      $\mathit{Remove}(s_1) := \mathit{Remove}(s_1) \cup \{ s \}$;
    fi
  od
fi

causes the cost:

$$\mathcal{O}\Big(\underbrace{\sum_{s_2 \in S} |\mathit{Pre}(s_2)|}_{M} \cdot \underbrace{\sum_{s_1 \in S} 1}_{|S|} \Big) \quad = \quad \mathcal{O}(M \cdot |S|),$$

where we add up the costs over all iterations for which $s_2 \in \mathit{Sim}(s_1)$. The condition $s_2 \in \mathit{Remove}(s_1')$ of the outermost for-loop is fulfilled at most once (see Lemma 7.82). Thus, the total costs for all inner loops

for all $s_2 \in \mathit{Remove}(s_1')$ **do**
  for all $s_1 \in \mathit{Pre}(s_2')$ **do**
    $\vdots$
  od
od

is bounded by $\mathcal{O}(M)$. Under the assumption that $M \geqslant |S|$, we thus obtain:

Theorem 7.83. Complexity of Algorithm 36

The simulation order $\preceq$ of finite transition system $TS = (S, \mathit{Act}, \to, I, AP, L)$ can be computed with Algorithm 36 in time $\mathcal{O}(M \cdot |S| \ + \ |S| \cdot |AP|)$.

The simulation equivalence $\simeq_{TS}$ and the simulation quotient system $TS/\simeq$ can be computed with the same complexity.

7.7 Stutter Linear-Time Relations

The equivalence and preorder relations considered so far require that each outgoing transition of state s is mimicked by an outgoing transition of the related state s'. We now consider variants of trace equivalence and bisimulation that relax this requirement. Rather than considering all transitions as "visible", it is allowed to mimic an outgoing transition s by a *sequence* of transitions starting from s'. Such sequences of transitions need to be "invisible", that is to say, all state changes (except the last state change) in such sequences should not change the truth value of the atomic propositions that hold in s'. Such state changes are called stutter steps. Equivalences and preorders that abstract from these stutter steps, also referred to as internal or nonobservable steps, are called *weak*. In contrast, trace equivalence, $\sim$, $\preceq$, and $\simeq$ are *strong* implementation relations as these relations consider stutter steps as any other transition. Weak implementation relations are important for system synthesis as well as system analysis. To compare transition systems that model a given system at different abstraction levels, it is often too demanding to require a statewise equivalence. Instead, an action in a transition system at a high level of abstraction can be modeled by a sequence of actions in the more concrete transition system. Establishing, e.g., trace equivalence is then simply impossible. This is illustrated by the following example.

Example 7.84. Abstraction of Internal Moves

Consider the abstract program fragment $x := y!$, and let TS_{abs} be its underlying transition system. Let TS_{conc} model the concrete program fragment

$i := y$; $z := 1$;
while $i > 1$ **do**
  $z := z * i$; $i := i - 1$;
od
$x := z$;

that computes the factorial of y iteratively. Clearly, TS_{abs} and TS_{conc} are not trace-equivalent. They, however, are related after abstracting from the iteration (and initial assignments) in the concrete program under the assumption that the iteration (plus initial assignments) does not affect the truth value of atomic propositions. This is guaranteed when restricting to atomic propositions that only refer to the values of x and y, and, e.g., not to the individual program locations. ■

Secondly, by abstracting from internal steps, quotient transition systems are obtained that may be significantly smaller than the quotient under the corresponding strong implementation relation. This is due to the fact that quotienting with respect to a weak relation

allows for abstracting from sequences of transitions such that all states on such paths can be aggregated. Interestingly, though, still a rather rich set of properties is preserved under such abstractions.

This section is concerned with a weak version of trace inclusion and trace equivalence. Section 7.8 (from page 536 onward) deals with weak versions of bisimulation.

7.7.1 Stutter Trace Equivalence

Internal steps are transitions that do not affect the state labels of successive states. Thus, intuitively, an internal step operates on program or control variables that are either not visible from the outside or viewed to be irrelevant at a certain abstraction level. Such transitions are called stutter steps.

Definition 7.85. Stutter Step

Transition $s \to s'$ in transition system $TS = (S, Act, \to, I, AP, L)$ is a *stutter step* if $L(s) = L(s')$. ■

The notion of stuttering is lifted to paths as follows. Two paths are called stutter-equivalent if their traces only differ in their stutter steps, i.e., if there is a sequence $A_0 A_1 A_2 \ldots$ of sets of atomic propositions $A_i \subseteq AP$ such that the traces of both paths have the form $A_0^+ A_1^+ A_2^+ \ldots$.

Definition 7.86. Stutter-Equivalence of Paths

Let $TS_i = (S_i, Act_i, \to_i, I_i, AP, L_i)$ be transition systems without terminal states and $\pi_i \in Paths(TS_i)$, $i = 1, 2$. π_1 and π_2 are *stutter-equivalent*, denoted $\pi_1 \triangleq \pi_2$, if there exists an infinite sequence $A_0 A_1 A_2 \ldots$ with $A_i \subseteq AP$ and natural numbers $n_0, n_1, n_2, \ldots$, $m_0, m_1, m_2, \ldots \geqslant 1$ such that

$$\begin{aligned} trace(\pi_1) &= \underbrace{A_0 \ldots A_0}_{n_0\text{-times}} \underbrace{A_1 \ldots A_1}_{n_1\text{-times}} \underbrace{A_2 \ldots A_2}_{n_2\text{-times}} \ldots \\ trace(\pi_2) &= \underbrace{A_0 \ldots A_0}_{m_0\text{-times}} \underbrace{A_1 \ldots A_1}_{m_1\text{-times}} \underbrace{A_2 \ldots A_2}_{m_2\text{-times}} \ldots \end{aligned}$$

Finite path fragments $\widehat{\pi}_1$ in TS_1 and $\widehat{\pi}_2$ in TS_2 are *stutter equivalent*, denoted $\widehat{\pi}_1 \triangleq \widehat{\pi}_2$, if there exists a finite sequence $A_0 \ldots A_n \in (2^{AP})^+$ such that $trace(\widehat{\pi}_1)$ and $trace(\widehat{\pi}_2)$ are contained in the language given by the regular expression $A_0^+ A_1^+ \ldots A_n^+$. ■

Note that it is not required that $A_0, A_1, A_2, \ldots$ are distinct. The notion of stutter equivalence can be applied to (finite or infinite) path fragments of transition system TS by setting $TS_1 = TS_2 = TS$.

Example 7.87. Stutter-Equivalent Paths

Consider the transition system TS_{Sem} depicted in Figure 7.29 and let $AP = \{ crit_1, crit_2 \}$.

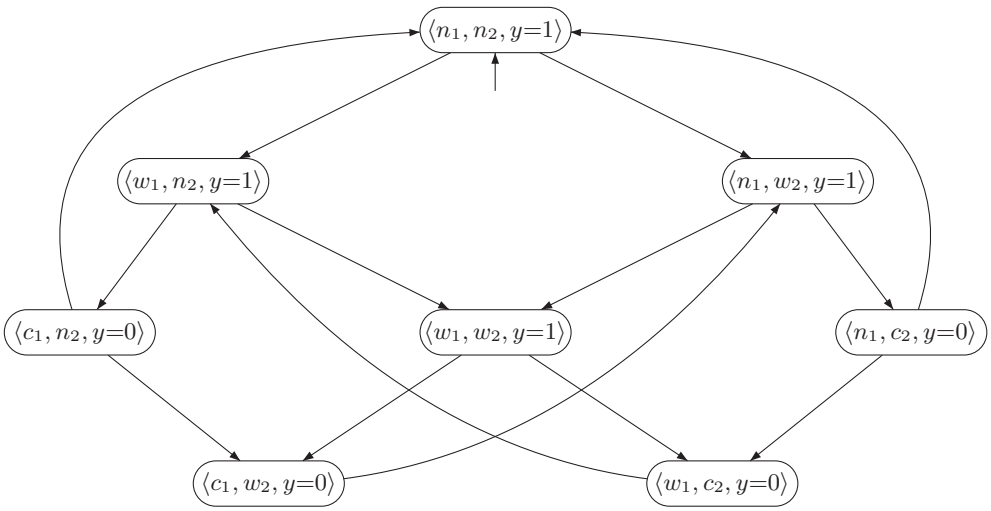


Figure 7.29: Transition system of semaphore-based mutual exclusion algorithm.

All infinite paths in TS_{Sem} that agree on the first process acquiring access to the critical section, and in which the two processes acquire access in a strictly alternating fashion, are stutter-equivalent. For the case that process P_1 is the first to enter the critical section, all such paths have a trace of the form

$$\varnothing \ldots \varnothing \underbrace{\{ crit_1 \} \ldots \{ crit_1 \}}_{P_1 \text{ in cs}} \varnothing \ldots \varnothing \underbrace{\{ crit_2 \} \ldots \{ crit_2 \}}_{P_2 \text{ in cs}} \varnothing \ldots \varnothing \underbrace{\{ crit_1 \} \ldots \{ crit_1 \}}_{P_1 \text{ in cs}} \ldots$$

Consider, for instance, the following two infinite paths in TS_{Sem}:

$$\begin{aligned} \pi_1 &= \langle n_1, n_2 \rangle \to \langle w_1, n_2 \rangle \to \langle w_1, w_2 \rangle \to \langle c_1, w_2 \rangle \to \langle n_1, w_2 \rangle \to \\ &\quad \langle n_1, c_2 \rangle \to \langle n_1, n_2 \rangle \to \langle w_1, n_2 \rangle \to \langle w_1, w_2 \rangle \to \langle c_1, w_2 \rangle \to \ldots \\ \pi_2 &= \langle n_1, n_2 \rangle \to \langle w_1, n_2 \rangle \to \langle c_1, n_2 \rangle \to \langle c_1, w_2 \rangle \to \langle n_1, w_2 \rangle \to \\ &\quad \langle w_1, w_2 \rangle \to \langle w_1, c_2 \rangle \to \langle w_1, n_2 \rangle \to \langle c_1, n_2 \rangle \to \ldots \end{aligned}$$

where for the sake of simplicity, we omitted the value of variable y in each state. Hence,

$\pi_1 \triangleq \pi_2$, since for $AP = \{ crit_1, crit_2 \}$

$$\begin{aligned} trace(\pi_1) &= \varnothing^3 \{ crit_1 \} \varnothing \{ crit_2 \} \varnothing^3 \{ crit_1 \} \ldots \text{ and} \\ trace(\pi_2) &= \varnothing^2 (\{ crit_1 \})^2 \varnothing^2 \{ crit_2 \} \varnothing \{ crit_1 \} \ldots \end{aligned}$$

Figure 7.30 indicates $trace(\pi_1)$ and $trace(\pi_2)$ and their stutter equivalence. ■

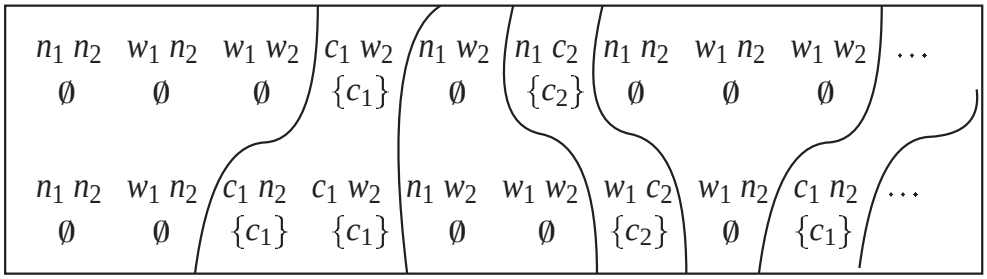


Figure 7.30: Stutter-equivalent paths in TS_{Sem}.

Notation 7.88. Stutter Equivalence for Executions and Traces

The notion of stutter equivalence can be adapted to execution fragments and words over 2^{AP} in the obvious way. Two execution fragments ρ_1 and ρ_2 of TS_1 and TS_2 are stutter-equivalent, denoted $\rho_1 \triangleq \rho_2$, if the induced path fragments are stutter-equivalent. Accordingly, traces σ_1 and σ_2 over 2^{AP} are stutter-equivalent, denoted $\sigma_1 \triangleq \sigma_2$, if they are both of the form $A_0^+ A_1^+ A_2^+ \ldots$ for $A_0, A_1, A_2, \ldots \subseteq AP$. ■

Transition systems TS_1 and TS_2 are stutter-equivalent whenever each trace of TS_1 can be mimicked by a stutter equivalent trace in TS_2, and vice versa.

Definition 7.89. Stutter Equivalence of Transition Systems

Transition systems TS_i over AP, i=1, 2, are *stutter trace equivalent*, denoted $TS_1 \triangleq TS_2$, if $TS_1 \trianglelefteq TS_2$ and $TS_2 \trianglelefteq TS_1$, where $\trianglelefteq$ is defined by:

$$TS_1 \trianglelefteq TS_2 \quad \text{iff} \quad \forall \sigma_1 \in Traces(TS_1) \ \Big(\exists \sigma_2 \in Traces(TS_2). \ \sigma_1 \triangleq \sigma_2 \Big).$$

■

Evidently, it follows that $Traces(TS_1) \subseteq Traces(TS)2$ implies $TS_1 \trianglelefteq TS_2$.

Example 7.90. Stutter Trace Inclusion

Consider the transition systems TS_1 (left), TS_2 (middle) and TS_3 (right) in Figure 7.31. It follows:

$$\begin{aligned} \mathit{Traces}(TS_1) &= \{\, (a\varnothing)^\omega, (a\varnothing)^*\, a^\omega \,\} \\ \mathit{Traces}(TS_2) &= \{\, (a^+\varnothing)^\omega, (a^+\varnothing)^*\, a^\omega \,\} \\ \mathit{Traces}(TS_3) &= \{\, a^\omega, a^+\, (\varnothing a)^\omega \,\}. \end{aligned}$$

We have $TS_1 \trianglelefteq TS_2$ since $\mathit{Traces}(TS_1) \subseteq \mathit{Traces}(TS_2)$, and $TS_2 \trianglelefteq TS_1$ as trace $(a^+\varnothing)^\omega$ is stutter-equivalent to $(a\varnothing)^\omega$ and trace $(a^+\varnothing)^*\, a^\omega$ is stutter-equivalent to $(a\varnothing)^*\, a^\omega$. Accordingly,

$$TS_1 \triangleq TS_2.$$

Note that $\mathit{Traces}(TS_2) \not\subseteq \mathit{Traces}(TS_1)$, as e.g., $a^2\varnothing \ldots \in \mathit{Traces}(TS_2)$ but is not a trace of TS_1. It follows that

$$TS_1 \not\trianglelefteq TS_3 \text{ and } TS_2 \not\trianglelefteq TS_3,$$

as TS_1 and TS_2 have traces of the form $(a^+\,\varnothing)^+ a^\omega$ whereas there is no trace in TS_3 that contains $\varnothing$ and ends with a^ω. Since $\mathit{Traces}(TS_3) \subseteq \mathit{Traces}(TS_2)$, it follows $TS_3 \trianglelefteq TS_2$. $TS_3 \trianglelefteq TS_1$ since a^ω is a trace of both transition systems, and $a^+\,(\varnothing a)^\omega$ of TS_3 is stutter-equivalent to $(a\varnothing)^\omega$. ■

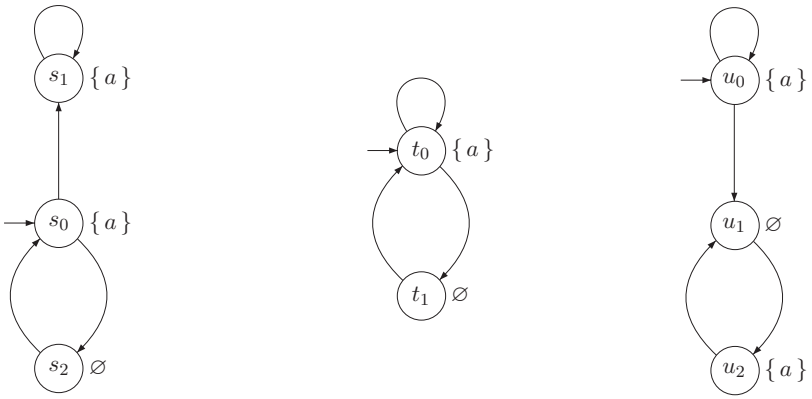


Figure 7.31: Three transition systems.

7.7.2 Stutter Trace and LTL$_{\setminus\bigcirc}$ Equivalence

In analogy to the result stating that trace equivalence is finer than LTL equivalence, we now consider the fragment of LTL that is preserved by stutter trace equivalence. To that end, we first determine the properties that are preserved by stutter-equivalent paths and words over 2^{AP}, respectively. Let us discuss why a restriction to logics without the next operator is necessary. Consider, e.g., the traces $\sigma_1 = A\,B\,B\,B\ldots$ and $\sigma_2 = A\,A\,A\,B\,B\,B\,B\ldots$ with $A, B \subseteq AP$ and $A \neq B$. Clearly,

$$\sigma_1 \triangleq \sigma_2 \quad \text{but} \quad \sigma_1 \models \bigcirc b \quad \text{and} \quad \sigma_2 \not\models \bigcirc b \text{ for } b \in B \setminus A.$$

Stated in words: stutter equivalence does not preserve the truth value of formulae with the next operator. In fact, it turns out that this is the only modal operator that is not preserved.

Notation 7.91. LTL without Next Step

LTL$_{\setminus\bigcirc}$ denotes the class of LTL formulae without the next step operator $\bigcirc$. ■

Theorem 7.92. Stutter Equivalence and LTL$_{\setminus\bigcirc}$ Equivalence

For $\sigma_1, \sigma_2 \in (2^{AP})^\omega$:

$$\sigma_1 \triangleq \sigma_2 \;\Rightarrow\; (\sigma_1 \models \varphi \textit{ if and only if } \sigma_2 \models \varphi)$$

for any LTL$_{\setminus\bigcirc}$ formula φ over AP.

Proof: The proof is by structural induction over the formula φ. Let $A_0 A_1 A_2 \ldots$ be an infinite word over 2^{AP} and

$$\sigma_1 = A_0^{n_0} A_1^{n_1} A_2^{n_2} \ldots \quad \text{and} \quad \sigma_2 = A_0^{m_0} A_1^{m_1} A_2^{m_2} \ldots$$

where $n_0, n_1, n_2, \ldots$ and $m_0, m_1, m_2, \ldots$ are positive natural numbers. Hence: $\sigma_1 \triangleq \sigma_2$.

Basis: For $\varphi = \text{true}$ the proposition is clear. For $\varphi = a \in AP$, we have

$$\sigma_1 \models a \quad \text{iff} \quad a \in A_0 \quad \text{iff} \quad \sigma_2 \models a.$$

Induction step: For $\varphi = \varphi_1 \wedge \varphi_2$ or $\varphi = \neg\varphi'$, the claim follows immediately from the induction hypothesis applied to φ_1 and φ_2 or φ', respectively. The remaining case is $\varphi = \varphi_1 \,\mathsf{U}\, \varphi_2$. Assume $\sigma_1 \models \varphi$. From the semantics of LTL$_{\setminus\bigcirc}$, it follows that there exists a natural number j such that

$$\sigma_1[j..] \models \varphi_2 \quad \text{and} \quad \sigma_1[i..] \models \varphi_1 \text{ for all } 0 \leqslant i < j.$$

Recall that for $\sigma = B_0\, B_1\, B_2 \ldots$ and $h \geqslant 0$, the suffix $B_h B_{h+1} \ldots$ of σ is denoted by $\sigma[h..]$.

Let $r \geqslant 0$ be such that

$$n_0 + \ldots + n_{r-1} \;<\; j \;\leqslant\; n_0 + \ldots + n_{r-1} + n_r.$$

Stated in words, r is the index of the A-block in σ_1 that contains $\sigma_1[j]$. Then, $\sigma_1[j..]$ is obtained from σ_1 by eliminating the prefix $A_0^{n_0} \ldots A_{r-1}^{n_{r-1}} A_r^{n}$ where $n = n_0 + \ldots + n_{r-1} + n_r - j$. Note that $0 \leqslant n < n_r$. Thus, $\sigma_1[j..]$ is of the form $A_r^+ A_{r+1}^+ A_{r+2}^+ \ldots$. Since $\sigma_1 \triangleq \sigma_2$, it follows that for:

$$k = m_0 + \ldots + m_{r-1} + 1,$$

$\sigma_2[k..]$ is of the form $A_r^+ A_{r+1}^+ A_{r+2}^+ \ldots$. More precisely, we have

1. $\sigma_1[j..] \triangleq \sigma_2[k..]$ since both words are of the form $A_r^+ A_{r+1}^+ A_{r+2}^+ \ldots$, and
2. for all $0 \leqslant h < k$, there is an index $0 \leqslant i < j$ such that $\sigma_1[i..] \triangleq \sigma_2[h..]$.

As $\sigma_1[j..] \models \varphi_2$ and $\sigma_1[i..] \models \varphi_1$, for all $i < j$, applying the induction hypothesis yields $\sigma_2[k..] \models \varphi_2$ and $\sigma_2[h..] \models \varphi_1$, for $h < k$. Hence, $\sigma_2 \models \varphi_1 \,\mathsf{U}\, \varphi_2$.

By symmetry, we get the equivalence of σ_1 and σ_2 for $\mathrm{LTL}_{\setminus\bigcirc}$. ■

Corollary 7.93. Stutter Trace Relations and LTL$_{\setminus\bigcirc}$

For transition systems TS_1, TS_2 (over AP) without terminal states:

(a) $TS_1 \triangleq TS_2$ implies $TS_1 \equiv_{LTL\setminus\bigcirc} TS_2$.

(b) if $TS_1 \trianglelefteq TS_2$, then for any $LTL_{\setminus\bigcirc}$ formula φ: $TS_2 \models \varphi$ implies $TS_1 \models \varphi$.

A slightly more general preservation result can be established for stutter trace equivalence and inclusion.

Definition 7.94. Stutter-Insensitive LT property

LT property P is *stutter-insensitive* if $[\sigma]_{\triangleq} \;\subseteq\; P$, for any $\sigma \in P$. ■

Stated in words, P is stutter-insensitive if it is closed under stutter equivalence, i.e., for any $\sigma \in P$ all stutter-equivalent words are also contained in P. It follows immediately that stutter trace equivalent transition systems satisfy the same stutter-insensitive LT properties. Moreover, for any stutter-insensitive LT property P:

$$TS_1 \trianglelefteq TS_2 \text{ and } TS_2 \models P \quad \text{implies} \quad TS_1 \models P.$$

This is, in fact, a more general statement than Corollary 7.93 since for any $\text{LTL}_{\setminus\bigcirc}$ formula φ the induced LT property $\mathit{Words}(\varphi)$ is stutter-insensitive. On the other hand, there exist stutter-insensitive LT properties that cannot be expressed in $\text{LTL}_{\setminus\bigcirc}$. For instance, consider the set P of words over $\{a, b\}$ that contain an odd number of occurrences of the subword $a\,b$. The LT property that is stutter-insensitive and contains P, cannot be expressed in $\text{LTL}_{\setminus\bigcirc}$. However, if φ is an LTL formula such that $\mathit{Words}(\varphi)$ is stutter insensitive, then φ is equivalent to some $\text{LTL}_{\setminus\bigcirc}$ formula ψ; see Exercise 7.20.

7.8 Stutter Bisimulation

This section is concerned with stutter bisimulation. Stutter bisimulation is defined in a coinductive manner, as bisimulation. Whereas bisimulation requires for equivalent states s_1 and s_2 that each transition $s_1 \to t_1$ (with s_1 inequivalent to t_1) is matched by some transition $s_2 \to t_2$, stutter bisimulation allows $s_1 \to t_1$ to be matched by a path fragment $s_2\, u_1\, u_2 \ldots u_n\, t_2$ (for $n \geqslant 0$) such that t_1 and t_2 are equivalent, and u_i is equivalent to s_2. That is, single transitions may be matched by (suitable) path fragments.

The following definition considers the notion of a stutter bisimulation for a single transition system. Later on, this notion will be adapted for pairs of transition systems.

Definition 7.95. Stutter Bisimulation

Let $TS = (S, \mathit{Act}, \to, I, \mathit{AP}, L)$ be a transition system. A *stutter bisimulation* for TS is a binary relation $\mathcal{R}$ on S such that for all $(s_1, s_2) \in \mathcal{R}$:

1. $L(s_1) = L(s_2)$.
2. If $s_1' \in \mathit{Post}(s_1)$ with $(s_1', s_2) \notin \mathcal{R}$, then there exists a finite path fragment $s_2\, u_1\, \ldots\, u_n\, s_2'$ with $n \geqslant 0$ and $(s_1, u_i) \in \mathcal{R}$, $i = 1, \ldots, n$ and $(s_1', s_2') \in \mathcal{R}$.
3. If $s_2' \in \mathit{Post}(s_2)$ with $(s_1, s_2') \notin \mathcal{R}$, then there exists a finite path fragment $s_1\, v_1\, \ldots\, v_n\, s_1'$ with $n \geqslant 0$ and $(v_i, s_2) \in \mathcal{R}$, $i = 1, \ldots, n$ and $(s_1', s_2') \in \mathcal{R}$.

s_1, s_2 are *stutter bisimulation equivalent* (stutter-bisimilar, for short), denoted $s_1 \approx_{TS} s_2$, if there exists a stutter bisimulation $\mathcal{R}$ for TS with $(s_1, s_2) \in \mathcal{R}$. ■

Condition (1) is standard, and requires equivalent states to be equally labeled. According to condition (2), every outgoing transition $s_1 \to t_1$ (where s_1 is not equivalent to t_1) must be matched by a path fragment that leads from s_2 to t_2 such that t_1 and t_2 are equivalent, and all intermediate states in the path fragment are equivalent to s_2. Roughly speaking, if s_1 changes its equivalence class and moves to t_1, this must be mimicked by s_2, but only after some transitions that are internal to the equivalence class of s_2. Condition (3) is the symmetric counterpart of condition (2).

Lemma 7.96. Coarsest Stutter Bisimulation

For transition system TS with state space S:

1. $\approx_{TS}$ *is an equivalence relation on* S.
2. $\approx_{TS}$ *is a stutter bisimulation for TS.*
3. $\approx_{TS}$ *is the coarsest stutter bisimulation for TS and coincides with the union of all stutter bisimulations for TS.*

Proof: Similar to the case for bisimulation; see Exercise 7.26. ■

(Note that $\approx_{TS}$ is an equivalence, but this does not hold for any stutter bisimulation relation.) Since $\approx_{TS}$ is a stutter bisimulation, condition (2) of Definition 7.95 can be rephrased by means of $\approx_{TS}$, as illustrated in Figure 7.32.

Example 7.97. Stutter Bisimulation in Semaphore-Based Mutual Exclusion

Consider the semaphore-based solution to the mutual exclusion problem; see the transition system TS_{Sem} in Figure 5.5 (page 239). Let $AP = \{ crit_1, crit_2 \}$. Relation $\mathcal{R}$ that induces the following partitioning of the state space is a stutter bisimulation:

$$\{ \{ \langle n_1, n_2 \rangle, \langle n_1, w_2 \rangle, \langle w_1, n_2 \rangle, \langle w_1, w_2 \rangle \}, \{ \langle c_1, n_2 \rangle, \langle c_1, w_2 \rangle \}, \{ \langle c_2, n_1 \rangle, \langle w_1, c_2 \rangle \} \} .$$

This can be checked by verifying the conditions in Definition 7.95. For instance, $\langle n_1, n_2 \rangle \approx_{TS} \langle w_1, w_2 \rangle$, since $\langle w_1, w_2 \rangle \to \langle w_1, c_2 \rangle$ can be mimicked by the path fragment $\langle n_1, n_2 \rangle \to \langle n_1, w_2 \rangle \to \langle n_1, c_2 \rangle$, and $\langle w_1, w_2 \rangle \to \langle c_1, w_2 \rangle$ is matched by path fragment $\langle n_1, n_2 \rangle \to \langle w_1, n_2 \rangle \to \langle c_1, n_2 \rangle$. As $\langle n_1, n_2 \rangle$ does not have any direct successor that is inequivalent to

$$
\begin{array}{lll}
s_1 & \approx_{TS} & s_2 \\
\downarrow & & \\
s_1' & & \\
(\text{with } s_1 \not\approx_{TS} s_1') & &
\end{array}
\qquad \text{can be completed to} \qquad
\begin{array}{lll}
s_1 & \approx_{TS} & s_2 \\
 & & \downarrow \\
s_1 & \approx_{TS} & u_1 \\
 & & \downarrow \\
s_1 & \approx_{TS} & u_2 \\
 & & \downarrow \\
 & & \vdots \\
 & & \downarrow \\
s_1 & \approx_{TS} & u_n \\
\downarrow & & \downarrow \\
s_1' & \approx_{TS} & s_2'
\end{array}
$$

Figure 7.32: Condition (2) for stutter bisimulation equivalence.

it, no requirements are imposed on $\langle w_1, w_2 \rangle$. In a similar way, the stutter-bisimilarity of other pairs of states can be checked. ■

Example 7.98. Stutter Bisimulation in Peterson's Mutual Exclusion Algorithm

Consider the transition system TS_{Pet} for Peterson's mutual exclusion algorithm (see Figure 7.33 on page 539) and let $AP = \{ crit_1, crit_2 \}$. The initial states

$$s_{0,1} \;=\; \langle n_1, n_2, x{=}1 \rangle \quad \text{and} \quad s_{0,2} \;=\; \langle n_1, n_2, x{=}2 \rangle$$

are stutter-bisimilar. All other states represent separate stutter bisimulation equivalence classes. Essentially, this is due to the fact that

$$\underbrace{\langle c_1, n_2, x{=}2 \rangle}_{s_1} \quad \not\approx_{TS} \quad \underbrace{\langle c_1, w_2, x{=}1 \rangle}_{s_2}.$$

This follows from the fact that s_1 can move to one of the initial states, whereas s_2 cannot mimic this by a series of stutter steps. In state s_1, it is possible that process 1 reenters the critical section next, whereas in state s_2, process 2 is always granted access to the critical section next. For the other states, a similar reasoning applies. ■

Stutter bisimulation for pairs of transition systems is defined as follows.

Definition 7.99. Stutter-Bisimilar Transition Systems

Let $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP, L_i)$, $i = 1, 2$, be transition systems over AP. TS_1 and

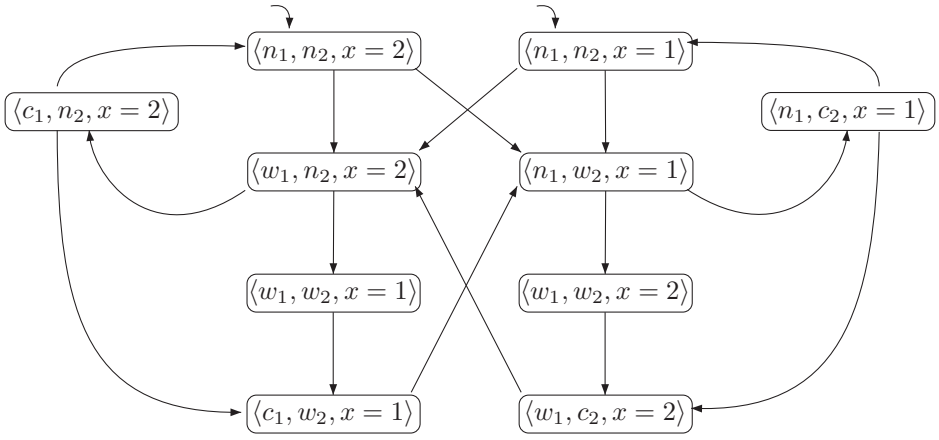


Figure 7.33: Transition system for Peterson's mutual exclusion algorithm.

TS_2 are stutter bisimulation equivalent (stutter-bisimilar, for short), denoted $TS_1 \approx TS_2$, if there exists a stutter bisimulation $\mathcal{R}$ on $(S_1 \times S_2) \cup (S_1 \times S_2)$ such that

$$\forall s_1 \in I_1.\, (\exists s_2 \in I_2.\, (s_1, s_2) \in \mathcal{R}) \text{ and } \forall s_2 \in I_2.\, (\exists s_1 \in I_1.\, (s_1, s_2) \in \mathcal{R}).$$

■

The connection between Definitions 7.95 and 7.99 is as for ordinary bisimulation. We have $TS_1 \approx TS_2$ if and only if for every initial state of TS_1 there exists a stutter-bisimilar initial state of TS_2, and vice versa, where stutter bisimulation equivalence of states s_1 in TS_1 and s_2 in TS_2 thereby refers to the stutter bisimulation equivalence $\approx_{TS_1 \oplus TS_2}$. Recall that the operation $\oplus$ on transition systems amounts to the union of transition systems, see page 457). Vice versa, stutter bisimulation equivalence $\approx_{TS}$ of a single transition system can be obtained from Definition 7.99 by the observation that $s_1 \approx_{TS} s_2$ if and only if $TS_{s_1} \approx TS_{s_2}$, where TS_s is obtained from TS by declaring s as the unique initial state.

Figure 7.34 surveys the stutter implementation relations in this monograph. Stutter bisimulation equivalence with divergence is a variant of stutter bisimulation and will be introduced in Section 7.8.1 (page 543 and further).

Example 7.100. Door Opener

Consider an automatic door opener as modeled by the (concrete) transition system in Figure 7.35; for simplicity, the action labels are omitted from the transitions. Let $AP = \{\, \mathit{alarm}, \mathit{open} \,\}$. The door opener requires a three-digit code $d_1\, d_2\, d_3$ as input with $d_i \in$

stutter trace inclusion:

$TS_1 \trianglelefteq TS_2$ iff $\forall \sigma_1 \in \mathit{Traces}(TS_1)\ \exists \sigma_2 \in \mathit{Traces}(TS_2).\ \ \sigma_1 \stackrel{\triangle}{=} \sigma_2$

stutter trace equivalence:

$TS_1 \stackrel{\triangle}{=} TS_2$ iff $TS_1 \trianglelefteq TS_2$ and $TS_2 \trianglelefteq TS_1$

stutter bisimulation equivalence:

$TS_1 \approx TS_2$ iff there exists a stutter bisimulation for (TS_1, TS_2)

stutter bisimulation equivalence with divergence:

$TS_1 \approx^{div} TS_2$ iff there exists a divergence-sensitive stutter bisimulation for (TS_1, TS_2)

Figure 7.34: Stutter implementation relations.

$\{0, \ldots, 9\}$. It allows an erroneous digit to be entered, but this may happen at most twice. Location ℓ_i (for $i = 0, 1, 2$) indicates that the first i digits of the code have been correctly entered.

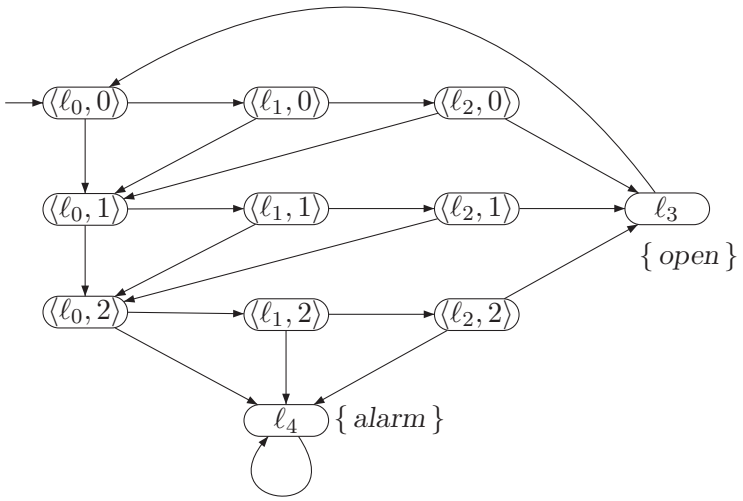


Figure 7.35: Transition system of the door opener

The transition system in Figure 7.36 is stutter-bisimilar to the transition system in Figure 7.35. ■

The quotient transition system under stutter bisimulation $\approx_{TS}$ is defined as the quotient under bisimulation. As a stutter bisimulation is not guaranteed to be transitive, the

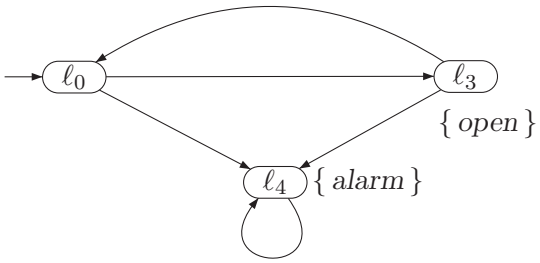


Figure 7.36: Stutter-bisimilar transition system for the door opener.

quotient under the equivalence $\approx_{TS}$ is considered. Note that, of course, states are now equivalence classes under $\approx_{TS}$ If $s \to s'$ and $s \not\approx s'$, there is a transition from $[s]_\approx$ to $[s']_\approx$. As a result, $TS/\approx$ contains no self-loops.

Definition 7.101. Stutter Bisimulation Quotient System

For transition system $TS = (S, Act, \to, I, AP, L)$, the *stutter bisimulation quotient* transition system $TS/\approx$ is defined as follows:

$$TS/\approx \;=\; (S/\approx_{TS}, \{\,\tau\,\}, \to_\approx, I_\approx, AP, L_\approx)$$

where

- $I_\approx \;=\; \{\,[s]_\approx \mid s \in I\,\}$,
- $\to_\approx$ is defined by $\dfrac{s \xrightarrow{\alpha} s' \;\wedge\; s \not\approx s'}{[s]_\approx \xrightarrow{\tau}_\approx [s']_\approx}$,
- $L_\approx([s]_\approx) = L(s)$.

■

Theorem 7.102. Stutter Bisimulation Equivalence of TS ***and*** $TS/\approx$

For any transition system TS, we have $TS \approx TS/\approx$.

Proof: Follows from the fact that $\mathcal{R} = \{\,(s, s') \mid s' \in [s]_\approx, s \in S\,\}$ is a stutter bisimulation for $(TS, TS/\approx)$. ■

Example 7.103. Semaphore-Based Mutual Exclusion

Consider again the transition system TS_{Sem} for the semaphore-based solution to the mutual exclusion problem (see Figure 5.5 on page 239). Let $AP = \{ crit_1, crit_2 \}$. Recall that $\mathcal{R}$ inducing the following partitioning of the state space is a stutter bisimulation:

$$\{ \{ \langle n_1, n_2 \rangle, \langle n_1, w_2 \rangle, \langle w_1, n_2 \rangle, \langle w_1, w_2 \rangle \}, \{ \langle c_1, n_2 \rangle, \langle c_1, w_2 \rangle \}, \{ \langle c_2, n_1 \rangle, \langle w_1, c_2 \rangle \} \} \ .$$

In fact, this is the coarsest stutter bisimulation, i.e., $\mathcal{R}$ equals $\approx_{TS}$. It is not difficult to see that the quotient under bisimulation, i.e., $TS_{Sem}/\sim$ does not yield any state space reduction. The quotient transition system under $\approx$, i.e., $TS_{Sem}/\approx$, is depicted in Figure 7.37 and does yield a substantial reduction of the state space. ■

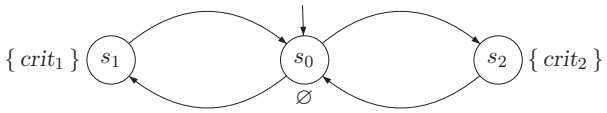


Figure 7.37: The stutter bisimulation quotient system of TS_{Sem}.

Let us now discuss the relationship between stutter trace equivalence and stutter bisimulation. Obviously, for transition systems that do not have any stutter steps, $\sim$ and $\approx$ coincide. The same applies to trace equivalence and stutter trace equivalence ($\triangleq$). Therefore, in general, stutter trace equivalent transition systems are *not* stutter-bisimilar. As an example, consider trace equivalent transition systems without stutter steps, which are not bisimilar. As we have seen, $TS_1 \sim TS_2$ implies that TS_1 and TS_2 are trace equivalent, see Theorem 7.6 (page 456). Based on this fact, one is tempted to assume that stutter bisimulation is finer than stutter trace equivalence. This is, however, *not* true, as stutter bisimulation does not impose any restrictions on paths that just consist of stutter steps. Only paths that "switch" equivalence class have to be matched, but pure stutter paths not—condition (2) of Definition 7.95 (page 536) only requires that, if s_1 and s_2 are stutter-bisimilar, then every transition $s_1 \to s_1'$ with $s_1 \not\approx s_1'$ can be simulated by a path fragment $s_2 \to \ldots \to s_2'$. For stutter steps $s_1 \to s_1'$, within the stutter bisimulation equivalence class of s_1 (i.e., $s_1 \approx s_1'$), no conditions are imposed. That is, it is possible that

$$s_1 \approx s_2 \quad \text{while} \quad \exists \pi_1 \in \mathit{Paths}(s_1).\ \forall \pi_2 \in \mathit{Paths}(s_2).\ \pi_1 \stackrel{\triangleq}{\neq} \pi_2.$$

Theorem 7.104. Stutter Trace vs. Stutter Bisimulation Equivalence

$\triangleq$ *and* $\approx$ *are incomparable.*

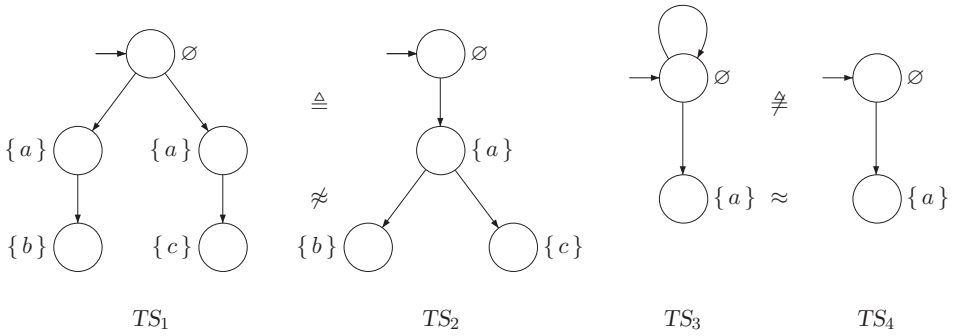


Figure 7.38: Stutter trace vs. stutter bisimulation equivalence.

Proof: see Figure 7.38. For the two leftmost transition systems TS_1 and TS_2, we have

$$TS_1 \triangleq TS_2, \text{ but } TS_1 \not\approx TS_2.$$

(In fact, we have $\mathit{Traces}(TS_1) = \mathit{Traces}(TS_2)$ and $TS_1 \not\sim TS_2$.) For the two rightmost transition systems, we have $TS_3 \approx TS_4$, while $TS_3 \not\triangleq TS_4$. The fact that the transition systems are not stutter trace equivalent follows from the fact that TS_3 exhibits the trace $\varnothing^\omega$, whereas TS_4 cannot generate this trace. Note that the trace $\varnothing^\omega$ just consists of stutter steps. ■

As a consequence, stutter bisimulation does not preserve the truth value of all $\mathrm{LTL}_{\setminus\bigcirc}$ formulae. This is illustrated by the following example.

Example 7.105. Stutter Bisimulation Equivalence vs. $LTL_{\setminus\bigcirc}$ Equivalence

Consider the transition systems TS_1 (left) and TS_2 (right) in Figure 7.39. Then $TS_1 \approx TS_2$ and $TS_2 \models \Diamond a$. However, $TS_1 \not\models \Diamond a$ since the path s_0^ω violates $\Diamond a$. This is due to the fact that the path s_0^ω just consists of stutter steps, and by definition of $\approx$ is not required to have a stutter-equivalent path starting in t_0. ■

7.8.1 Divergence-Sensitive Stutter Bisimulation

The fact that stutter trace equivalence ($\triangleq$) and stutter bisimulation ($\approx$) are incomparable is caused by stutter paths, i.e., paths that only consists of stutter steps. The existence of

Figure 7.39: Stutter bisimulation, but not $\mathrm{LTL}_{\setminus\bigcirc}$ equivalent systems.

these paths is also the reason that $\approx$ does not imply $\mathrm{LTL}_{\setminus\bigcirc}$ equivalence. Stutter paths stay forever in an equivalence class without performing any visible step. This behavior is called *divergent*.

The topic of this section is to adapt stutter bisimulation such that states may only be related if they both exhibit divergent paths or none of them has a divergent path. This yields a variant of stutter bisimulation that will be shown to coincide with $\mathrm{CTL}^*_{\setminus\bigcirc}$ equivalence, i.e., equivalence with respect to CTL^* formulae that do not contain the next-step operator. (Note that $\mathrm{CTL}^*_{\setminus\bigcirc}$ includes $\mathrm{LTL}_{\setminus\bigcirc}$.) This divergence-sensitive variant of stutter bisimulation is the coarsest equivalence that preserves all $\mathrm{CTL}^*_{\setminus\bigcirc}$ formulae. Besides, in contrast to $\approx$, this variant is strictly finer that stutter trace equivalence.

Definition 7.106. $\mathcal{R}$-Divergent State, Divergence Sensitivity

Let TS be a transition system and $\mathcal{R}$ an equivalence relation on S.

- $s \in S$ is *$\mathcal{R}$-divergent* if there exists an infinite path fragment $\pi = s\, s_1\, s_2 \ldots \in \mathit{Paths}(s)$ such that $(s, s_j) \in \mathcal{R}$ for all $j > 0$.
- $\mathcal{R}$ is *divergence-sensitive* if for any $(s_1, s_2) \in \mathcal{R}$: if s_1 is $\mathcal{R}$-divergent, then s_2 is $\mathcal{R}$-divergent.

■

Stated in words, a state is $\mathcal{R}$-divergent if there is an infinite path starting in s that only visits states in $[s]_{\mathcal{R}}$. As $\mathcal{R}$ is an equivalence, for $(s_1, s_2) \in \mathcal{R}$ we have s_1 is $\mathcal{R}$-divergent if and only if s_2 is $\mathcal{R}$-divergent. Equivalence $\mathcal{R}$ is thus divergence-sensitive if in any $\mathcal{R}$ equivalence class, either all states are $\mathcal{R}$-divergent or none is $\mathcal{R}$-divergent.

Example 7.107. Divergence Sensitivity

For the transition system in Figure 7.40, we have $s_0 \approx_{TS} s_1 \approx_{TS} s_2$. States s_0 and s_1 are

$\approx_{TS}$-divergent as in both states it is possible to infinitely often alternate between s_0 and s_1, i.e., stay forever in $[s_0]_{\approx}$. As state s_2 is not $\approx_{TS}$-divergent, it follows that $\approx_{TS}$ is not divergence-sensitive. Equivalence $\mathcal{R} = \{ (s_0, s_1), (s_1, s_0) \} \cup Id$ is divergence-sensitive. ■

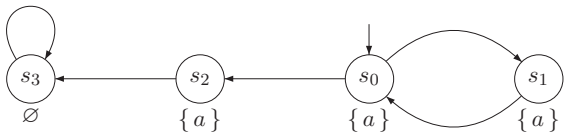


Figure 7.40: s_0 and s_1 are $\approx_{TS}$-divergent, but s_2 is not.

Example 7.108. Alternating Bit Protocol

Consider the alternating bit protocol as introduced in Chapter 2; see Example 2.32 on page 57. Recall that messages from the sender to the receiver are equipped with a bit that toggles between transmissions of new messages and stays constant on retransmissions. Accordingly, the sender can be either in a "0-mode" to transmit a message with bit 0, or in the "1-mode". Similarly, the receiver can be either expecting a message with bit 0 or with bit 1. Let

$$AP = \{ \mathit{s\_mode} = 0, \mathit{s\_mode} = 1, \mathit{r\_mode} = 0, \mathit{r\_mode} = 1 \}$$

where *s_mode* indicates the mode of the sender, and *r_mode* that of the receiver. The state space of the underlying TS_{ABP} of the alternating bit protocol consists of hundreds of states for a channel capacity of one (for both channels). The stutter bisimulation quotient just consists of four states; see Figure 7.41 where the state labeling (i, j) is a shorthand for $\mathit{s\_mode} = i$ and $\mathit{r\_mode} = j$, $i, j \in \{ 0, 1 \}$. By definition, no state in $TS_{ABP}/\approx$

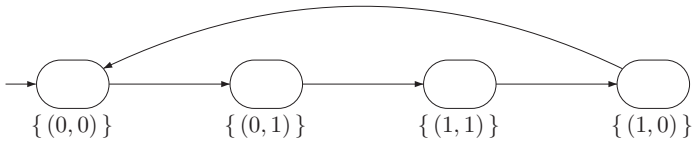


Figure 7.41: Stutter bisimulation quotient of the alternating bit protocol.

is $\approx$-divergent. However, TS_{ABP} contains (various) states that are $\approx$-divergent, as in these states it is possible to continuously lose a message. For instance, in states in the equivalence class $\{ (0,0) \}$ it is possible that the transmitted message with bit 0 as sent by the sender is lost infinitely often. Thus:

$$TS_{ABP} \not\models \forall\Box\Diamond(\mathit{s\_mode} = 0) \quad \wedge \quad \forall\Box\Diamond(\mathit{s\_mode} = 1),$$

but

$$TS_{ABP}/\approx \;\models\; \forall\Box\Diamond(s\_mode = 0) \quad \wedge \quad \forall\Box\Diamond(s\_mode = 1).$$

This confirms our earlier observation that $\approx$ does not guarantee the preservation of $\mathrm{LTL}_{\setminus\bigcirc}$ formulae. ■

Definition 7.109. Stutter Bisimulation with Divergence

States s_1, s_2 in transition system TS are *divergent stutter bisimilar*, denoted $s_1 \approx_{TS}^{div} s_2$, if there exists a divergence-sensitive stutter bisimulation $\mathcal{R}$ on TS such that $(s_1, s_2) \in \mathcal{R}$. ■

It is not difficult to show that $\approx_{TS}^{div}$ is an equivalence on S, and the coarsest divergence-sensitive stutter bisimulation for TS which is obtained by the union of all divergence-sensitive stutter bisimulations for TS.

Example 7.110. Stutter Bisimulation with Divergence

Consider the transition system TS in Figure 7.40 (page 545). Its equivalence classes under $\approx_{TS}^{div}$ are $\{ s_0, s_1 \}$, $\{ s_2 \}$ and $\{ s_3 \}$. State s_2 is $\approx_{TS}^{div}$-divergent whereas s_0 and s_1 are not. Thus, $s_2 \not\approx_{TS}^{div} s_0$ and $s_2 \not\approx_{TS}^{div} s_1$.

As a second example, consider the transition system in Figure 7.42 where the state labeling is indicated by the grey scale. $s_1 \not\approx_{TS}^{div} s_2$ as u_3 and u_1, u_2 are not equivalent under $\approx_{TS}^{div}$ (since only u_3 is divergent) and only s_2 (but not s_1) has a transition to u_3. ■

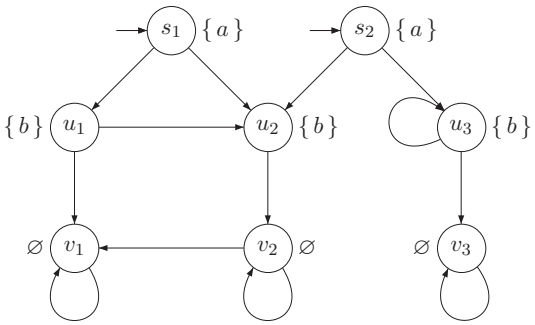


Figure 7.42: $s_1 \not\approx_{TS}^{div} s_2$.

$TS_1 \approx^{div} TS_2$ if and only if each initial state of TS_1 is divergence stutter bisimilar (according to $\approx_{TS_1 \oplus TS_2}^{div}$) to an initial state in TS_2, and vice versa. The quotient transition

system with respect to stutter bisimulation $\approx_{TS}^{div}$ is defined as usual with equivalence classes under $\approx_{TS}^{div}$ as states. In addition to the usual transitions, every equivalence class C consisting of divergent states is equipped with a self-loop. This self-loop indicates the divergence. (Recall that $TS/\mathcal{R}$ for stutter bisimulation $\mathcal{R}$ does not contain any self-loop.) That is, the transition relation of $TS/\approx^{div}$ is defined by

$$\frac{s \xrightarrow{\alpha} s' \ \wedge \ s \not\approx^{div} s'}{[s]_{div} \xrightarrow{\tau}_{div} [s']_{div}} \quad \text{and} \quad \frac{s \text{ is } \approx^{div}\text{-divergent}}{[s]_{div} \xrightarrow{\tau}_{div} [s]_{div}}$$

where $[s]_{div}$ denotes the equivalence class of s under $\approx^{div}$.

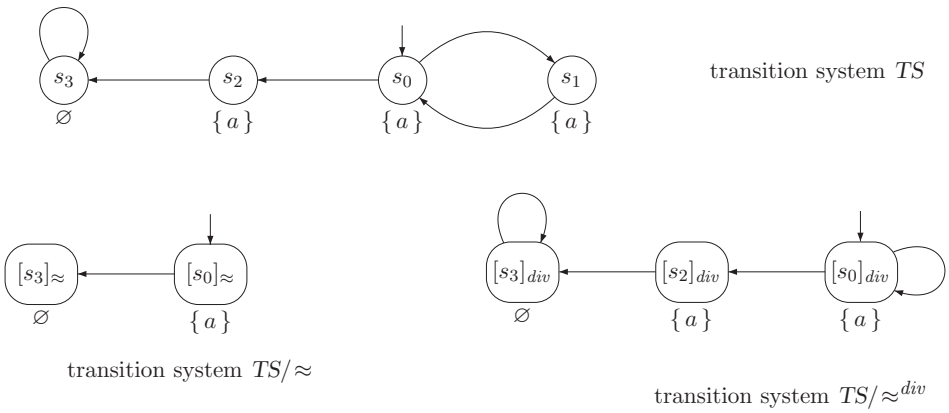


Figure 7.43: Transition system TS, $TS/\approx$, and $TS/\approx^{div}$.

Example 7.111. Quotient Transition System under $\approx$ and $\approx^{div}$

Consider the transition system TS in Figure 7.43 (upper part). Its quotient under stutter bisimulation, $TS/\approx$, is depicted in the left lower part of the figure. States s_0, s_1, and s_2 are all stutter-bisimilar. Note that, by definition, no self-loops occur in $TS/\approx$; only the transitions that change equivalence class are of importance. $TS/\approx^{div}$ is depicted in the right lowerpart of Figure 7.43. State s_2 is not equivalent to s_0 and s_1 as it is not divergent, whereas s_0 and s_1 are divergent. ■

Theorem 7.112. *TS and $TS/\approx^{div}$ are Divergence Stutter Bisimilar*
For any transition system TS, we have $TS \approx^{div} TS/\approx^{div}$.

Proof: Follows from the fact that $\mathcal{R} = \{(s, [s]_{div}) \mid s \in S\}$ is a divergence-sensitive stutter bisimulation for $\left(TS, TS/\approx^{div}\right)$. ■

Since divergence-sensitive stutter bisimulations are special cases of arbitrary stutter bisimulations, stutter bisimulation with divergence $\approx^{div}$ is strictly finer than stutter bisimulation $\approx$. In the example of Figure 7.40 on page 545, $s_0 \approx_{TS} s_2$ but $s_0 \not\approx^{div}_{TS} s_2$.

Lemma 7.113. $\approx^{div}$ ***is Strictly Finer Than*** $\approx$

For transition systems TS_1 and TS_2:

$$\underbrace{TS_1 \approx^{div} TS_2}_{\substack{\text{stutter bisimulation equivalence} \\ \text{with divergence}}} \quad \textit{implies} \quad \underbrace{TS_1 \approx TS_2}_{\substack{\text{stutter bisimulation equivalence} \\ \text{(without divergence)}}}$$

while the reverse does not hold in general.

Remark 7.114. Terminal and Purely Divergent States

State s is *purely divergent* if all paths starting in s are infinite and completely consist of stutter steps. Since $\approx$ imposes no restrictions on transitions inside equivalence classes, any purely divergent state is stutter-bisimilar to any equally labeled state which is terminal. However, $\approx^{div}$ distinguishes terminal states and purely divergent states. Thus,

$$s_t \approx_{TS} s_{pd}, \quad \text{while} \quad s_t \not\approx^{div}_{TS} s_{pd}$$

where s_t is a terminal state and s_{pd} a purely divergent state and $L(s_t) = L(s_{pd})$.

Both $\approx_{TS}$ and $\approx^{div}_{TS}$ identify any terminal state s_t with any state s with $L(s) = L(s_t)$ and for which $Paths(s)$ consists of *finite* stutter paths only. In other words, all reachable states from s have the same labeling as s and there is no infinite path starting in s. In fact, there are no other states that are divergence stutter bisimilar to s_t. That is, if s_t is terminal and $s_t \approx^{div}_{TS} s$, then $L(s) = L(s_t)$ and each path of s is finite and only consists of stutter steps. ■

Remark 7.115. Divergence-Sensitive Transition Systems

Transition system TS is called divergence-sensitive if $\approx_{TS}$ on TS is divergence-sensitive. Clearly, for divergence-sensitive transition systems, $\approx_{TS}$ corresponds to $\approx^{div}_{TS}$. For instance, the transition system shown in Figure 7.40 on page 545 is not divergence-sensitive. ■

Recall that $\triangleq$ (stutter trace equivalence) and $\approx$ (stutter bisimulation) are incomparable, see Theorem 7.104 (page 543). This is due to the fact that $\approx$ ignores divergent paths, whereas $\triangleq$ does not. In the sequel, it will be shown that $\approx^{div}$ is strictly finer than $\triangleq$. The proof technique is rather similar to proving that bisimulation is finer than trace equivalence and is based on a lifting of $\approx^{div}$ to paths.

Definition 7.116. Divergence Stutter-Bisimilar Paths

Let TS be a transition system.

1. For infinite path fragments $\pi_i = s_{0,i}\, s_{1,i}\, s_{2,i} \ldots$, $i = 1, 2$, in TS:

$$\pi_1 \approx_{TS}^{div} \pi_2$$

if and only if there exists an infinite sequence of indices $0 = j_0 < j_1 < j_2 < \ldots$ and $0 = k_0 < k_1 < k_2 < \ldots$ with:

$$s_{j,1} \approx_{TS}^{div} s_{k,2} \text{ for all } j_{r-1} \leqslant j < j_r \text{ and } k_{r-1} \leqslant k < k_r \text{ with } r = 1, 2, \ldots.$$

2. For finite paths fragments $\widehat{\pi}_1 = s_{0,i}\, s_{1,i} \ldots s_{K_i,i}$, $i = 1, 2$, in TS:

$$\widehat{\pi}_1 \approx_{TS}^{div} \widehat{\pi}_2$$

if and only if there exists a finite sequence of indices $0 = j_0 < j_1 < \ldots < j_\ell = K_1 + 1$ and $0 = k_0 < k_1 < \ldots < k_\ell = K_2 + 1$ with $s_{j,1} \approx_{TS}^{div} s_{k,2}$ for all $j_{r-1} \leqslant j < j_r$ and $k_{r-1} \leqslant k < k_r$ with $r = 1, 2, \ldots, \ell$.

■

Paths π_1 and π_2 are stutter-bisimilar with divergence if both paths can be divided into segments $s_{j_r,1}\, s_{j_r+1,1}\, s_{j_r+2,1} \ldots s_{j_{r+1},1}$ and $s_{k_r,2}\, s_{k_r+1,2}\, s_{k_r+2,2} \ldots s_{k_{r+1},2}$, respectively, that are statewise bisimilar (under $\approx^{div}$).

The following lemma follows directly from the definition of $\approx^{div}$ on paths and $\triangleq$ on paths; see Definition 7.86 on page 530.

Lemma 7.117. Equivalences $\approx^{div}$ and $\triangleq$ for Paths

For all infinite paths π_1 and π_2, we have $\pi_1 \approx_{TS}^{div} \pi_2$ implies $\pi_1 \triangleq_{TS} \pi_2$.

Divergence stutter bisimilar states have divergence stutter bisimilar paths:

Lemma 7.118. Path Lifting for Divergence Stutter Bisimilar States

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system, $s_1, s_2 \in S$. Then:

$$s_1 \approx_{TS}^{div} s_2 \quad \textit{implies} \quad \forall \pi_1 \in Paths(s_1).\ \Big(\exists \pi_2 \in Paths(s_2).\, \pi_1 \approx_{TS}^{div} \pi_2\Big).$$

Proof: Let $\pi_1 = s_{0,1}\, s_{1,1}\, s_{2,1} \ldots \in \mathit{Paths}(s_1)$. The proof technique is to successively define a statewise stutter-bisimilar path π_2 starting in s_2 by "lifting" the transitions $s_{i,1} \to s_{i+1,1}$ with $s_{i,1} \not\approx^{div}_{TS} s_{i+1,1}$ to finite path fragments $s_{i,2}\, u_{i,1} \ldots u_{i,n_i}\, s_{i+1,2}$ such that $s_{i+1,1} \approx^{div}_{TS} s_{i+1,2}$ and $s_{i,2} \approx^{div}_{TS} u_{i,1} \approx^{div}_{TS} \ldots \approx^{div}_{TS} u_{i,n_i}$.

The proof is by induction on i. The base case $i{=}0$ is straightforward and omitted. Assume $i \geqslant 0$ and that the path fragment:

$$s_2 = s_{0,2}\, u_{0,1}, \ldots, u_{0,n_0}\, s_{1,2}\, u_{1,1} \ldots u_{1,n_1}\, s_{2,2} \ldots s_{i,2} (*)$$

is already constructed. In particular, $s_{i,1} \approx^{div}_{TS} s_{i,2}$. If $s_{i,1}$ is a terminal state, then there exists a finite path fragment $s_{i,2}\, v_1 \ldots v_m$ consisting of stutter steps such that v_m is a terminal (see Remark 7.114 on page 548). Hence, the path π_2 resulting by concatenating the above path fragment $(*)$ from s_2 to $s_{i,2}$ and the path fragment $s_{i,2}\, v_1 \ldots v_m$ fulfills the desired conditions. In the sequel, we assume that $s_{i,1}$ is not a terminal state, and, hence, π_1 does not end in state $s_{i,1}$. Distinguish two cases:

1. $s_{i,1} \not\approx^{div}_{TS} s_{i+1,1}$. Since $s_{i,1} \approx^{div}_{TS} s_{i,2}$ and $s_{i,1} \to s_{i+1,1}$, there exists a finite path fragment $s_{i,2}\, u_{i,1} \ldots u_{i,n_i} s_{i+1,2}$ such that:

   $$s_{i+1,1} \approx^{div}_{TS} s_{i+1,2} \quad \text{and} \quad s_{i,2} \approx^{div}_{TS} u_{i,1} \approx^{div}_{TS} \ldots \approx^{div}_{TS} u_{i,n_i}.$$

   Concatenating the path $(*)$ with the path fragment $s_{i,2}\, u_{i,1} \ldots u_{i,n_i} s_{i+1,2}$ yields a path fragment that fulfills the desired conditions.

2. $s_{i,1} \approx^{div}_{TS} s_{i+1,1}$. Distinguish between $s_{i,1}$ is divergent and not divergent:

   (a) $s_{i,1}$ is not divergent, i.e, there exists an index $j > i{+}1$ with $s_{i,1} \not\approx^{div}_{TS} s_{j,1}$. Without loss of generality, assume that j is minimal, i.e., $s_{i,1}, s_{i+1,1}, \ldots, s_{j-1,1}$ are pairwise equivalent under $\approx^{div}$. In particular,

   $$s_{i,2} \approx^{div}_{TS} s_{j-1,1} \quad \text{and} \quad s_{j-1,1} \not\approx^{div}_{TS} s_{j,1}.$$

   As $s_{i,2} \approx^{div}_{TS} s_{j-1,1}$ and $s_{j-1,1} \to s_{j,1}$, there exists a finite path fragment $s_{i,2}\, u_{i,1} \ldots u_{i,n_i}\, s_{i+1,2}$ such that

   $$s_{j,1} \approx^{div}_{TS} s_{i+1,2} \quad \text{and} \quad s_{i,2} \approx^{div}_{TS} u_{i,1} \approx^{div}_{TS} \ldots \approx^{div}_{TS} u_{i,n_i}.$$

   Concatenation of the path fragment $(*)$ with $s_{i,2}\, u_{i,1} \ldots u_{i,n_i}\, s_{i+1,2}$ yields a path that fulfills the desired conditions.

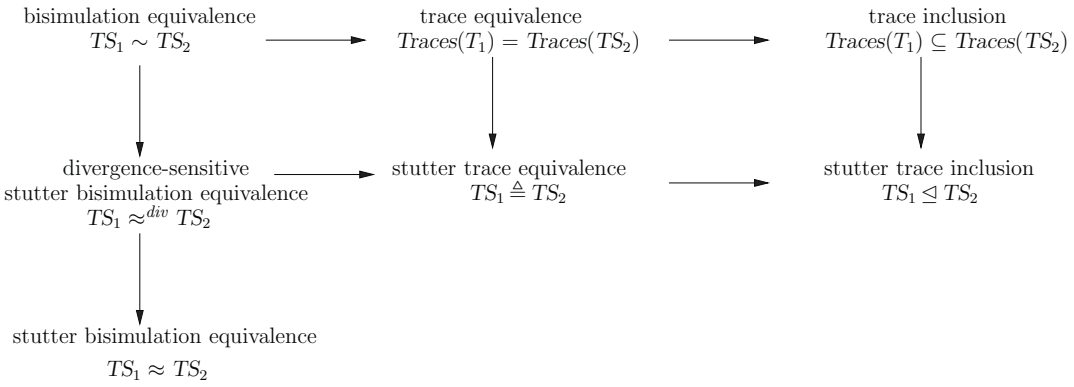


Figure 7.44: Relation between weak equivalences and preorders on transition systems.

(b) $s_{i,1}$ is divergent, i.e., $s_{i,1} \approx_{TS}^{div} s_{j,1}$ for all $j \geqslant i$. As $s_{i,1} \approx_{TS}^{div} s_{i,2}$, and $s_{i,1}$ is divergent, $s_{i,2}$ is divergent. That is, there is a path $s_{i,2}\, s_{i+1,2}\, s_{i+2,2} \ldots$ with $s_{i,2} \approx_{TS}^{div} s_{i+1,2} \approx_{TS}^{div} s_{i+2,2} \approx_{TS}^{div} \ldots$. Concatenation of the path fragment $(*)$ with $s_{i,2}\, s_{i+1,2}\, s_{i+2,2} \ldots$ yields a path that fulfills the desired conditions.

The thus resulting path fragment π_2 is divergence stutter bisimilar to π_1. ■

The fact that $\approx^{div}$ can be lifted from states to paths enables to establish that $\approx^{div}$ on transition systems is strictly finer than $\triangleq$, i.e., stutter trace-equivalence.

Theorem 7.119. Stutter Trace vs. Divergence Stutter Bisimulation

Let TS_1 and TS_2 be transition systems over AP. Then:

$$\underbrace{TS_1 \approx^{div} TS_2}_{\substack{\text{stutter bisimulation equivalence} \\ \text{with divergence}}} \quad \textit{implies} \quad \underbrace{TS_1 \triangleq TS_2}_{\text{stutter trace equivalence}} \quad ,$$

whereas the reverse implication does not hold in general.

Proof: Assume $TS_1 \approx^{div} TS_2$. Consider the transition system $TS_1 \oplus TS_2$, and let (s_1, s_2) be a pair of initial states of TS_1 and TS_2 such that $s_1 \approx_{TS}^{div} s_2$. From Lemma 7.118, it follows that for every $\pi_1 \in Paths(s_1)$, there exists $\pi_2 \in Paths(s_2)$ such that $\pi_1 \approx_{TS}^{div} \pi_2$. By Lemma 7.117, $\pi_1 \triangleq \pi_2$. Thus, $TS_1 \triangleq TS_2$. The reverse direction does not apply in general. Let TS_1, TS_2 be trace equivalent but not bisimilar, and neither contains any stutter steps. The absence of stutter steps yields $TS_1 \triangleq TS_2$, but $TS_1 \not\approx^{div} TS_2$. ■

The relationship between the various stutter equivalences and stutter trace preorder relations are summarized in Figure 7.44. An arrow from relation $\mathcal{R}$ to $\mathcal{R}'$ denotes that $\mathcal{R}$ is strictly finer than $\mathcal{R}'$. Note that for AP-deterministic transition systems, trace equivalence coincides with bisimulation, and stutter trace equivalence coincides with divergence-sensitive stutter bisimulation. The latter fact is stated without proof, and is left as an exercise for the reader.

7.8.2 Normed Bisimulation

This section introduces the notion of normed bisimulation. These bisimulations are defined using so-called *norm functions*. Normed bisimulation turns out to be strictly finer than $\approx^{div}$. As norm functions allow establishing a bisimulation by just local reasoning about the states—only considering their direct successors—means that it is often simpler to establish a normed bisimulation than a divergence-sensitive stutter bisimulation. We introduce a norm function which yields a sufficient (but not necessary) requirement for $\approx^{div}$. This will be used in Chapter 8.

Definition 7.120. Normed (Bi)Simulation

Let $TS_1 = (S_1, Act_1, \rightarrow_1, I_1, AP, L_1)$ and $TS_2 = (S_2, Act_2, \rightarrow_2, I_2, AP, L_2)$ be transition systems over AP.

A *normed simulation* for (TS_1, TS_2) is a triple $(\mathcal{R}, \nu_1, \nu_2)$ consisting of a binary relation $\mathcal{R} \subseteq S_1 \times S_2$ such that:

$$\forall s_1 \in I_1 . \exists s_2 \in I_2 . (s_1, s_2) \in \mathcal{R}$$

and functions $\nu_1, \nu_2 : S_1 \times S_2 \to \mathbb{N}$ such that for all $(s_1, s_2) \in \mathcal{R}$:

(NI) $L_1(s_1) = L_2(s_2)$.

(NII) For all $s_1' \in Post(s_1)$, at least one of the following three conditions holds:

- (N1) There exists $s_2' \in Post(s_2)$ with $(s_1', s_2') \in \mathcal{R}$.
- (N2) $(s_1', s_2) \in \mathcal{R}$ and $\nu_1(s_1', s_2) < \nu_1(s_1, s_2)$.
- (N3) There exists $s_2' \in Post(s_2)$ with $(s_1, s_2') \in \mathcal{R}$ and $\nu_2(s_1, s_2') < \nu_2(s_1, s_2)$.

A *normed bisimulation* for (TS_1, TS_2) is a normed simulation $(\mathcal{R}, \nu_1, \nu_2)$ for (TS_1, TS_2) such that $(\mathcal{R}^{-1}, \nu_2^-, \nu_1^-)$ is a normed simulation for (TS_2, TS_1). Here, ν_i^- denotes the function $S_2 \times S_1 \to \mathbb{N}$ that results from ν_i by swapping the arguments, i.e., $\nu_i^-(u, v) = \nu_i(v, u)$ for all $u \in S_2$ and $v \in S_1$.

TS_1 and TS_2 are *normed-bisimilar*, denoted $TS_1 \approx^n TS_2$, if there exists a normed bisimulation for (TS_1, TS_2). ■

For transition system TS, a normed bisimulation for TS is a normed bisimulation for (TS, TS). States s_1 and s_2 of TS are called normed-bisimilar, denoted $s_1 \approx^n_{TS} s_2$, if there exists a normed bisimulation $(\mathcal{R}, \nu_1, \nu_2)$ for TS such that $(s_1, s_2) \in \mathcal{R}$.

ν_1 and ν_2 are norm functions. The values $\nu_i(s_1, s_2)$ for $(s_1, s_2) \notin \mathcal{R}$ are irrelevant. Hence, ν_i could have been defined as function $\mathcal{R} \to \mathbb{N}$. The intuitive meaning of $\nu_1(s_1, s_2)$ for $(s_1, s_2) \in \mathcal{R}$ is as follows. $\nu_1(s_1, s_2)$ serves as a count down for the number "allowed" stutter steps from s_1 that cannot be mimicked by transitions of s_2. Similarly, $\nu_2(s_1, s_2)$ can be regarded as a counter for the number of stutter steps that s_2 may perform to reach a state where the visible (nonstutter) steps of s_1 can be simulated.

Example 7.121. Normed Bisimulation Equivalence

Consider the transition system TS in Figure 7.40 on page 545. The coarsest equivalence $\mathcal{R}$ which identifies s_0 and s_1 together with $\nu_1(s_0, s_1) = \nu_2(s_1, s_0) = 1$ and $\nu_2(s_1, s_2) = 1$, $\nu_1(s, s) = \nu_2(s, s) = 0$ for all $s \in \{ s_0, s_1, s_2, s_3 \}$ (and arbitrary values for ν_1 and ν_2 for the remaining cases) is a normed bisimulation. This is checked as follows. It suffices to only consider $(s_0, s_1), (s_1, s_0) \in \mathcal{R}$ and their outgoing transitions.

- For $s_0 \to s_1$, case (N1) applies as $s_0 \to s_1$ and $(s_0, s_1) \in \mathcal{R}$.
- For $s_1 \to s_0$, case (N1) applies as $s_1 \to s_0$ and $(s_0, s_1) \in \mathcal{R}$.
- For $s_0 \to s_2$, case (N3) applies as $s_1 \to s_0$, $(s_0, s_0) \in \mathcal{R}$ and $0 = \nu_2(s_0, s_0) < \nu_2(s_0, s_1) = 1$.
- For $s_1 \to s_3$, case (N3) applies as $s_0 \to s_1$, $(s_1, s_1) \in \mathcal{R}$ and $0 = \nu_2(s_1, s_1) < \nu_2(s_1, s_0)$.

Thus, $s_0 \approx^n_{TS} s_1$.

We have $s_1 \not\approx^n_{TS} s_2$. This can be seen as follows. Assume there is normed bisimulation $(\mathcal{R}, \nu_1, \nu_2)$ for TS with $(s_1, s_2) \in \mathcal{R}$. Neither case (N1) nor (N3) applies to $s_1 \to s_0$ since $L(s_3) \neq L(s_0)$, and thus $s_3 \not\approx^n_{TS} s_0$. In order for case (N2) to apply to $s_1 \to s_0$ and $s_2 \to s_3$, we should have

$$s_0 \approx^n_{TS} s_2 \quad \text{and } \nu_1(s_0, s_2) < \nu_1(s_1, s_2).$$

But for $s_0 \approx^n_{TS} s_2$, neither case (N1) nor (N3) applies to $s_0 \to s_1$, as s_2 can only move to a state that is not labeled with $\{ a \}$. Thus, case (N2) applies to $s_0 \to s_1$. This yields

$$\nu_1(s_1, s_2) < \nu_1(s_0, s_2)$$

and contradicts the condition $\nu_1(s_0, s_2) < \nu_1(s_1, s_2)$. ■

It is not difficult to prove that $\approx^n_{TS}$ is an equivalence. Moreover, $\approx^n_{TS}$ can be equipped with norm functions ν_1, ν_2 such that $(\approx^n_{TS}, \nu_1, \nu_2)$ is a normed bisimulation; see Exercise 7.24.

Lemma 7.122. $\approx^n_{TS}$ ***is Finer Than*** $\approx^{div}_{TS}$

If s_1, s_2 are states in a transition system TS, then $s_1 \approx^n_{TS} s_2$ implies $s_1 \approx^{div}_{TS} s_2$.

Proof: Let $(\mathcal{R}, \nu_1, \nu_2)$ be a normed bisimulation. We show that the symmetric, reflexive, and transitive closure of $\mathcal{R}$—the coarsest equivalence that contains $\mathcal{R}$—is a divergence-sensitive stutter bisimulation. This amounts showing that $\mathcal{R}$ fulfills the conditions of a stutter bisimulation and that $\mathcal{R}$ is divergence-sensitive.

1. As normed bisimulation requires states to be equally labeled, it suffices to first prove condition (2) of a stutter bisimulation (see Definition 7.95 on page 536).

   Let $(s_1, s_2) \in \mathcal{R}$ and $s_1' \in \mathit{Post}(s_1)$ with $(s_1', s_2) \notin \mathcal{R}$. It is to be proven that there is a path fragment $s_2\, u_1 \ldots u_n\, s_2'$ with $(s_1', s_2') \in \mathcal{R}$ and $(s_2, u_j) \in \mathcal{R}$ for $j \geqslant 0$. By (NII), one of the conditions (N1), (N2) or (N3) holds. The assumption $(s_1', s_2) \notin \mathcal{R}$ rules out case (N2). If (N1) holds then the claim directly follows. Assume case (N3) applies. Then there exists $u_1 \in \mathit{Post}(s_2)$ with

   $$(s_1, u_1) \in \mathcal{R} \text{ and } \nu_2(s_1, u_1) < \nu_2(s_1, s_2).$$

   For state u_1 the same argument as for s_2 applies: only the cases (N1) and (N3) could apply. If (N1) holds, then there exists $s_2' \in \mathit{Post}(u_1)$ with $(s_1', s_2') \in \mathcal{R}$ and it follows that $s_2\, u_1\, s_2'$ is a path that satisfies the necessary requirements. Assume case (N3) applies. Then there exists $u_2 \in \mathit{Post}(u_1)$ with

   $$(s_1, u_2) \in \mathcal{R} \text{ and } \nu_2(s_1, u_2) < \nu_2(s_1, u_1).$$

   The reasoning of u_1 can now be applied to u_2, and if applicable u_3, u_4, and so on: if (N1) applies, the claim directly follows; otherwise (N3) applies. Assume that in this way we obtain a path fragment $s_2\, u_1\, u_2 \ldots u_n$ with $(s_1, u_i) \in \mathcal{R}$, for $0 < i \leqslant n$ and

   $$0 \leqslant \nu_2(s_1, u_n) < \nu_2(s_1, u_{n-1}) < \ldots < \nu_2(s_1, u_1) < \nu_2(s_1, s_2).$$

   Since the values of ν_2 are natural numbers, case (N3) can only apply finitely many times. Assume (N3) does not apply to (s_1, u_n). Then (N1) holds for (s_1, u_n) and transition $s_1 \to s_1'$ for some $n \leqslant \nu_2(s_1, s_2)$. Hence, there exists a transition $u_n \to s_2'$ with $(s_1', s_2') \in \mathcal{R}$. This yields a path fragment $s_2\, u_1\, u_2 \ldots u_n\, s_2'$ with $(s_1, u_i) \in \mathcal{R}$, and as $\mathcal{R}$ is an equivalence with $(s_1, s_2) \in \mathcal{R}$, we have

$$(s_2, u_i) \in \mathcal{R} \text{ for } 0 < i \leqslant n \text{ and } (s_1', s_2') \in \mathcal{R},$$

as required in condition (2) of stutter bisimulations.

2. The divergence sensitivity of $\mathcal{R}$ is shown as follows. Let $(s_1, s_2) \in \mathcal{R}$ and assume s_1 is $\mathcal{R}$-divergent. We have to prove that s_2 is $\mathcal{R}$-divergent. This is done by showing the existence of a transition $s_2 \to v_1$ such that $(s_2, v_1) \in \mathcal{R}$. We then may repeat this argument for (s_1, v_1). This yields the existence of a transition $v_1 \to v_2$ with $(s_1, v_2) \in \mathcal{R}$. By repeated application, we obtain an infinite path $s_2\, v_1\, v_2\, v_3 \ldots$ with $(s_1, v_i) \in \mathcal{R}$ for all $i > 0$. Thus s_2 is $\mathcal{R}$-divergent.

   Let us now consider the proof. Since s_1 is $\mathcal{R}$-divergent, there is an infinite path $u_0\, u_1\, u_2\, u_3 \ldots$ starting in $s_1 = u_0$ for which $(s_1, u_0) \in \mathcal{R}$ for all $i \geqslant 0$. If case (N2) applies to $(u_i, s_2) \in \mathcal{R}$ and transitions $u_i \to u_{i+1}$, $i = 0, 1, \ldots, k$, then

$$\nu_1(s_1, s_2) = \nu_1(u_0, s_2) > \nu_1(u_1, s_2) > \nu_1(u_2, s_2) > \ldots > \nu_1(u_k, s_2) \geqslant 0.$$

   Let $k_1 \geqslant 0$ be the smallest index such that case (N2) does not apply to $(u_{k_1}, s_2) \in \mathcal{R}$ and transition $u_{k_1} \to u_{k_1+1}$. If case (N1) holds, then there exists a transition $s_2 \to v_1$ where $(u_{k_1+1}, v_1) \in \mathcal{R}$ and hence $(s_2, v_1) \in \mathcal{R}$. If case (N3) holds for $(u_{k_1}, s_2) \in \mathcal{R}$ and $u_{k_1} \to u_{k_1+1}$, then there exists $v_1 \in \mathit{Post}(s_2)$ with $(u_{k_1}, v_1) \in \mathcal{R}$ and $\nu_2(u_{k_1}, v_1) < \nu_2(u_{k_1}, s_2)$. Thus, $s_2 \to v_1$ and $(s_2, v_1) \in \mathcal{R}$.

■

Lemma 7.122 can be rewritten for pairs of transition systems as follows:

Corollary 7.123. Normed and Stutter Bisimulation with Divergence

For all transition systems TS_1 and TS_2 over AP, we have:

$$TS_1 \approx^{n} TS_2 \text{ implies } TS_1 \approx^{div} TS_2.$$

The reverse of Corollary 7.123 does not hold; see Exercise 7.25.

To obtain a sufficient and necessary local norm function criterion for stutter bisimulation equivalence with divergence, a more complex notion of norm functions is needed which also depends on the transitions to be simulated. We concentrate here on the case of finite transition systems where the natural numbers can serve as a range for the norm functions. (To treat transition systems with infinitely many states, one has to deal with well-founded sets of arbitrary cardinality.)

Definition 7.124. Step-Dependent Normed Bisimulation

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system. A *step-dependent normed bisimulation* for TS is a pair $(\mathcal{R}, \nu)$ where $\mathcal{R}$ is an equivalence on S and $\nu : S \times S \times S \rightarrow \mathbb{N}$ a function such that for all $(s_1, s_2) \in \mathcal{R}$:

(SI) $L(s_1) = L(s_2)$.

(SII) For all $s_1' \in Post(s_1)$, at least one of the following three conditions holds:

- (S1) There exists $s_2' \in Post(s_2)$ with $(s_1', s_2') \in \mathcal{R}$.
- (S2) $(s_1', s_2) \in \mathcal{R}$ and $\nu(s_1', s_1', s_2) < \nu(s_1, s_1, s_2)$.
- (S3) There exists $s_2' \in Post(s_2)$ with $(s_1, s_2') \in \mathcal{R}$ and $\nu(s_1', s_1, s_2') < \nu(s_1', s_1, s_2)$.

States s_1 and s_2 are *step-dependent normed bisimilar*, denoted $s_1 \approx^s_{TS} s_2$, if there exists a step-dependent normed bisimulation $(\mathcal{R}, \nu)$ for TS such that $(s_1, s_2) \in \mathcal{R}$. ■

For $\nu(s_1', s_1, s_2)$, the last two arguments denote a pair $(s_1, s_2) \in \mathcal{R}$ while the s_1' stands either for a direct successor of s_1 or s_1 itself. If $s_1' \in Post(s_1)$, $\nu(s_1', s_1, s_2)$ denotes an upper bound for the number of stutter steps $s_2\, u_1 \ldots u_n$ inside $[s_2]_{\mathcal{R}}$ that s_2 may perform before taking a matching transition $u_n \rightarrow s_2'$ with $(s_1', s_2') \in \mathcal{R}$. In case $s_1' = s_1$, $\nu(s_1, s_1, s_2)$ is an upper bound for the stutter steps $s_1\, u_1 \ldots u_n$ inside $[s_1]_{\mathcal{R}}$ that cannot be matched by s_2.

Example 7.125. Step-Dependent Bisimulation Equivalence

For the transition system TS shown in Figure 7.45, we have $s_1 \approx^s_{TS} s_2$. This is justified by establishing a step-dependent normed bisimulation for TS that contains (s_1, s_2).

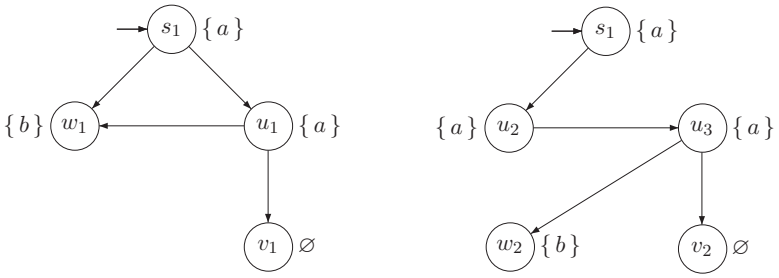


Figure 7.45: $s_1 \approx^s_{TS} s_2$.

Let $\mathcal{R}$ be the equivalence that identifies states s_1, u_1, u_2, u_3, s_2, states t_1, t_2, and states

w_1, w_2. Let the step-dependent norm function $\nu : S^3 \to \mathbb{N}$ be defined as follows:

$$\begin{array}{lcllcl}
\nu(s_1, s_1, s_2) & = & 2 & \nu(u_1, u_1, s_2) & = & 1 \\
\nu(w_1, s_1, s_2) & = & 2 & \nu(w_1, s_1, u_2) & = & 1 \\
\nu(w_1, u_1, s_2) & = & 2 & \nu(w_1, u_1, u_2) & = & 1 \\
\nu(t_1, u_1, s_2) & = & 2 & \nu(t_1, u_1, u_2) & = & 1 \\
\nu(s_2, s_2, s_1) & = & 2 & \nu(u_2, u_2, s_1) & = & 1 \\
\nu(u_3, u_3, s_1) & = & 1 & \nu(t_2, u_3, s_1) & = & 1
\end{array}$$

and $\nu(\cdot) = 0$ in all other cases. Then, $(\mathcal{R}, \nu)$ is a step-dependent normed bisimulation for *TS*. Let us check the required conditions. Condition (SI) is obvious since $\mathcal{R}$ identifies only equally labeled states. Consider the pair $(s_1, s_2) \in \mathcal{R}$ and check whether (SII) holds for the outgoing transitions of s_1.

- For the transition $s_1 \to w_1$, we consider the transition $s_2 \to u_2$ and get $(s_1, u_2) \in \mathcal{R}$ and $\nu(w_1, s_1, u_2) = 1 < 2 = \nu(w_1, s_1, s_2)$ Hence, condition (S3) is fulfilled.
- For the transition $s_1 \to u_1$, case (S2) applies as we have $(u_1, s_2) \in \mathcal{R}$ and $\nu(u_1, u_1, s_2) = 1 < 2 = \nu(s_1, s_1, s_2)$.

Let us now consider the pair (u_1, s_2). For the transition $u_1 \to v$ where $v \in \{w_1, t_1\}$ we consider the stutter step $s_2 \to u_2$ and get $\nu(x, u_1, u_2) = 1 < 2 = \nu(x, u_1, s_2)$, which yields condition (S3). Similarly, for the pair $(u_1, u_2) \in \mathcal{R}$ and the transitions $u_1 \to x$ case (S3) applies as we may take the transition $u_2 \to u_3$ and get

$$\nu(x, u_1, u_3) = 0 < 1 = \nu(x, u_1, u_2).$$

For (u_1, u_3) and transition $u_1 \to x$ condition (S1) is fulfilled as we may take the corresponding transition from u_3.

Let us now prove condition (SII) for the pair $(s_2, s_1) \in \mathcal{R}$. We have to regard the transition $s_2 \to u_2$. In fact, case (S2) applies since s_2 and u_2 are $\mathcal{R}$ equivalent and

$$\nu(s_2, s_2, s_1) = 2 > 1 = \nu(u_2, u_2, s_1).$$

Similarly, for the pair $(u_2, s_1) \in \mathcal{R}$ we have to consider the transition $u_2 \to u_3$ where again condition (S2) is fulfilled as we have

$$\nu(u_2, u_2, s_1) = 1 > 0 = \nu(u_3, u_3, s_1).$$

For $(u_3, s_1) \in \mathcal{R}$ and transition $u_3 \to w_2$ condition (S1) holds as we may move from s_1 to w_1. For the transition $u_3 \to t_2$ case (S3) applies for the stutter step $s_1 \to u_1$ as we have $\nu(t_2, u_3, s_1) = 1 > 0 = \nu(t_2, u_3, u_1)$. ■

The following theorem shows that step-dependent norm functions yield a sound and complete criterion for divergence stutter bisimilarity.

Theorem 7.126. Alternative Characterization of $\approx_{TS}^{div}$

Let TS be a finite transition system and s_1 *and* s_2 *states in TS. Then:*

$$s_1 \approx_{TS}^{s} s_2 \quad \textit{if and only if} \quad s_1 \approx_{TS}^{div} s_2.$$

Proof: We show that $\approx_{TS}^{s}$ is finer than $\approx_{TS}^{div}$, and vice versa.

Claim 1. $\approx_{TS}^{s}$ is finer than $\approx_{TS}^{div}$.

Proof of Claim 1. Similar arguments as in the proof of Lemma 7.122 can be used. In fact, for this direction the cardinality of *TS* is irrelevant. We take a step-dependent normed bisimulation $(\mathcal{R}, \nu)$ and show that $\mathcal{R}$ is a divergence-sensitive stutter bisimulation. Let us first check conditions (1) and (2) of stutter bisimulations. (Condition (3) is obtained from (2) by the symmetry of $\mathcal{R}$.) (1) is immediate from the definition of normed bisimulations (see condition (SI)). Let us check condition (2). Let $(s_1, s_2) \in \mathcal{R}$ and $s_1' \in \mathit{Post}(s_1)$ with $(s_1', s_2) \notin \mathcal{R}$. (SII) yields that one of the conditions (S1), (S2), or (S3) holds. (S2) is impossible as $(s_1', s_2) \notin \mathcal{R}$. If (S1) holds, then (2) is obvious. If (S3) applies, then we pick a state $u_1 \in \mathit{Post}(s_2)$ such that u_1 is $\mathcal{R}$-equivalent to s_1 and s_2 and

$$\nu(s_1', s_1, u_1) < \nu(s_1', s_1, s_2).$$

Since $(s_1, u_1) \in \mathcal{R}$ we may apply the same argument: either (S1) applies, i.e., there exists $s_2' \in \mathit{Post}(u_1)$ with $(s_1', s_2') \in \mathcal{R}$, or (S3) applies, which yields a state $u_2 \in \mathit{Post}(u_1)$ such that u_2 is $\mathcal{R}$-equivalent to s_1, s_2, u_1 and

$$\nu(s_1', s_1, u_2) < \nu(s_1', s_1, u_1).$$

Repeating this argument yields a path fragment $s_2\, u_1\, u_2 \ldots u_n$ such that $(s_1, u_i) \in \mathcal{R}$, $1 \leqslant i \leqslant n$ and

$$0 \leqslant \nu(s_1', s_1, u_n) < \nu(s_1', s_1, u_2) < \ldots < \nu(s_1', s_1, u_1) < \nu(s_1', s_1, s_2)$$

and such that (S3) does not apply to $(s_1, u_n) \in \mathcal{R}$ and the transition $s_1 \to s_1'$. Since s_1' is not $\mathcal{R}$-equivalent to s_1, (S1) applies which yields a transition $u_n \to s_2'$ with $(s_1', s_2') \in \mathcal{R}$. Hence, we have a path fragment $s_2\, u_1\, u_2 \ldots u_n\, s_2'$ with $(s_1, u_i) \in \mathcal{R}$ and $(s_1', s_2') \in \mathcal{R}$, as required in condition (2).

It remains to show the divergence sensitivity of $\mathcal{R}$. Let $(s_1, s_2) \in \mathcal{R}$ and let s_1 be $\mathcal{R}$-divergent. We have to prove the $\mathcal{R}$-divergency of s_2. For this, we show the existence

of a transition $s_2 \to v_1$ such that s_2 and v_1 are $\mathcal{R}$-equivalent. We then may repeat this argument for (s_1, v_1) which yields the existence of a transition $v_1 \to v_2$ with $(s_1, v_2) \in \mathcal{R}$ and the existence of a transition $v_2 \to v_3$ with $(s_1, v_3) \in \mathcal{R}$, and so on. In this way we obtain an infinite path $s_2\, v_1\, v_2\, v_3 \ldots$ with $(s_1, v_i) \in \mathcal{R}$ for all $i \geqslant 0$, and thus, the $\mathcal{R}$-divergency of s_2.

Since s_1 is $\mathcal{R}$-divergent there is an infinite path $u_0\, u_1\, u_2\, u_3 \ldots$ starting in $s_1 = u_0$ where s_1 and all states u_i are $\mathcal{R}$-equivalent. If case (S2) applies to the pair $(u_i, s_2) \in \mathcal{R}$ and transitions $u_i \to u_{i+1}$, $i = 0, 1, \ldots, k$, then

$$\nu(s_1, s_1, s_2) > \nu(u_1, u_1, s_2) > \nu(u_2, u_2, s_2) > \ldots > \nu(u_k, u_k, s_2) \geqslant 0.$$

Let $k_1 \geqslant 0$ be the smallest index such that case (S2) does not apply to the pair $(u_{k_1}, s_2) \in \mathcal{R}$ and transition $u_{k_1} \to u_{k_1+1}$. If case (S1) holds, then there is a transition $s_2 \to v_1$ where $(u_{k_1+1}, v_1) \in \mathcal{R}$. Since all u-states are $\mathcal{R}$-equivalent to s_1 and s_2, this yields $(s_2, v_1) \in \mathcal{R}$, and we are done. If case (S3) holds for $(u_{k_1}, s_2) \in \mathcal{R}$ and transition $u_{k_1} \to u_{k_1+1}$, then there exists $v_1 \in \mathit{Post}(s_2)$ with $(u_{k_1}, v_1) \in \mathcal{R}$ and $\nu(u_{k_1+1}, u_{k_1}, v_1) < \nu_2(u_{k_1+1}, u_{k_1}, s_2)$. Again, we have $s_2 \to v_1$ and the $\mathcal{R}$-equivalence of s_2 and v_1.

Claim 2. $\approx_{TS}^{div}$ is finer than $\approx_{TS}^{s}$, provided that TS is finite.

Proof of Claim 2. We establish a ternary norm function ν that satisfies the conditions in Definition 7.124 for $\mathcal{R} = \approx_{TS}^{div}$.

- For $s_1 \approx_{TS}^{div} s_2$ and $s_1' \in \mathit{Post}(s_1)$ with $s_1 \not\approx_{TS}^{div} s_1'$ we define $\nu(s_1', s_1, s_2)$ as the length of a *shortest* path fragment $s_2\, u_1\, u_2 \ldots u_n$ such that
  - $u_i \approx_{TS}^{div} s_2$, $i = 1, \ldots, n$
  - $[s_1']_{div} \cap \mathit{Post}(u_n) \neq \varnothing$, where $[s]_{div}$ denotes the equivalence class of s under $\approx_{TS}^{div}$.
- If $s_1 \approx_{TS}^{div} s_2$ and s_1, s_2 are $\approx_{TS}^{div}$-divergent then we put $\nu(s_1, s_1, s_2) = 0$.
- If $s_1 \approx_{TS}^{div} s_2$ and s_1, s_2 are not $\approx_{TS}^{div}$-divergent then we define $\nu(s_1, s_1, s_2)$ as the length of a *longest* path fragment $s_1\, v_1\, v_2 \ldots v_n$ such that

  $$v_i \approx_{TS}^{div} s_1 \text{ for } i = 1, \ldots, n \quad \text{and} \quad [s_1]_{div} \cap \mathit{Post}(v_n) = \varnothing.$$

  Note that such a longest finite path fragment exists since TS is finite and s_1 is not divergent.

We now show that $(\approx_{TS}^{div}, \nu)$ is a step-dependent normed bisimulation for TS. Obviously, $\approx_{TS}^{div}$ is an equivalence on S where (SI) holds. Let us check condition (SII) for states s_1, s_2 with $s_1 \approx_{TS}^{div} s_2$ and transition $s_1 \to s_1'$.

- If $s_1 \not\approx_{TS}^{div} s_1'$, then either $[s_1']_{div} \cap Post(s_2) \neq \varnothing$ (in which case condition (S1) holds) or there is a path fragment $s_2\, u_1 \ldots u_n s_2'$ with $n \geqslant 1$ and

$$u_i \approx_{TS}^{div} s_2,\ i = 1, \ldots, n \text{ and } s_1' \approx_{TS}^{div} s_2'.$$

  We may assume that n is minimal. Then, $s_2 \to u_1$ is a transition with $(u_1, s_2) \in \mathcal{R}$ and $n - 1 = \nu(s_1', s_1, u_1) < \nu(s_1', s_1, s_2) = n$. Hence, condition (S3) holds.

- Let us now consider a stutter step $s_1 \to s_1'$ inside the equivalence class of s_1, i.e., we suppose $s_1 \approx_{TS}^{div} s_1'$.
  - If s_1 is divergent, then so is s_2 and there is a step $s_2 \to s_2'$ where s_2' is $\approx_{TS}^{div}$ equivalent to s_1 and s_2. Thus, condition (S1) holds.
  - Let us now suppose that s_1 and s_2 are not divergent. For $s \approx_{TS}^{div} s_1$ let $\ell(s)$ be the length of a *longest* path fragment $s\, v_1\, v_2 \ldots v_n$ such that $v_i \approx_{TS}^{div} s$, $i = 1, \ldots, n$ and $[s]_{div} \cap Post(v_n) = \varnothing$. Clearly, we have

$$\ell(s) = \max_{s' \in Post(s) \cap [s]_{div}} \ell(s') + 1.$$

    $\nu(s, s, s_2) = \ell(s)$ (by definition of ν). Hence,

$$\nu(s_1, s_1, s_2) = \ell(s_1) \geqslant \ell(s_1') + 1 > \nu(s_1', s_1', s_2).$$

    This shows that condition (S2) holds.

■

7.8.3 Stutter Bisimulation and $\mathrm{CTL}^*_{\setminus\bigcirc}$ Equivalence

Stutter bisimulation ($\approx$) does not preserve $\mathrm{LTL}_{\setminus\bigcirc}$ formulae since it does not impose any restrictions on divergent paths. That is, if $TS_1 \approx TS_2$, and TS_1 contains a divergent path violating some $\mathrm{LTL}_{\setminus\bigcirc}$ formula φ, whereas TS_2 does not exhibit such behaviour, then $TS_1 \not\models \varphi$ whereas $TS_2 \models \varphi$. To avoid the complications induced by stutter paths, we have considered $\approx^{div}$ that requires two equivalent states to either both have a divergent path, or both have no divergent paths. The central result in this section is that $\approx^{div}$ coincides with the logical equivalences for the next-free fragments of CTL^* and CTL. This entails that any formula in these logics can be checked on a transition system that is stutter bisimulation-equivalent with divergence, in particular, the quotient under $\approx^{div}$. Moreover, to show $TS_1 \not\approx^{div} TS_2$ it suffices to provide a CTL^* formula Φ that does not contain $\bigcirc$, such that $TS_1 \models \Phi$ and $TS_2 \not\models \Phi$.

Notation 7.127. CTL* and CTL without Next Step

Let $\mathrm{CTL}^*_{\setminus\bigcirc}$ denote the class of all CTL* formulae that do not contain any occurrence of the next step operator $\bigcirc$. Similarly, $\mathrm{CTL}_{\setminus\bigcirc}$ denotes the sublogic of CTL which does not permit the next-step operator $\bigcirc$. ■

Theorem 7.128. $CTL^*_{\setminus\bigcirc}$/$CTL_{\setminus\bigcirc}$ Equivalence and $\approx^{div}$

For finite transition system TS without terminal states, and states s_1, s_2 in TS:

$$s_1 \approx^{div}_{TS} s_2 \quad \textit{iff} \quad s_1 \equiv_{CTL^*_{\setminus\bigcirc}} s_2 \quad \textit{iff} \quad s_1 \equiv_{CTL_{\setminus\bigcirc}} s_2.$$

Proof: The proof strategy is analogous to the proof of Theorem 7.20 (page 469).

1. Lemma 7.129 (see below) asserts that $\approx^{div}_{TS}$ is finer than $\mathrm{CTL}^*_{\setminus\bigcirc}$ equivalence for arbitrary (possibly infinite) transition systems.
2. $\mathrm{CTL}^*_{\setminus\bigcirc}$ equivalence is finer than $\mathrm{CTL}_{\setminus\bigcirc}$ equivalence, since $\mathrm{CTL}_{\setminus\bigcirc}$ is a sublogic of $\mathrm{CTL}^*_{\setminus\bigcirc}$.
3. Lemma 7.130 (see below) asserts that $\mathrm{CTL}_{\setminus\bigcirc}$ equivalence is finer than $\approx^{div}_{TS}$.

Note that the proof of (3) for CTL equivalence and bisimulation ($\sim$) exploits the next-step operator (and not the until operator). As this operator is absent in $\mathrm{CTL}_{\setminus\bigcirc}$, an alternative logical characterization has to be pursued. ■

Lemma 7.129. Divergence Stutter Bisimilarity Implies $CTL^*_{\setminus\bigcirc}$ Equivalence

Let TS be a transition system without terminal states, s_1, s_2 states in TS, and $\pi_1, \pi_2 \in Paths(TS)$. Then:

1. *if $s_1 \approx^{div}_{TS} s_2$, then for all $CTL^*_{\setminus\bigcirc}$ state formulae Φ: $s_1 \models \Phi$ iff $s_2 \models \Phi$.*
2. *if $\pi_1 \approx^{div}_{TS} \pi_2$, then for all $CTL^*_{\setminus\bigcirc}$ path formulae φ: $\pi_1 \models \varphi$ iff $\pi_2 \models \varphi$.*

Proof: The proof is provided by structural induction over the $\mathrm{CTL}^*_{\setminus\bigcirc}$ formulae and is similar to the proof of Lemma 7.26 (page 473). We omit the details and only consider the case $\Phi = \exists\varphi$, where φ is a $\mathrm{CTL}^*_{\setminus\bigcirc}$ path formula satisfying claim 2. Assume $s_1 \approx^{div}_{TS} s_2$

and $s_1 \exists\varphi$. Let $\pi_1 \in \mathit{Paths}(s_1)$ such that $\pi_1 \models \varphi$. According to Lemma 7.118 (page 549), there exists $\pi_2 \in \mathit{Paths}(s_2)$ with $\pi_1 \approx_{TS}^{div} \pi_2$. Applying the induction hypothesis to φ and paths π_1, π_2 yields $\pi_2 \models \varphi$. Hence, $s_2 \models \exists\varphi$. ■

Lemma 7.130. $CTL_{\setminus\bigcirc}$ Equivalence is Finer Than $\approx_{TS}^{div}$

For finite transition system TS without terminal states, and s_1, s_2 states in TS:

$$s_1 \text{ and } s_2 \text{ are } CTL_{\setminus\bigcirc} \text{ equivalent implies } s_1 \approx_{TS}^{div} s_2.$$

Proof: Let S be the state space of *TS*. It suffices to show that

$$\mathcal{R} \;=\; \{\, (s_1, s_2) \in S \times S \mid s_1 \equiv_{CTL_{\setminus\bigcirc}} s_2 \,\}$$

is a divergence-sensitive stutter bisimulation for *TS*. We show that (1) $\mathcal{R}$ is a stutter bisimulation for *TS* and (2) $\mathcal{R}$ is divergence-sensitive.

Claim 1. $\mathcal{R}$ is a stutter bisimulation for *TS*.

Proof of Claim 1. This is proven by checking the conditions of stutter bisimulation. Let $(s_1, s_2) \in \mathcal{R}$.

1. Consider the propositional logic formula

$$\Phi \;=\; \bigwedge_{a \in L(s_1)} a \;\wedge\; \bigwedge_{a \in AP \setminus L(s_1)} \neg a.$$

   Clearly, Φ is a $\mathrm{CTL}_{\setminus\bigcirc}$ formula with $s_1 \models \Phi$. Hence, by definition of $\mathcal{R}$, $s_2 \models \Phi$. By construction of Φ, we have $L(s_1) = L(s_2)$.

2. For each equivalence class $C \in S/\mathcal{R}$, let $\mathrm{CTL}_{\setminus\bigcirc}$ formula Φ_C be defined by

$$\Phi_C \;=\; \bigwedge_{\substack{D \in S/\mathcal{R} \\ D \neq C}} \Phi_{C,D}$$

   where $\Phi_{C,D}$ is a $\mathrm{CTL}_{\setminus\bigcirc}$ formula such that $\mathit{Sat}(\Phi_{C,D}) \supseteq C$ and $\mathit{Sat}(\Phi_{C,D}) \cap D = \varnothing$, for each pair (C, D) of equivalence classes C, D under $\mathcal{R}$ with $C \neq D$. This yields $\mathit{Sat}(\Phi_C) = C$.

   It remains to prove that for all $s_1' \in \mathit{Post}(s_1)$ with $(s_1, s_1') \notin \mathcal{R}$, there exists a finite path fragment $s_2\, u_1 \ldots u_n\, s_2'$ with $(s_1', s_2') \in \mathcal{R}$ and $(s_2, u_i) \in \mathcal{R}$, $i = 1, \ldots, n$. Let

$B \in S/\mathcal{R}$ and $s_1, s_2 \in B$, i.e., $B = [s_1]_\mathcal{R} = [s_2]_\mathcal{R}$. Further, let $s_1' \in \mathit{Post}(s_1)$ such that $(s_1, s_1') \notin \mathcal{R}$, say $C = [s_1']_\mathcal{R}$ with $B \neq C$. Then:

$$s_1 \models \exists(\Phi_B \,\mathsf{U}\, \Phi_C) \quad \text{as } s_1 \in B \text{ and } s_1' \in C$$

Since $(s_1, s_2) \in \mathcal{R}$ and $\Phi_B \,\mathsf{U}\, \Phi_C$ is a $\mathrm{CTL}_{\setminus\bigcirc}$ formula, we get

$$s_2 \models \exists(\Phi_B \,\mathsf{U}\, \Phi_C).$$

Thus, there exists a finite path fragment $s_2\, u_1 \ldots u_n\, s_2'$ with $s_2' \models \Phi_C$ and $s_2 \models \Phi_B$, $u_1 \models \Phi_B, \ldots, u_n \models \Phi_B$. Since $\mathit{Sat}(\Phi_C) = C$ and $\mathit{Sat}(\Phi_B) = B$, we obtain $s_2' \in C$ and $u_1, \ldots, u_n \in B$. As $C = [s_1']_\mathcal{R}$ and $B = [s_1]_\mathcal{R} = [s_2]_\mathcal{R}$, we conclude that $(s_1', s_2') \in \mathcal{R}$ and $(s_2, u_i) \in \mathcal{R}$, for $i = 1, \ldots, n$.

Claim 2. $\mathcal{R}$ is divergence-sensitive.

Proof of Claim 2. Assume $(s_1, s_2) \in \mathcal{R}$ and s_1 is $\mathcal{R}$-divergent. We have to show that s_2 is $\mathcal{R}$-divergent. As s_1 is $\mathcal{R}$-divergent there exists an $\mathcal{R}$-divergent path fragment:

$$\pi_1 \;=\; s_1\, s_{1,1}\, s_{2,1}\, s_{3,1} \ldots \in \mathit{Paths}(s_1).$$

Let $C = [s_1]_\mathcal{R}$ and Φ_C be a $\mathrm{CTL}_{\setminus\bigcirc}$ formula such that $\mathit{Sat}(\Phi_C) = C$. Then $s_{j,1} \in C$, i.e., $s_{j,1} \models \Phi_C$ for all $j > 0$. Hence, $s_1 \models \exists\Box\Phi_C$. As $(s_1, s_2) \in \mathcal{R}$ and $C = [s_1]_\mathcal{R}$, we get $s_2 \models \exists\Box\Phi_C$. Thus, there exists an infinite path fragment:

$$\pi_2 \;=\; s_2\, s_{1,2}\, s_{2,2}\, s_{3,2} \ldots \in \mathit{Paths}(s_2)$$

with $s_{j,2} \models \Phi_C$ for $j > 0$. As $\mathit{Sat}(\Phi_C) = C$, we get $s_{j,2} \in C$ for $j > 0$. Since $C = [s_2]_\mathcal{R}$, π_2 is an $\mathcal{R}$-divergent path. Hence, s_2 is $\mathcal{R}$-divergent. ∎

A few remarks are in order. $\approx_{TS}^{div}$ is the coarsest equivalence which preserves all $\mathrm{CTL}_{\setminus\bigcirc}$ formulae. Theorem 7.128 can be reformulated for pairs of transition systems in the usual way. Thus, for all finite transition systems TS_1, TS_2 without terminal states:

$$\begin{aligned} TS_1 \approx^{div} TS_2 \quad & \text{iff} \quad TS_1 \text{ and } TS_2 \text{ satisfy the same } \mathrm{CTL}^*_{\setminus\bigcirc} \text{ formulae} \\ & \text{iff} \quad TS_1 \text{ and } TS_2 \text{ satisfy the same } \mathrm{CTL}_{\setminus\bigcirc} \text{ formulae.} \end{aligned}$$

We finally remark that in the proof of the logical characterization of $\approx_{TS}^{div}$, the until operator is explicitly used. Thus, whereas bisimulation and simulation can be characterized by a CTL (or CTL*) fragment that does not contain the until operator, this is not the case for $\approx_{TS}^{div}$. This is not surprising, as $\approx_{TS}^{div}$ does not preserve the next-step operator and some modal operator is needed to characterize finite stutter path fragments.

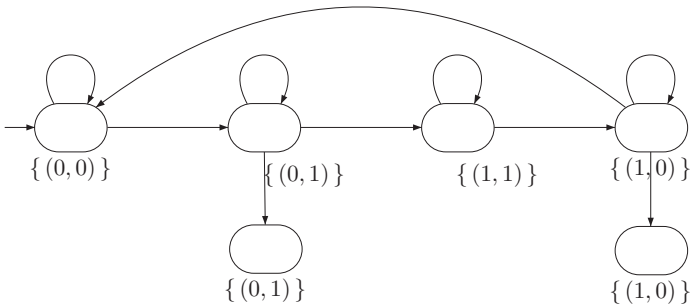


Figure 7.46: $TS_{ABP}/\approx^{div}$.

In particular, since $TS \approx^{div} TS/\approx^{div}$, it suffices to check $\mathrm{CTL}^*_{\setminus\bigcirc}$ formula Φ on $TS/\approx^{div}$ to decide whether $TS \models \Phi$. This is exemplified by the alternating bit protocol.

Example 7.131. Alternating Bit Protocol

Consider again the alternating bit protocol; see Example 2.32 on page 57. Let

$$AP = \{\, s\_mode = 0, s\_mode = 1, r\_mode = 0, r\_mode = 1 \,\}$$

where *s_mode* indicates the mode of the sender, and *r_mode* that of the receiver. In Example 7.108 (page 545), it was concluded that TS_{ABP}, the transition system underlying the ABP, and its stutter bisimulation quotient $TS_{ABP}/\approx$, do not satisfy the same $\mathrm{CTL}^*_{\setminus\bigcirc}$ formulae. This is due to the fact that the transition system

$$TS \;=\; (TS_{ABP} \;\oplus\; TS_{ABP})\,/\approx$$

is *not* divergence-sensitive, i.e., there are stutter-bisimilar states in TS such that one of them is $\approx$-divergent, while the other is not. In fact, in order to prove a $\mathrm{CTL}^*_{\setminus\bigcirc}$, $\mathrm{CTL}_{\setminus\bigcirc}$, or $\mathrm{LTL}_{\setminus\bigcirc}$ formula of the ABP, we must consider $TS_{ABP}/\approx^{div}$ instead of $TS_{ABP}/\approx$.

To obtain the quotient under $\approx^{div}$, each state in $TS_{ABP}/\approx$ for which $s\_mode \neq r\_mode$ needs to be decomposed in two states. Accordingly, the transition system $TS_{ABP}/\approx^{div}$ is obtained as depicted in Figure 7.46. Two of those states are labeled with

$$A_1 \;=\; \{s\_mode = 0, r\_mode = 1\}, \quad A_2 \;=\; \{s\_mode = 1, r\_mode = 0\}.$$

One of these two states represents the divergent A_i-states in TS_{ABP}, while the other represents the nondivergent A_i-states $\langle chk\_ack(i), \ldots, x = i, \ldots\rangle$ in TS_{ABP}. The $\mathrm{CTL}^*_{\setminus\bigcirc}$ safety property

$$\Phi \;=\; \forall\Box \;\Big(\; (s\_mode = 0) \;\longrightarrow\; (s\_mode = 0)\,\mathsf{W}\,(r\_mode = 1) \;\Big)$$

expresses that the sender does not leave the 0-mode (in which it only transmits message with bit 0) before the receiver changes to the 1-mode (in which it expects to receive messages with bit 1, i.e., before the receiver has acknowledged the correct receipt of the message with bit 0. It is easy to establish that $TS_{ABP}/{\approx^{div}} \models \Phi$. Theorem 7.128 yields $TS_{ABP} \models \Phi$. ■

Example 7.132. Producer-Consumer System

Consider a concurrent program that involves a producer and consumer process that shares a buffer of data items of capacity $m > 0$. The producer repeatedly produces n items and inserts these data items one by one in the buffer. Local variable *in* indicates the next buffer cell that needs to be written. The producer can only insert an element in cell *buffer* at index *in* when this cell is empty. An empty buffer cell is indicated by $\perp$. The consumer process successively attempts to get items from the buffer.

```
Producer
    in := 0;
    while true {
      produce d_1, ..., d_n;
      for i = 1 to n {
      wait until (buffer[in] = ⊥) {
        buffer[in] := d_i;
        in := (in + 1) mod m; }
      }
    }
```

```
Consumer
    out := 0;
    while true {
      for j = 1 to n {
      wait until (buffer[out] ≠ ⊥) {
        e_j := buffer[out];
        buffer[out] := ⊥;
        out := (out + 1) mod m; }
      }
      consume e_1, ..., e_n
    }
```

Let $TS(m, n)$ denote the transition system underlying this producer-consumer program. The size of $TS(n, m)$ grows exponentially in the number n of data items generated per cycle and in the buffer capacity m.

Suppose we are interested in the situation in which the producer and the consumer are simultaneously in their produce and consume phase, respectively. Let $AP = \{ \mathit{prod\_and\_cons} \}$. Observe that the content of the buffer can be ignored. This also applies to the variables *in* and *out*. In order to keep track of the number of free buffer cells, we introduce the bounded integer variable *free* with domain $\{ 0, 1, \ldots, m \}$. On completion of the production phase, variable i indicates the number of data items the producer has already stored in the buffer. Accordingly, j stands for the number of data items the consumer has withdrawn from the buffer. This yields the following abstract programs:

```
Producer
    while true {
      produce;
      for i = 1 to n {
      wait until (free > 0) {
        free := free − 1;
      }
    }
```

```
Consumer
    while true {
      for j = 1 to n {
      wait until (free < m) {
        free := free + 1;
      }
      consume
    }
```

Let $TS_{abstract}(m,n)$ denote the transition system that is obtained for this abstract program. $TS_{abstract}(2,2)$ is indicated in Figure 7.47, where it is assumed that executing the body of the for loop is executed atomically. Each state is of the form $\langle \ell_i, \ell'_i, v_f, v_i, v_j \rangle$ where ℓ_i and ℓ'_i indicate the program locations of the producer and consumer, respectively; v_f is the value of *free*, and v_i, v_j the value of integers i and j. In particular, we have:

$$\begin{array}{lcl} \ell_0 & : & \mathit{produce} \\ \ell_1 & : & \langle \textbf{if } (\mathit{free} > 0) \textbf{ then } i := 1; \mathit{free}{-}{-} \textbf{ fi} \rangle \\ \ell_2 & : & \langle \textbf{if } (\mathit{free} > 0) \textbf{ then } i := 0; \mathit{free}{-}{-} \textbf{ fi} \rangle \end{array}$$

The labels ℓ'_i are defined for the consumer process in a similar way. Note that the states with control locations ℓ_0 and ℓ'_2 indicate the global states in which the producer and consumer are in their produce and consume phase, respectively. Thus, all states of the form $\langle 0, 2, -, -, - \rangle$ are labeled with the proposition *prod_and_cons*; all other states are labeled with $\varnothing$. The size of $TS_{abstract}(n,m)$ is polynomial in n and m. It follows that

$$TS(m,n) \;\approx^{div}\; TS_{abstract}(m,n)\ .$$

Observe that

$$TS_{abstract}(m,n) \;\not\models\; \forall\Box\forall\Diamond \mathit{prod\_and\_cons}\ .$$

As this is a $\mathrm{CTL}_{\setminus\bigcirc}$ formula, it follows that $TS(m,n)$ refutes it. A counterexample in $TS_{abstract}(m,n)$ is, e.g.,

$$\begin{array}{l} \langle \ell_0, \ell'_0, 2, 0, 0 \rangle \to \langle \ell_1, \ell'_0, 2, 0, 0 \rangle \to \langle \ell_2, \ell'_0, 1, 1, 0 \rangle \to \langle \ell_0, \ell'_0, 0, 0, 0 \rangle \to \\ \langle \ell_0, \ell'_1, 1, 0, 1 \rangle \to \langle \ell_1, \ell'_1, 1, 0, 1 \rangle \to \langle \ell_0, \ell'_2, 2, 0, 0 \rangle \to \langle \ell_0, \ell'_0, 2, 0, 0 \rangle \to \ldots \end{array}$$

The fact that $TS_{abstract}(n,m) \not\models \forall\Box\forall\Diamond \mathit{prod\_and\_cons}$ can easily be shown by considering its quotient under $\approx^{div}$; see the self-loop in the initial states in Figure 7.48.

■

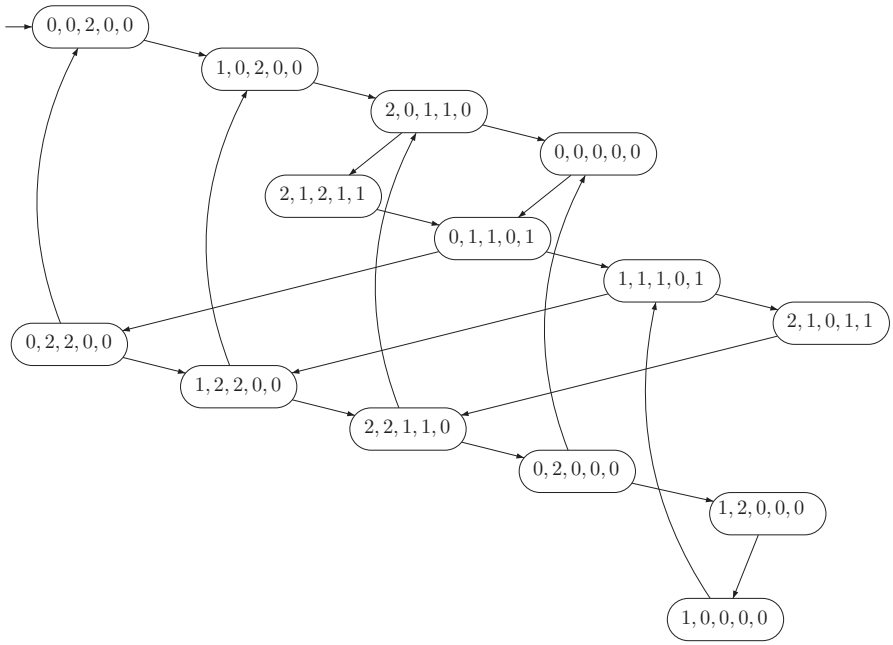


Figure 7.47: Transition system $TS_{abstract}(2, 2)$.

7.8.4 Stutter Bisimulation Quotienting

This section is concerned with algorithms to compute the quotient transition system under the stutter bisimulations $\approx$ and $\approx^{div}$. Such algorithms serve the following two purposes. First, they can be used to check whether two transition systems are stutter bisimulation equivalent. Secondly, the algorithm for $\approx^{div}$ can be used to minimize a transition system with respect to divergence-sensitive stutter bisimulation. This can be used as a preprocessing phase prior to model checking a $\mathrm{CTL}^*_{\setminus\bigcirc}$ formula. We first outline how to compute the equivalence classes under $\approx$ (i.e., without divergence) and then explain how this algorithm can be adapted to $\approx^{div}$.

In the remainder of this section, let $TS = (S, Act, \rightarrow, I, AP, L)$ be a finite transition system, possibly with terminal states. The quotienting algorithms for $\approx$ and $\approx^{div}$ are based on partition refinement, as for bisimulation. Recall that partition refinement algorithms are based on successively refining partitions until a stable partition is reached. The refinement is based on splitting a block in the current partition with respect to a superblock, i.e., a disjoint union of blocks. The idea of refinement for $\approx$ is to split a given block B according to another block C into two subblocks: a block of states in B that can reach C via a path fragment that only visits states in B, and a block that contains all other states in B.

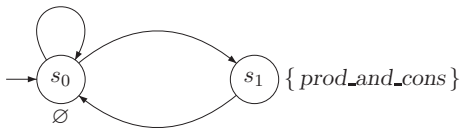


Figure 7.48: Quotient transition system $TS_{abstract}(2,2)/\approx^{div}$.

Notation 7.133. Constrained Predecessors

For partition Π of S and blocks B, C in Π with $s \in B$, we have $s \in Pre^*_\Pi(C)$ whenever there exists a finite path fragment:

$$s = s_1\, s_2 \ldots s_{n-1}\, s_n \in Paths(s) \text{ with } s_n \in C \text{ and } s_i \in B \text{ for all } 0 < i < n.$$

■

Stated in words, $Pre^*_\Pi(C)$ contains any state s that can reach a state in C via a (possibly single-state) path that solely consists of states that are in the same block of Π as s.

Definition 7.134. Π-Splitter, C-Stability

Let Π be a partition of S and let C, $B \in \Pi$.

1. C is a Π-*splitter* for B if and only if

$$B \neq C \quad \text{and} \quad B \cap Pre(C) \neq \varnothing \quad \text{and} \quad B \setminus Pre^*_\Pi(C) \neq \varnothing.$$

2. Π is C-*stable* if there is no block $B \in \Pi$ such that C is a Π-splitter for B.
3. Π is *stable* if Π is C-stable for all blocks $C \in \Pi$.

■

Let us consider the requirements of C being a Π-splitter of B. As a block cannot be split by itself, the requirement that $B \neq C$ is natural. Furthermore, B can only be splitted according to C when C can be reached in a single transition from at least one state in B. Finally, the third conjunct asserts that B should contain states that can only reach C via some state that is neither in B nor in C.

Algorithm 37 Computing the stutter bisimulation quotient

Input: finite transition system TS with state space S
Output: stutter bisimulation quotient space $S/\approx$

$\Pi := \Pi_{AP}$; (* see Algorithm 29, page 479 *)
while $(\exists B, C \in \Pi.\ C$ is a Π-splitter for $B)$ **do**
  choose such $B, C \in \Pi$;
  $\Pi := (\Pi \setminus \{ B \}) \cup \mathit{Refine}_{\approx}(B, C)$
od
return Π

Lemma 7.135. Coarsest Partition

The stutter bisimulation quotient space $S/\approx$ is the coarsest partition Π of S such that

(i) Π *is finer than* Π_{AP}.

(ii) for all $B, C \in \Pi$: $B \cap \mathit{Pre}(C) = \varnothing$ *or* $B \subseteq \mathit{Pre}^*_\Pi(C)$.

Proof: Similar to the proof of Lemma 7.34 on page 480. ■

That is, $S/\approx$ is the coarsest partition such that (i) states are equally labeled, and (ii) either there is no transition between B and C, or in case such transition exists, there are some states in B that cannot reach C by only visiting states in B. For such states, the only possibility to reach C is via some other block $D \neq B, C$. This lemma suggests the following refinement operator:

Definition 7.136. The Refinement Operator

Let Π be a partition for S and C be a block of Π. Then:

$$\mathit{Refine}_{\approx}(\Pi, C) \;=\; \bigcup_{B \in \Pi} \mathit{Refine}_{\approx}(B, C)$$

where $\mathit{Refine}_{\approx}(B, C) \;=\; \{B \cap \mathit{Pre}^*_\Pi(C),\ B \setminus \mathit{Pre}^*_\Pi(C)\} \setminus \{ \varnothing \}$. ■

The essential steps of the partition refinement technique for quotienting with respect to $\approx$ are summarized in Algorithm 37. Note that unlike the partition refinement algorithms for ordinary bisimulation $\sim$, Algorithm 37 refines per iteration just one block B. As C is a Π-splitter for B, each iteration yields a proper refinement. Therefore, the number of

iterations of the while-loop is bounded by the number of states. In the remainder of this section, some implementation details will be provided to enable an efficient realization of the refinement operator.

Removal of Stutter Cycles As a first step, the transition system TS is simplified by removing all its *stutter cycles*, i.e., cycles that completely consist of stutter steps. Formally:

Definition 7.137. Stutter Cycle

Cycle $s_0\, s_1\, s_2\, \ldots\, s_n$ (with $s_0 = s_n$) for $n > 0$ in transition system TS is a *stutter cycle* when $s_i\, s_{i+1}$ is a stutter step for $0 \leqslant i < n$. ■

All states on a stutter cycle belong to the same equivalence class under $\approx$, and thus belong to the same block of $S/\!\approx$. In fact, this even holds for $\approx^{div}$:

Lemma 7.138. Stutter Cycle

For stutter cycle $s_0\, s_1\, s_2\, \ldots\, s_n$ in transition system TS:

$$s_0 \approx^{div}_{TS} s_1 \approx^{div}_{TS} \ \ldots\ \approx^{div}_{TS} s_n.$$

Proof: Let $\mathcal{R}$ be the smallest equivalence relation on S which identifies the states $s_0 = s_n, s_1, \ldots, s_{n-1}$ and no other states. That is, $\mathcal{R}$ is the equivalence relation that induces the partition:

$$S/\mathcal{R} \;=\; \{\{\, s_0, \ldots, s_{n-1} \,\}\} \cup \{\{\, s \,\} \mid s \in S \setminus \{\, s_0, \ldots, s_{n-1} \,\}\}.$$

All states in $\{\, s_0, \ldots, s_{n-1} \,\}$ are stutter-bisimilar as they all lie on the same stutter cycle, i.e., any visible step by $s_i \to s'$ can be matched by a path fragment starting in s_j ($i{\neq}j$) by first traversing the cycle to s_i and then taking the transition $s_i \to s'$. Clearly, these states are all divergent. All singleton partitions are stutter-bisimilar and nondivergent. Hence, $\mathcal{R}$ is a divergence-sensitive stutter bisimulation. ■

Consequently, all states on a stutter cycle are divergent. Note that for finite transition systems it holds that any divergent path must contain a cycle. Since all states on such cycle are pairwise $\approx^{div}$ equivalent—and thus equally labeled—they form a stutter cycle.

Corollary 7.139. Complete Characterization of Divergent States

For finite transition system TS and state s in TS:

$$s \text{ is } \approx^{div}\text{-divergent} \quad \textit{iff} \quad \begin{array}{l}\textit{a stutter cycle is reachable from } s \\ \textit{via a path fragment in } [s]_{\approx^{div}}.\end{array}$$

To simplify the refinement operator, we first remove stutter cycles from transition system TS. This is done in the following way. In the state graph $G(TS)$, determine the strongly connected components (SCCs) that only contain stutter steps. This can be carried out using a (standard) depth-first search algorithm. Then, any stutter SCC is collapsed into a single state. This yields a new transition system TS' where the states are the stutter SCCs in $G(TS)$, and $C \to' C'$ with $C \neq C'$ whenever $s \to s'$ is a transition in TS with $s \in C$ and $s' \in C'$. By construction, TS' does not contain any stutter cycles. The time complexity of this step is $\mathcal{O}(M+|S|)$. Note that

$$s_1 \approx_{TS} s_2 \text{ if and only if } C_1 \approx_{TS'} C_2$$

where C_i denotes the stutter SCC which contains state s_i.

Efficient Realization of Refining a Partition From now on, it is assumed that transition system TS has *no* stutter cycles. We now focus our attention on an efficient realization of the refinement operator and the search for a splitter (see the loop condition in Algorithm 37). Assume that list representations for the sets $Pre(s)$ and $Post(s)$ are available for any state s. Let Π be a partition of S that is coarser than $S/\approx$ and finer than Π_{AP}. The goal is to check the instability of Π for a block C by a linear search in the lists $Pre(s)$ for $s \in C$ rather than to consider the set $Pre^*_\Pi(C)$. To that end, we aim at a characterization of a splitter in terms of direct predecessors of C. An important notion to enable such characterization is that of an *exit* state in a block B, say, i.e., a state whose only outgoing transitions lead to states that are outside B.

Definition 7.140. Exit States

Let TS be a finite transition system without stutter cycles and B be a block of a partition Π for S, the state space of TS. The *exit states* of B and partition Π are defined by

$$Bottom(B) = \{s \in B \mid Post(s) \cap B = \varnothing\} \text{ and } Bottom(\Pi) = \bigcup_{B \in \Pi} Bottom(B).$$

Since TS does not have stutter cycles and is finite, any block B of partition Π that is finer than Π_{AP} has at least one exit state. This can be seen by contraposition. Assume block $B \in \Pi$ does not contain an exit state. Then there exists an infinite path $s_0\, s_1\, s_2 \ldots$ consisting of states in B. Since TS is finite, there exists a pair of indices i, j such that ■

$0 \leqslant i < j$ with $s_i = s_j$. As Π is finer than Π_{AP}, all states in B are equally labeled. Hence, the path fragment $s_i\, s_{i+1} \ldots s_j$ is a stutter cycle. This contradicts TS having no stutter cycles. Thus, $Bottom(B) \neq \varnothing$ for any block $B \in \Pi$ where Π refines Π_{AP}.

The following lemma is the key for checking whether a block $C \in \Pi$ is a Π-splitter by a linear search in the lists $Pre(s)$ for $s \in C$.

Lemma 7.141. Local Splitter Criterion

Let TS be a finite transition system without stutter cycles, Π a partition of TS's state space S and C, $B \in \Pi$. Then:

$$C \text{ is a } \Pi\text{-splitter for } B \;\; \textit{iff} \;\; (B \neq C \;\wedge\; B \cap Pre(C) \neq \varnothing \;\wedge\; Bottom(B) \setminus Pre(C) \neq \varnothing)\,.$$

Proof: $\Leftarrow$: Assume $B \neq C$, $B \cap Pre(C) \neq \varnothing$ and $Bottom(B) \setminus Pre(C) \neq \varnothing$. Then, there exist states $t \in Bottom(B) \setminus Pre(C)$ and $s \in B \cap Pre(C)$. Since $t \in Exit(B)$ and $t \notin Pre(C)$, all direct successors of t are outside $B \cup C$, and thus $t \in B \setminus Pre^*_\Pi(C)$. Thus, by Definition 7.134, C is a Π-splitter for B.

$\Rightarrow$: Let C be a Π-splitter for B. By Definition 7.134 of a Π-splitter, it follows that $B \neq C$ and $s \in B \cap Pre(C)$ for some s. It remains to show that $B \setminus Pre^*_\Pi(C) \neq \varnothing$ implies $Bottom(B) \setminus Pre(C) \neq \varnothing$. This is done by contraposition. Assume $Bottom(B) \setminus Pre(C) = \varnothing$, i.e., $Bottom(B) \subseteq Pre(C)$. Let $u \in B$. Since there are no stutter cycles in TS and TS is finite, there exists a path fragment $u \ldots t$ consisting of B-states only with $t \in Bottom(B)$. But, since $Bottom(B) \subseteq Pre(C)$, we have $t \in Pre(C)$, and $u \in Pre^*_\Pi(C)$. Thus, $u \in B \cap Pre^*_\Pi(C)$. As this applies to any $u \in B$, it follows $B \subseteq Pre^*_\Pi(C)$, which contradicts that C is a Π-splitter of B. ∎

This result allows checking whether block C of partition Π is a Π-splitter for some $B \in \Pi$, by investigating the *direct* predecessors of C. For any state $s \in C$, the list $Pre(s)$ is traversed and all states $s' \in Pre(s)$ and blocks $[s']_\Pi$ (not all states in the block, just the block itself) are marked. Then, for each block $B \in \Pi$, $B \neq C$, we have

$$C \text{ is a } \Pi\text{-splitter for } B \quad \text{iff} \quad \begin{array}{l} B \text{ is marked and } Bottom(B) \\ \text{contains an unmarked state} \end{array}$$

In this way, the C-stability of Π can be checked in time $\mathcal{O}(|Pre(C)|)$ when using appropriate data structures (such as lists) for Π and $Bottom(B)$ for $B \in \Pi$. For instance, a list representation of Π with pointers from Π's blocks to the states in the blocks and vice versa, enables generating a list consisting of (pointers to) the marked blocks in time $\mathcal{O}(|Pre(C)|)$.

To check whether there is a marked block B where $Bottom(B)$ contains an unmarked state, a linear search through the list of marked blocks suffices. If B is the current (marked) block, then we traverse the list representing $Bottom(B)$ until an unmarked state (i.e., a state $t \notin Pre(C)$) has been found (or the end of the list has been reached). The time complexity of this strategy is $\mathcal{O}(|Pre(C)|)$, since the number of marked blocks is at most $|Pre(C)|$ and the total number of elements to be considered when traversing the lists for the exit states is bounded by $|Pre(C)| + 1$. (The "+1" covers the case where $t \notin Pre(C)$ has been reached.) Ranging over all blocks $C \in \Pi$, the stability of Π can be checked in time $\mathcal{O}(M)$ since $M = \sum_{C \in \Pi} |Pre(C)|$, where (as before) M denotes the number of edges in the state graph of TS.

Moreover, if C is a Π-splitter (i.e., Π is not C-stable), then the above technique returns a block $B \in \Pi$ such that C is a Π-splitter for B. It remains to explain how the refinement of B into the two subblocks:

$$B_1 = B \cap Pre^*_\Pi(C) \text{ and } B_2 = B \setminus Pre^*_\Pi(C)$$

can be realized in time $\mathcal{O}(M)$. For this, we proceed as follows. We start with two new (variables for the) blocks B_1 and B_2, which are initially empty. On traversing the list of all exit states of B, any marked state $s \in Bottom(B)$ is added to B_1 and any unmarked state $s \in Bottom(B)$ is added to B_2. (Recall that exactly the states in $Pre(C)$ are marked.) We then apply a standard graph algorithm to determine a *reversed topological order* $s_1, \ldots, s_k$ of the nonexit states in B, i.e., $B \setminus Bottom(B) = \{ s_1, \ldots, s_k \}$ and $i > j$ whenever there is a transition $s_i \to s_j$. Note that such a topological order exists since TS has no stutter cycles. It can be computed in $\mathcal{O}(M+|S|)$. The nonexit states of B are traversed in reversed topological order. On encountering state $s = s_i$ for some $0 < i \leqslant k$, we perform the following. If s is unmarked, i.e., $s \in B \setminus Pre(C)$, the list $Post(s)$ is traversed. If $t \in Post(s) \cap B_1$ is encountered for some t, then s is added to B_1; otherwise, s is added to B_2. This is justified by the observation that

- either s is marked in which case $s \in B \cap Pre(C)$,
- or s is unmarked and $s \to t$ for some $t \in B_1$, in which case either $t \in Bottom(B) \cap Pre(C)$ or $t \in \{ s_1, \ldots, s_{i-1} \}$ and there exists a path fragment $t \ldots v$ consisting of states in B_1 with $v \in Bottom(B) \cap Pre(C)$.

If all states $s \in \{ s_1, \ldots, s_k \} = B \setminus Bottom(B)$ have been considered, then B_1 consists of all states in $B \cap Pre^*_\Pi(C)$, while B_2 contains all states $B \setminus Pre^*_\Pi(C)$.

The above yields that there is an algorithm with time complexity $\mathcal{O}(M)$ that checks

whether for a given partition Π there is a pair (B, C) of blocks in Π such that C is a Π-splitter for C, and returns such a pair (B, C), if there is some. Moreover:

Lemma 7.142. Time Complexity of the Refinement Operator

$Refine_{\approx}(B, C)$ can be computed in time $\mathcal{O}(M)$.

Hence, each iteration in Algorithm 37 causes the cost $\mathcal{O}(M)$. As each iteration yields a proper refinement of the current partition Π, the number of iterations is at most $|S|$. We conclude:

Theorem 7.143. Time Complexity of Algorithm 37

The stutter bisimulation quotient of TS can be computed with Algorithm 37 in time $\mathcal{O}(\ |S| \cdot (|AP| + M)\ )$, under the assumption that $M \geqslant |S|$, where M is the number of edges in the state graph $G(TS)$.

Proof: The costs of computing the initial partition are $\mathcal{O}(|S| \cdot |AP|)$. At most $|S|$ iterations take place. Each refinement in an iteration is in $\mathcal{O}(M)$. Checking the loop condition can also be done in $\mathcal{O}(M)$. ■

Although irrelevant for the asymptotic time complexity, the following observation can be helpful in practice to increase the efficiency of the splitter search: it provides a criterion to skip checking the Π-splitter-condition for certain blocks C.

Lemma 7.144. Stability Criterion

Let TS be a finite transition system without stutter cycles and let Π and Π' partitions such that Π' is finer than Π and $Bottom(\Pi) = Bottom(\Pi')$. Then: there is no common block C of Π and Π' such that Π is stable for C, while Π' is not.

Proof: By contraposition. Assume $C \in \Pi \cap \Pi'$ and C is a Π'-splitter for some block $B' \in \Pi'$, and Π is stable for C. By Lemma 7.141, $C \neq B'$, there exists $s \in B' \cap Pre(C)$, and there exists $t \in Bottom(B') \setminus Pre(C)$. Since Π refines Π', there exists a block $B \in \Pi$ which subsumes B'. $C \neq B$, since otherwise $C = B'$ as $C \in \Pi \cap \Pi'$, and $s \in B \cap Pre(C)$. From $Bottom(\Pi) = Bottom(\Pi')$, we derive that $t \in Bottom(B') \subseteq Bottom(B)$. By Lemma 7.141, C is a Π-splitter for B. This, however, contradicts the assumption that Π is stable for C. ■

Quotienting for Divergence-Sensitive Stutter Bisimulation The computation of the equivalence classes under $\approx^{div}$ is essentially based on the following steps. Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a finite transition system. As an initial step, TS is transformed into a *divergence-sensitive* transition system $\overline{TS}$ such that the equivalence classes under $\approx$ in $\overline{TS}$ coincide with the equivalence classes under $\approx^{div}$ in TS. (Recall that TS is divergence-sensitive if $\approx_{TS}$ is divergence-sensitive, and hence coincides with $\approx^{div}_{TS}$.) Applying Algorithm 37 to $\overline{TS}$ then provides the quotient space $S/\approx^{div}$ of TS.

The divergence-sensitive transition system $\overline{TS}$ is obtained as follows. TS is extended with a new state s_{div} that is not stutter-bisimilar to any other state in TS. This is established by equipping s_{div} with a unique label, *div* say. The goal is to define the transitions in $\overline{TS}$ such that the $\approx^{div}_{TS}$-divergent states of TS have a path fragment of the form $s\, v_1 \ldots v_m\, s_{div}$ in $\overline{TS}$ such that

$$s \approx^{div}_{TS} v_1 \approx^{div}_{TS} \ldots \approx^{div}_{TS} v_m.$$

That is to say, the new state s_{div} can only be reached from a $\approx^{div}_{TS}$-divergent state. Since TS is finite, every $\approx^{div}_{TS}$-divergent path contains a cycle containing only states that are equivalent under $\approx^{div}_{TS}$. By Lemma 7.138 on page 570, these are exactly the stutter cycles! So, $\overline{TS}$ is obtained from TS by inserting transitions from s to s_{div} for any state s that lies on a stutter cycle.

Definition 7.145. Divergence-Sensitive Expansion $\overline{TS}$

The *divergence-sensitive expansion* of finite transition system $TS = (S, Act, \rightarrow, I, AP, L)$ is:

$$\overline{TS} \;=\; (S \cup \{ s_{div} \}, Act \cup \{ \tau \}, \rightarrow, I, AP \cup \{ div \}, \overline{L})$$

where $s_{div} \notin S$, $\rightarrow$ extends the transition relation of TS by the transitions $s_{div} \xrightarrow{\tau} s_{div}$ and $s \xrightarrow{\tau} s_{div}$ for every state $s \in S$ on a stutter cycle in TS, and $\overline{L}(s) = L(s)$ if $s \in S$ and $\overline{L}(s_{div}) = \{ div \}$. ■

Example 7.146. Divergence-Sensitive Expansion

Consider the transition system TS over $AP = \{ a \}$ in Figure 7.49 (upper part). Its divergence-sensitive expansion is depicted below it. It is not difficult to establish that

$$S/\approx^{div}_{TS} \;=\; \{ \{ s_0, s_1 \}, \{ s_2 \}, \{ s_3 \} \} \;=\; S/\approx_{\overline{TS}}.$$

Hence, the $\approx$ equivalence classes of $\overline{TS}$ with respect to $\overline{AP} = \{ a, div \}$ correspond to the equivalence classes of TS under $\approx^{div}$ for $AP = \{ a \}$. Note that we ignored the stutter bisimulation equivalence class $\{ s_{div} \}$ in $\overline{TS}$. ■

The following theorem shows that $\approx^{div}$ in TS coincides with $\approx$ in $\overline{TS}$.

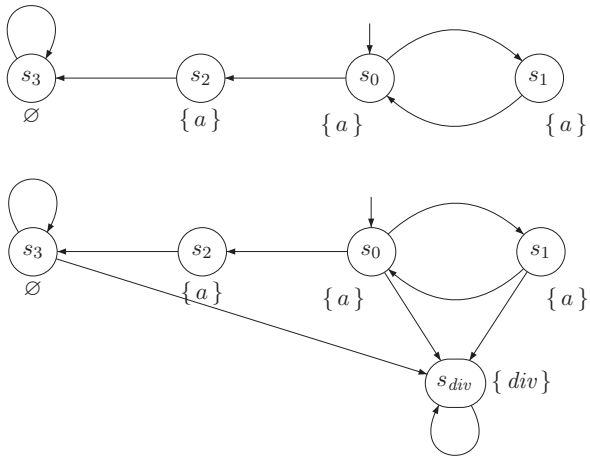


Figure 7.49: Transition system TS (upper) and its divergence-sensitive expansion $\overline{TS}$ (lower).

Theorem 7.147. Connection between TS ***and*** $\overline{TS}$

For finite transition system TS:

1. $\overline{TS}$ *is divergence-sensitive, and*
2. *for all states* $s_1, s_2 \in S$*:* $s_1 \approx^{div}_{TS} s_2$ *if and only if* $s_1 \approx_{\overline{TS}} s_2$.

Proof: We first show that

$$\mathcal{R} \;=\; \{(s_1, s_2) \in S \times S \mid s_1 \approx^{div}_{TS} s_2 \} \cup \{ (s_{div}, s_{div}) \}$$

is a stutter bisimulation for $\overline{TS}$. Clearly, all states in $\mathcal{R}$ are equally labeled. It remains to check that for $(s_1, s_2) \in \mathcal{R}$ we have that s_2 simulates transitions of s_1. This is evident for $(s_{div}, s_{div}) \in \mathcal{R}$; consider the remaining cases.

Let $(s_1, s_2) \in \mathcal{R}$ and $s_1' \in Post_{\overline{TS}}(s_1)$ such that $(s_1, s_1') \notin \mathcal{R}$. Distinguish two cases:

- If $s_1' \in S$ (i.e., $s_1' \neq s_{div}$), then $s_1' \in Post_{TS}(s_1)$ and $s_1 \not\approx^{div}_{TS} s_1'$. Since $s_1 \approx^{div}_{TS} s_2$, there exists a path fragment $s_2\, u_1 \ldots u_n\, s_2'$ in TS with

  $$s_2 \approx^{div}_{TS} u_1 \approx^{div}_{TS} \ldots \approx^{div}_{TS} u_n \quad \text{and} \quad s_1' \approx^{div}_{TS} s_2'.$$

  Thus, $s_2\, u_1 \ldots u_n\, s_2'$ is a path fragment satisfying the property $(s_2, u_j) \in \mathcal{R}$, for $0 < j \leqslant n$, and $(s_1', s_2') \in \mathcal{R}$.

- If $s_1' = s_{div}$, then s_1 is on a stutter cycle in TS, and thus s_1 is divergent. As $s_1 \approx_{TS}^{div} s_2$, state s_2 is divergent. Since TS is finite, there exists a path fragment $s_2\, u_1 \ldots u_n$ such that
$$s_2 \approx_{TS}^{div} u_1 \approx_{TS}^{div} \ldots \approx_{TS}^{div} u_n$$
where u_n is on a stutter cycle. Therefore, $u_n \to s_{div}$ is a transition in $\overline{TS}$ and $s_2\, u_1 \ldots u_n\, s_{div}$ is a path fragment in $\overline{TS}$ that matches $s_1 \to s_1'$.

As a next step, we show that $\mathcal{R} = \{(s_1, s_2) \in S \times S \mid s_1 \approx_{\overline{TS}} s_2\}$ is a divergence-sensitive stutter bisimulation for TS. Obviously, $\mathcal{R}$ only contains equally labeled states; condition (2) of being a stutter bisimulation follows from the following reasoning:

- Let $(s_1, s_2) \in \mathcal{R}$ and $s_1' \in Post_{TS}(s_1)$ such that $(s_1, s_1') \notin \mathcal{R}$. Then, $s_1 \approx_{\overline{TS}} s_2$, $s_1' \in Post_{\overline{TS}}(s_1)$, and $s_1 \not\approx_{\overline{TS}} s_1'$. Thus, there exists a path fragment $s_2\, u_1 \ldots u_n\, s_2'$ in $\overline{TS}$ with
$$s_2 \approx_{\overline{TS}} u_1 \approx_{\overline{TS}} \ldots \approx_{\overline{TS}} u_n \quad \text{and} \quad s_1' \approx_{\overline{TS}} s_2'.$$
As $s_1' \neq s_{div}$, we have $s_2' \neq s_{div}$. Therefore, $s_2\, u_1 \ldots u_n\, s_2'$ is a path fragment in TS that matches $s_1 \to s_1'$.

It remains to check the divergence sensitivity of $\mathcal{R}$. Let $(s_1, s_2) \in \mathcal{R}$ such that s_1 is $\mathcal{R}$-divergent. Then, there exists a path fragment $s_1\, v_1 \ldots v_m\, s_1'\, u_1 \ldots u_n\, s_1'$ such that:
$$s_1 \approx_{\overline{TS}} v_1 \approx_{\overline{TS}} \ldots \approx_{\overline{TS}} v_m \approx_{\overline{TS}} s_1' \approx_{\overline{TS}} u_1 \approx_{\overline{TS}} \ldots \approx_{\overline{TS}} u_n$$
In particular, s_1' is on a stutter cycle. Therefore, $s_1' \to s_{div}$ is a transition in $\overline{TS}$. As $(s_1, s_2) \in \mathcal{R}$, $s_1 \approx_{\overline{TS}} s_2$. Together with $s_1 \approx_{\overline{TS}} s_1'$, we get $s_1' \approx_{\overline{TS}} s_2$. Moreover, $s_{div} \in Post_{\overline{TS}}(s_1')$, and $s_1' \not\approx_{\overline{TS}} s_{div}$. Hence, there exists a path fragment $s_2\, w_1 \ldots w_k\, s_2'\, s_{div}$ such that
$$s_2 \approx_{\overline{TS}} w_1 \approx_{\overline{TS}} \ldots \approx_{\overline{TS}} w_k \approx_{\overline{TS}} s_2' \quad \text{and} \quad s_2' \to s_{div}.$$
Note, since only state s_{div} is labeled with div, if $s \approx_{\overline{TS}} s_{div}$ then $s = s_{div}$.

Since $s_2' \to s_{div}$ is a transition in $\overline{TS}$, s_2' belongs to a stutter cycle $s_2'\, t_1 \ldots t_\ell\, s_2'$. By Lemma 7.138 (page 570), $s_2' \approx_{TS} t_1 \approx_{\overline{TS}} \ldots \approx_{\overline{TS}} t_\ell$. As $s_2 \approx_{TS} s_2'$, it follows that s_2 is $\mathcal{R}$-divergent.

Finally, we consider the divergence sensitivity of $\overline{TS}$. By definition, $\overline{TS}$ is divergence-sensitive if $\approx_{\overline{TS}}$ is divergence-sensitive. This follows from the fact that $\approx_{\overline{TS}}$ coincides with $\approx_{TS}^{div}$. ■

Theorem 7.147 shows that the quotient space $S/\!\approx$ of transition system TS can be computed by applying Algorithm 37 to the divergence-sensitive expansion $\overline{TS}$. In order to

construct $\overline{TS}$, all states on a stutter cycle must be determined, as these states need to be equipped with a transition to s_{div}. This can be done by generating the nontrivial SCCs in the digraph that is obtained from $G(TS)$ by only considering stutter steps. Let $G_{stutter}(TS) = (V, E)$ with $V = S$, the state space of TS and $E = \{\, (s,t) \in S \times S \mid L(s) = L(t) \,\}$.

Summarizing, the required steps to compute the quotient transition system $TS/\!\approx^{div}$ are:

1. Construct the expansion $\overline{TS}$ by determining the SCCs in $G_{stutter}(TS)$, and inserting transitions $s \to s_{div}$ and $s_{div} \to s_{div}$ for any state s in a nontrivial SCC of $G_{stutter}$.
2. Apply Algorithm 37 to $\overline{TS}$ to obtain the quotient space
$$S/\!\approx^{div}_{TS} \;=\; S/\!\approx_{\overline{TS}}\,.$$
The transition system $\overline{TS}/\!\approx$ is constructed as follows. Any $C \in S/\!\approx^{div}$ that contains an initial state of TS is identified as initial state. The labeling of $C \in S/\!\approx^{div}$ equals the labeling of any $s \in C$. All transitions $s \to s'$ with $s \not\approx^{div}_{TS} s'$ are lifted to transitions of the corresponding state classes.
3. $TS/\!\approx^{div}$ is obtained from $\overline{TS}/\!\approx$ as follows. The transitions $s \to s_{div}$ in $\overline{TS}$ are replaced by self-loops $[s]_{div} \to [s]_{div}$, and the state s_{div} is deleted.

Example 7.148.

Consider again the transition system TS over $AP = \{\, a \,\}$ in Figure 7.49 (upper part) and its divergence-sensitive expansion $\overline{TS}$ (see lower part of the same figure). The quotient transition system $\overline{TS}/\!\approx$ is depicted in Figure 7.50 (upper part). The corresponding transition system $TS/\!\approx$ is obtained by turning all transitions to $[s]_{div}$ into self-loops at the source state, and deleting $[s]_{div}$. This yields the transition system depicted in the lower part of Figure 7.50. ■

The costs of the described technique are dominated by the costs of Algorithm 37 and the computation of the SCCs in digraph $G_{stutter}$. Theorem 7.143 on page 574 yields:

Theorem 7.149. Time Complexity of Constructing $TS/\!\approx^{div}$

The quotient space of TS under $\approx^{div}$ *can be computed in time* $\mathcal{O}(\,(|S|+M) \,+\, |S|\cdot(|AP|+M)\,)$*, under the assumption that* $M \geqslant |S|$*, where* M *denotes the number of edges in the state graph* $G(TS)$*.*

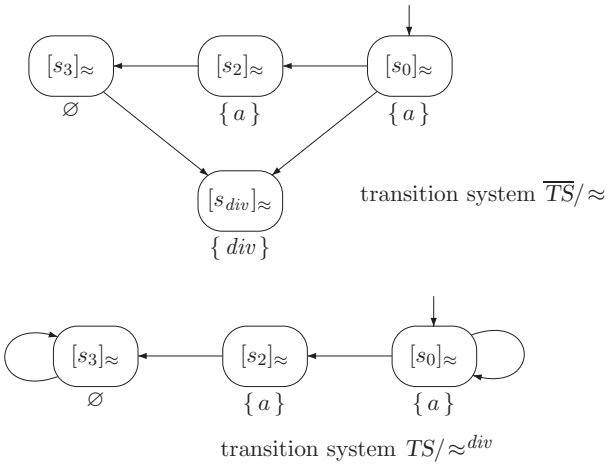


Figure 7.50: Example of computing the quotient system under $\approx^{div}$.

7.9 Summary

- A bisimulation equivalence only relates a pair of states if both states are equally labeled and can mutually mimic their outgoing transitions. In a simulation preorder it suffices that one state can mimic the other, but not necessarily the reverse.
- Bisimulation is strictly finer than simulation equivalence and trace equivalence.
- In general, simulation and trace equivalence are incomparable. For transition systems without terminal states, simulation equivalence is strictly finer than trace equivalence.
- For *AP*-deterministic transition systems, bisimulation, simulation, and trace equivalence coincide.
- Bisimulation equivalence and the simulation order can be computed for finite transition systems in polynomial time. The problem of checking trace equivalence or trace inclusion is PSPACE-complete.
- For finite transition systems without terminal states, bisimulation equivalence corresponds to CTL* and CTL equivalence. (This result is considerable, since CTL* includes the two incomparable logics CTL and LTL.)
- A logical characterization of the simulation preorder can be provided by the universal or the existential fragments of CTL* and CTL.
- Stutter trace equivalence is a variant of trace equivalence that abstracts from stutter steps, i.e, transitions between equally labeled states. It coincides with LTL$_{\setminus\bigcirc}$ equivalence.

- Stutter bisimulation is a variant of bisimulation where transitions can be mimicked by path fragments (rather than by single transitions). It is incomparable to stutter trace equivalence.
- A state is divergent if a path consisting only of stutter steps emanates from it. Divergence-sensitive stutter bisimulation is a stutter bisimulation that distinguishes divergent from nondivergent states.
- Divergence-sensitive stutter bisimulation is strictly finer than stutter trace equivalence and coincides with $\mathrm{CTL}^*_{\setminus\bigcirc}$- and $\mathrm{CTL}_{\setminus\bigcirc}$ equivalence.
- Divergence-sensitive bisimulation can be computed for finite transition systems in polynomial time.

Figure 7.51 on page 580 summarizes the results concerning the treated notions of bisimulation. For the sake of simplicity, the costs of the initialization phase of the quotienting algorithms have been omitted.

	bisimulation equivalence	simulation order	stutter equivalence with divergence	trace equivalence
preservation of temporal-logical properties	CTL^* CTL	$\forall\mathrm{CTL}^*/\exists\mathrm{CTL}^*$ $\forall\mathrm{CTL}/\exists\mathrm{CTL}$	$\mathrm{CTL}^*_{\setminus\bigcirc}$ $\mathrm{CTL}_{\setminus\bigcirc}$	LTL (LT properties)
checking equivalence	PTIME	PTIME	PTIME	PSPACE-complete
graph minimization	PTIME $\mathcal{O}(M \cdot \log \lvert S\rvert)$	PTIME $\mathcal{O}(M \cdot \lvert S\rvert)$	PTIME $\mathcal{O}(M \cdot \lvert S\rvert)$	—

Figure 7.51: Overview of equivalences on transition systems.

7.10 Bibliographic Notes

Bisimulation and simulation. The notions of bisimulation and simulation in this chapter are state-based versions of the action-based bisimulation and simulation. Action-based

bisimulation has been brought up independently by Milner [296] and Park [322]. Action-based simulation originates from Milner [295]. Milner [296] also introduced observational bisimulation, a weak bisimulation that abstracts away from internal steps (i.e., τ-labeled transitions). (Exercise 7.22 considers a state-based variant of observational equivalence.) Stutter bisimulation is the state-based variant of branching bisimulation introduced by van Glabbeek and Weijland [406]. As noticed by Groote and Vaandrager [176], divergence-sensitive stutter bisimulation coincides with stutter equivalence introduced by Browne, Clarke and Grumberg [67]. The alternative characterizations of stutter bisimulation by norm functions for finite and infinite transition systems is due to Namjoshi [311]. (Theorem 7.126 treats only finite transition systems.) Griffioen and Vaandrager [175] apply a (slightly) different notion of norm functions to define simulation relations. A comprehensive comparison of various trace-based and (bi)simulation relations has been provided by van Glabbeek [404, 405]. The concept of bisimulation and simulation relations has been refined and studied in various directions (such as axiomatization, refinement, coinduction, domain theoretic approaches); see, e.g., [298, 1, 4, 279, 397, 101]. The use of simulation relations for abstraction has been studied in, e.g., [90, 277, 133, 109, 105].

Logical characterizations. The first logical characterization of bisimulation equivalence has been provided in the action-based setting by Hennessy and Milner [197] for a modal logic, called Hennessy-Milner logic, with an action-labeled next step operator. Browne, Clarke and Grumberg [67] showed that state-based bisimulation coincides with CTL- and CTL* equivalence; see Theorem 7.20, and the analogous result for divergence-sensitive stutter bisimulation and $\mathrm{CTL}_{\setminus\bigcirc}/\mathrm{CTL}^*_{\setminus\bigcirc}$; see Theorem 7.128. Kucera and Schnoebelen [247] recently presented a refinement of the latter result. De Nicola and Vaandrager [314] studied the connection between temporal logics and variants of stutter bisimulation in the action- and state-based setting. The logical characterizations of the simulation order by means of the existential and universal fragment of CTL and CTL* is due to Clarke, Grumberg and Long [90]. The observation that any LTL formula which induces a stutter-insensitive LT property can be expressed in $\mathrm{LTL}_{\setminus\bigcirc}$ (see Exercise 7.20) is by Peled and Wilke [329]; see also Etessami [147]. Etessami [146] defined a variant LTL that is as expressive as stutter-invariant ω-regular languages.

Quotienting algorithms. The first partition refinement algorithm for bisimulation is due to Kanellakis and Smolka [231]. The refined version with the ternary refinement operator (see Algorithm 32) originates from Paige and Tarjan [318]. Kanellakis and Smolka [231] also showed that the trace equivalence problem is PSPACE-complete; see Theorem 7.46. The initial algorithm for calculating the simulation order has been suggested by Cleaveland, Parrow, and Steffen [97]. Henzinger, Henzinger, and Kopke [198] developed a more efficient algorithm; see Algorithm 36. Other algorithms for computing the simulation order have been suggested by, among others, Tan and Cleaveland [385] and Bustan and Grumberg [76]. A survey of bisimulation- and simulation-quotienting algorithms has been

given by Cleaveland and Sokolsky [98]. The quotienting algorithms for stutter bisimulation quotient (see Algorithm 37) go back to Groote and Vaandrager [176]. Moller and Smolka [302] discuss the computational complexity of checking the bisimilarity of several classes of processes. Thanks to extensive studies by Fisler and Vardi [152, 153, 154], it is known that bisimulation minimization for LTL model checking and invariant verification leads to drastic state space reductions (up to logarithmic savings) but at a time penalty: the time to minimize and model-check the resulting quotient significantly exceeds the time to verify the original transition system. In this monograph, we considered quotienting algorithms for finite transition systems. Semialgorithms for bisimulation quotienting of infinite transition systems (provided the bisimulation quotient is finite) have been presented by Bouajjani, Fernandez, and Halbwachs [60] and Lee and Yannakakis [266]. Henzinger, Henzinger, and Kopke [198] introduced a semialgorithmic approach for the simulation order to treat infinite state spaces. A classification and algorithms for infinite systems with finite quotients for other equivalences have been given by Henzinger, Majumdar, and Raskin [199].

For further reading we refer to the Handbook of Process Algebra [46] (and the literature mentioned therein) which treats many aspects of (bi)simulations and other implementation relations.

7.11 Exercises

Exercise 7.1. Consider the four transition systems in Figure 7.52. Determine for each pair (TS_i, TS_j) for $0 < i \neq j \leqslant 4$ of these transition systems whether they are bisimilar. Justify your answer by either providing the bisimulation relation for (TS_i, TS_j) or a CTL formula Φ such that $TS_i \models \Phi$ and $TS_j \not\models \Phi$.

Exercise 7.2. Let TS be a transition system.

(a) Show that the union of bisimulation relations for TS is a bisimulation for TS.

(b) Show that the union of simulation relations for TS is a simulation for TS.

Exercise 7.3. The goal of this exercise is to discuss the role of AP-determinism (see Definition 7.65) for the connection between (bi)simulation and trace relations. In particular, this exrecise completes the proofs for Theorem 7.64 on page 511 and Corollary 7.72 on page 514.

Suppose that TS_1 and TS_2 are transition systems over AP that are both AP-deterministic.

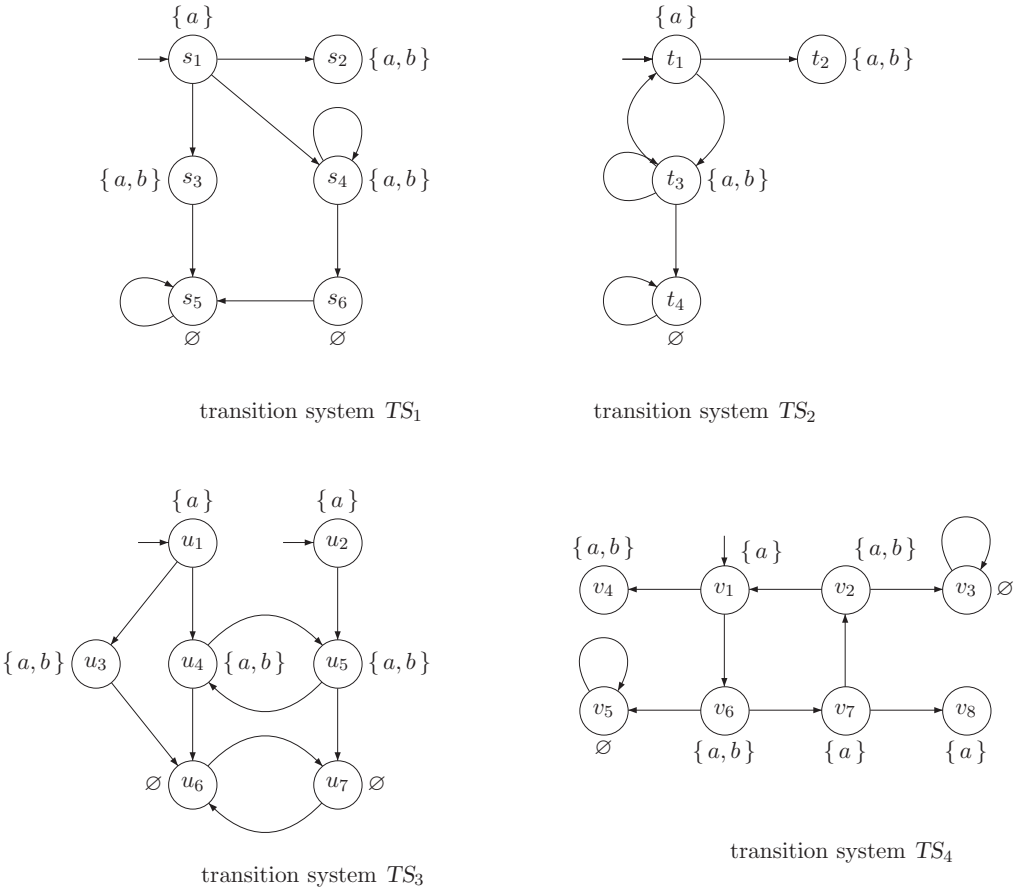


Figure 7.52: Four transition systems for Exercises 7.1, 7.11, 7.17, 7.21 and 7.22.

(a) Prove the equivalence of the following statements (1), (2), (3) and (4):

(1) $TS_1 \sim TS_2$

(2) $TS_1 \simeq TS_2$

(3) $Traces_{fin}(TS_1) = Traces_{fin}(TS_2)$

(4) $Traces(TS_1) = Traces(TS_2)$

(b) Prove or disprove: If $Traces_{fin}(TS_1) \subseteq Traces_{fin}(TS_2)$ then $TS_1 \preceq TS_2$.

(c) Prove or disprove: If $TS_1 \preceq TS_2$ then $Traces(TS_1) \subseteq Traces(TS_2)$.

(d) Prove or disprove: If $Traces_{fin}(TS_1) \subseteq Traces_{fin}(TS_2)$ then $Traces(TS_1) \subseteq Traces(TS_2)$.

Exercise 7.4. Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system. The relations $\sim_n \subseteq S \times S$ are inductively defined by:

(a) $s_1 \sim_0 s_2$ iff $L(s_1) = L(s_2)$.

(b) $s_1 \sim_{n+1} s_2$ iff

(i) $L(s_1) = L(s_2)$,

(ii) for all $s_1' \in \mathit{Post}(s_1)$ there exists $s_2' \in \mathit{Post}(s_2)$ with $s_1' \sim_n s_2'$,

(iii) for all $s_2' \in \mathit{Post}(s_2)$ there exists $s_1' \in \mathit{Post}(s_1)$ with $s_1' \sim_n s_2'$.

Show that for finite TS it holds that $\sim_{TS} = \bigcap_{n \geqslant 0} \sim_n$, i.e.,

$$s_1 \sim_{TS} s_2 \text{ if and only if } s_1 \sim_n s_2 \text{ for all } n \geqslant 0.$$

Does this also hold for infinite transition systems?

Exercise 7.5.

(a) Consider transition system TS shown in Figure 7.53. Apply the transformation $TS \mapsto TS_{act}$ (see page 467) and check that $\sim_{TS}$ coincides with action-based bisimulation $\sim^{Act}_{TS_{act}}$ on TS_{act}.

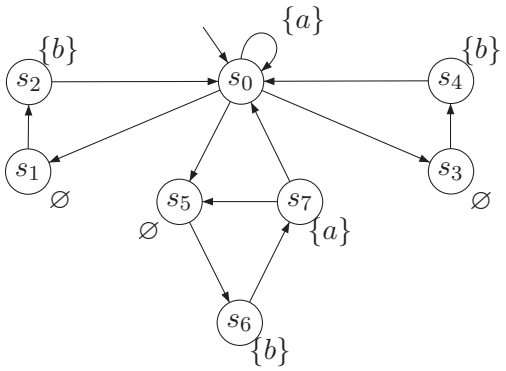


Figure 7.53: Transition system TS for Exercise 7.5.

(b) Prove that for all states s_1, s_2 in TS,

$$s_1 \sim^{Act}_{TS} s_2 \text{ if and only if } s_1 \sim_{TS_{state}} s_2$$

where TS_{state} is obtained from TS by the transformation defined on page 467.

EXERCISE 7.6.

(a) Give a transition system TS without terminal states and containing states s_1, s_2 of TS such that $s_1 \not\equiv_{LTL} s_2$ and there is no LTL formula φ with $s_2 \models \varphi$ and $s_1 \not\models \varphi$. (See the remark on page 469.)

(b) Let TS_1 and TS_2 be transition systems over AP without terminal states such that $TS_1 \not\equiv_{CLT} TS_2$. Prove or disprove the claim: there exists a CTL formula Φ over AP such that $TS_1 \models \Phi$ and $TS_2 \not\models \Phi$.

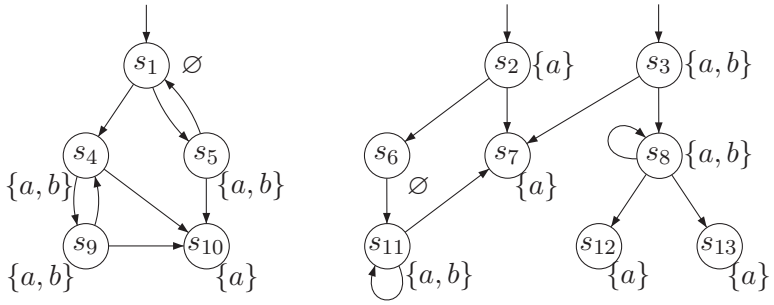


Figure 7.54: Transition system TS for Exercises 7.7, 7.8 and 7.23.

EXERCISE 7.7. Consider the transition system TS over $AP = \{a, b\}$ shown in Figure 7.54.

(a) Determine the bisimulation equivalence $\sim_{TS}$ and depict the bisimulation quotient system $TS/\!\sim$.

(b) Provide CTL master formulae Φ_C for each bisimulation equivalence class C.

EXERCISE 7.8. Consider again the transition system TS with state space $S = \{s_1, s_2, \ldots, s_{13}\}$ shown in Figure 7.54. Apply Algorithms 31 and 32 to compute the bisimulation quotient $S/\sim_{TS}$.

EXERCISE 7.9. Let TS be a finite transition system over AP without terminal states. Provide an algorithm that computes for each bisimulation equivalence class C a CTL master formula (i.e., a CTL formula Φ_C with $Sat(\Phi_C) = C$).

(*Hint: There is a simple algorithm which relies on an extension of the partition refinement technique*).

EXERCISE 7.10. Consider the transition systems TS_1 and TS_2 shown in Figure 7.55. Apply Algorithms 31 (page 487) and 32 (page 489) to check whether $TS_1 \sim TS_2$.

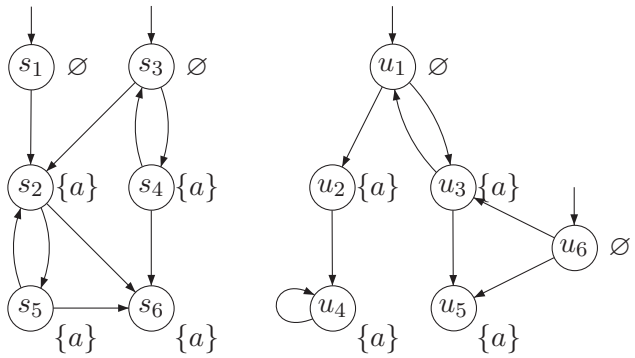


Figure 7.55: Are TS_1 (left) and TS_2 (right) bisimilar (Exercise 7.10)?

Exercise 7.11. Consider the four transition systems TS_1, TS_2, TS_3, TS_4 in Figure 7.52 on page 583. Check whether $TS_i \preceq TS_j$ and $TS_i \simeq TS_j$ for all indices i, j. Justify your answer by either establishing a simulation for (TS_i, TS_j) or by providing a $\forall$CTL formula Φ with $TS_j \models \Phi$ and $TS_i \not\models \Phi$ (if $TS_i \not\preceq TS_j$).

Exercise 7.12. Consider the transition systems TS_1 (left) and TS_2 (right) in Figure 7.56.

(a) Show that $TS_2 \preceq TS_1$ by providing a simulation relation for (TS_2, TS_1).

(b) Show that $TS_1 \not\preceq TS_2$ by providing a $\forall$CTL formula $\Phi_\forall$ and a $\exists$CTL formula $\Phi_\exists$ such that $TS_1 \not\models \Phi_\forall$, but $TS_2 \models \Phi_\forall$ and $TS_1 \models \Phi_\exists$, but $TS_2 \not\models \Phi_\exists$.

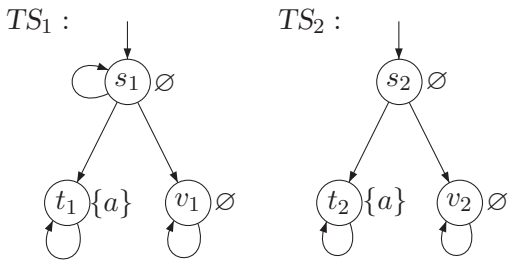


Figure 7.56: $TS_2 \preceq TS_1$, but $TS_1 \not\preceq TS_2$ (Exercise 7.12).

Exercise 7.13. Consider the transition systems TS_1 (left) and TS_2 (right) in Figure 7.57.

(a) Check whether $TS_1 \preceq TS_2$, $TS_2 \preceq TS_1$ or $TS_1 \simeq TS_2$.

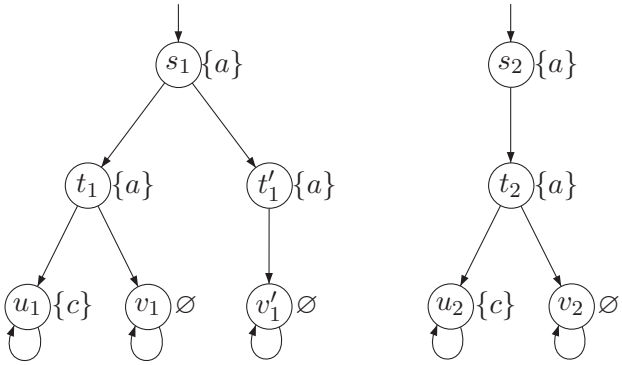


Figure 7.57: Transition systems TS_1 (left) and TS_2 (right) for Exercise 7.13.

(b) Consider the composite transition system $TS = TS_1 \oplus TS_2$. Depict the simulation quotient system $TS/\preceq$ and verify that $TS/\preceq$ and TS are simulation equivalent, but not bisimilar.

Exercise 7.14. Complete the proof of Theorem 7.76 by showing the correctness of the following two statements (a) and (b):

(a) If s_1 and s_2 are states in TS with $s_1 \preceq_{TS} s_2$ then for all $\forall$CTL* state formulae Φ: $s_2 \models \Phi$ implies $s_1 \models \Phi$.

(b) If $\pi_1 = s_{0,1}\, s_{1,1}\, s_{2,i} \ldots$ and $\pi_1 = s_{0,2}\, s_{1,2}\, s_{2,2} \ldots$ infinite path fragments in TS such that $s_{j,1} \preceq_{TS} s_{j,2}$ for all $j \geqslant 0$ then for all $\forall$CTL* path formulae φ: $\pi_2 \models \varphi$ implies $\pi_1 \models \varphi$.

Exercise 7.15. Indicate for each of the following claims whether it is a correct statement for finite transition systems:

(a) $\forall$CTL equivalence is finer than LTL equivalence.

(b) LTL equivalence is finer than $\forall$CTL equivalence.

(c) $\exists$CTL equivalence is finer than $\forall$CTL equivalence.

(d) $\exists$CTL equivalence is finer than LTL equivalence.

(e) $\exists$CTL* equivalence is finer than CTL equivalence.

Justify your answers.

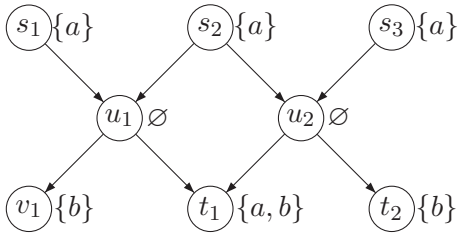
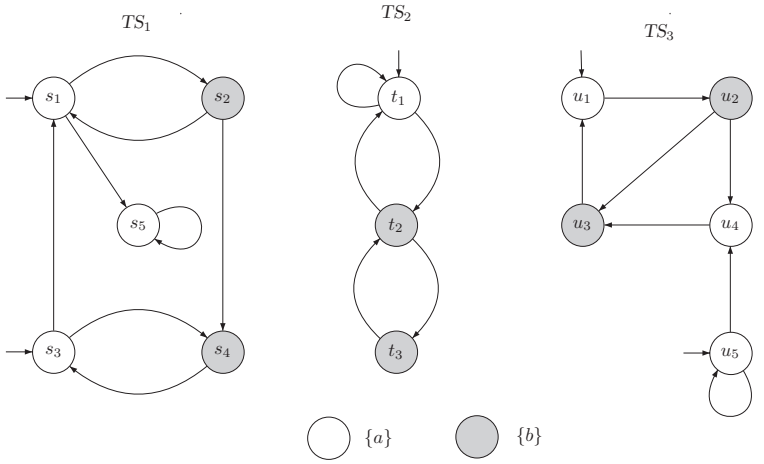


Figure 7.58: Transition system TS over $AP = \{a, b\}$ for Exercise 7.16.

Exercise 7.16. Consider the transition system TS as depicted in Figure 7.58. The initial states are irrelevant and have been omitted. Apply Algorithm 36 (page 526) to compute the simulation order $\preceq_{TS}$.

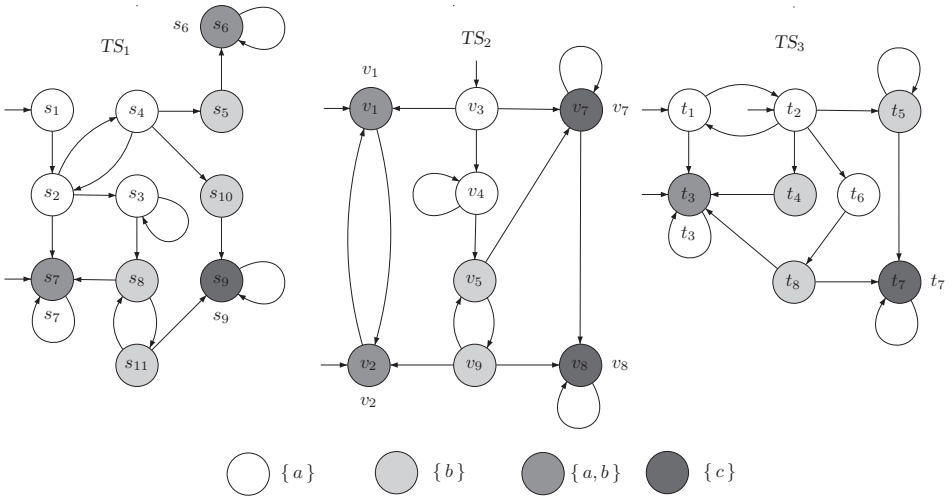
Exercise 7.17. Which of the transition systems in Figure 7.52 on page 583 are stutter trace equivalent?

Exercise 7.18. Consider the transition systems depicted below, where the gray filling of a state indicates the propositions that hold in that state.



For each $i, j \in \{\,1 \ldots 3\,\} \times \{\,1 \ldots 3\,\}$, $i \neq j$, determine whether $TS_i \cong TS_j$, $TS_i \sqsubseteq TS_j$ or $TS_i \not\sqsubseteq TS_j$. Justify your answer.

Exercise 7.19. Consider the transition systems depicted below, where the gray filling of a state indicates the propositions that are hold in that state.



For each $i, j \in \{1 \ldots 3\} \times \{1 \ldots 3\}$, $i \neq j$, determine whether $TS_i \approx TS_j$ or $TS_i \not\approx TS_j$. Justify your answer.

Exercise 7.20. Let φ be an LTL formula such that $\mathit{Words}(\varphi)$ is stutter-insensitive. Show that φ is equivalent to some $\mathrm{LTL}_{\setminus\bigcirc}$ formula ψ.

Exercise 7.21. Which of the transition systems in Figure 7.52 on page 583 are stutter bisimulation equivalent?

Exercise 7.22. Observational equivalence $\approx_{obs}$ is a slight variant of stutter bisimulation equivalence where state s_2 is allowed to perform a path fragment

$$\underbrace{s_2\, u_1 \ldots u_m}_{\text{stutter steps}}\ \underbrace{v_1\ \ldots v_k\, s_2'}_{\text{stutter steps}}$$

with arbitrary stutter steps at the beginning and at the end and $s_1' \approx_{obs} s_2'$ to simulate a transition $s_1 \to s_1'$ of an observational equivalent state s_1. I.e., it is not required that s_2 and states u_i are observationally equivalent, or that s_2' and v_i are observationally equivalent. For the special case where $s_1 \to s_1'$ is a stutter step the path fragment of length 0 (consisting of state $s_2 = s_2'$) can be used to simulate $s_1 \to s_1'$.

The formal definition of observational equivalence is as follows. Let TS_1 and TS_2 be two transition systems over AP with state spaces S_1 and S_2, respectively. A binary relation $\mathcal{R} \subseteq S_1 \times S_2$ is an *observational bisimulation* for (TS_1, TS_2) iff the following conditions are satisfied:

(A) Every initial state of TS_1 is related to an initial state of TS_2, and vice versa. That is,

$$\forall s_1 \in I_1\, \exists s_2 \in I_2.\ (s_1, s_2) \in \mathcal{R} \quad \text{and} \quad \forall s_2 \in I_2\, \exists s_1 \in I_1.\ (s_1, s_2) \in \mathcal{R}$$

(B) For all $(s_1, s_2) \in \mathcal{R}$ the following conditions (1), (2) and (3) hold:

(1) If $(s_1, s_2) \in \mathcal{R}$ then $L_1(s_1) \;=\; L_2(s_2)$

(2) If $(s_1, s_2) \in \mathcal{R}$ and $s_1' \in \mathit{Post}(s_1)$ then there exists a path fragment $u_0\, u_1 \ldots u_n$ such that $n \geqslant 0$ and $u_0 = s_2$, $(s_1', u_n) \in \mathcal{R}$ and, for some $m \leqslant n$, $L_2(u_0) = L_2(u_1) = \ldots = L_2(u_m)$ and $L_2(u_{m+1}) = L_2(u_{m+2}) = \ldots = L_2(u_n)$.

(3) If $(s_1, s_2) \in \mathcal{R}$ and $s_2' \in \mathit{Post}(s_1)$ then there exists a path fragment $u_0\, u_1 \ldots u_n$ such that $n \geqslant 0$ and $u_0 = s_2$, $(u_n, s_2') \in \mathcal{R}$ and, for some $m \leqslant n$, $L_1(u_0) = L_1(u_1) = \ldots = L_1(u_m)$ and $L_1(u_{m+1}) = L_1(u_{m+2}) = \ldots = L_1(u_n)$.

TS_1 and TS_2 are called *observational-equivalent*, denoted $TS_1 \approx_{obs} TS_2$, if there exists an observational bisimulation for (TS_1, TS_2).

(a) Which of the transition systems in Figure 7.52 on page 583 are observational-equivalent?

(b) Show that $TS_1 \approx TS_2$ implies $TS_1 \approx_{obs} TS_2$.

(c) Consider the two transition system TS_1 (left) and TS_2 (right) shown in Figure 7.59 where the colors stand for the state labels, e.g., we may deal with $L_1(s_i) = L_2(t_j) = \{a\}$ for $i \in \{1, 2, 3, 4\}$ and $j \in \{1, 2, 4\}$ and $L_1(s_6) = L_2(t_6) = \varnothing$ and $L_1(s_7) = L_1(s_5) = L_2(t_7) = \{b\}$.

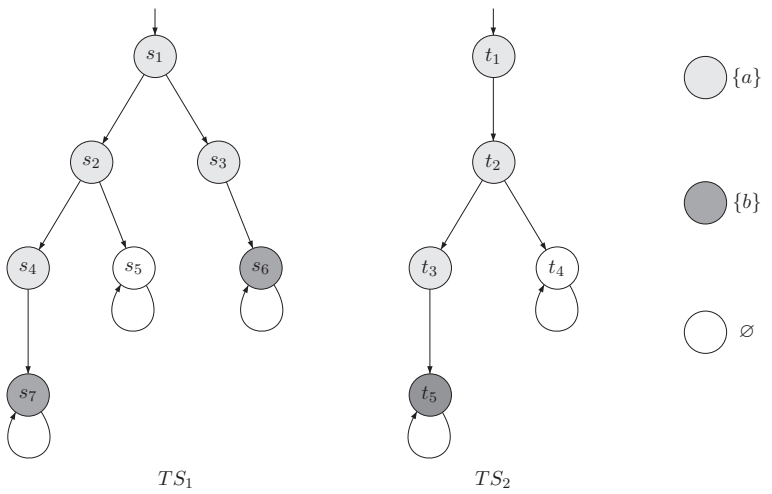


Figure 7.59: Transition systems TS_1 (left) and TS_2 (right) for Exercise 7.22.

Show that $TS_1 \not\approx TS_2$ and $TS_1 \approx_{obs} TS_2$.

EXERCISE 7.23. Consider the transition system TS shown in Figure 7.54 on page 585.

(a) Which states of TS are stutter-bisimilar (according to $\approx_{TS}$)? Depict the stutter bisimulation quotient system $TS/\approx$.

(b) Which states of TS are divergence stutter bisimilar? Justify the equivalence of states (according to $\approx_{TS}^{div}$) by providing a step-dependent norm function. Depict the quotient system $TS/\approx^{div}$ under stutter bisimulation equivalence with divergence.

(c) Depict the divergence-sensitive expansion $\overline{TS}$ and apply Algorithm 37 to compute the divergence stutter bisimulation equivalence classes.

(d) Provide $\mathrm{CTL}_{\setminus\bigcirc}$ master formulae for the divergence stutter bisimulation equivalence classes.

EXERCISE 7.24. Let TS be a transition system with state space S. Define functions $\nu_1^*, \nu_2^* : S \times S \to \mathbb{N}$ such that $(\approx_{TS}^n, \nu_1^*, \nu_2^*)$ is a normed bisimulation for TS.

EXERCISE 7.25. Provide an example for a transition system TS where $\approx_{TS}^n$ (normed bisimulation equivalence) is strictly finer than $\approx_{TS}^{div}$.

EXERCISE 7.26. Provide the proof for Lemma 7.96 (page 537).

EXERCISE 7.27. Let $\mathrm{CTL}_{\setminus \mathsf{U}}$ be the sublogic of CTL that does not permit the until operator. Similarly, $\mathrm{CTL}^*_{\setminus \mathsf{U}}$ means CTL without U. Which of the following statements are correct for finite transition systems?

(a) $\mathrm{CTL}_{\setminus \mathsf{U}}$ equivalence is finer than $\mathrm{CTL}_{\setminus\bigcirc}$ equivalence.

(b) $\mathrm{CTL}_{\setminus \mathsf{U}}$ equivalence is finer than divergence-sensitive stutter trace equivalence.

(c) $\mathrm{CTL}_{\setminus\bigcirc}$ equivalence is finer than $\mathrm{LTL}_{\setminus\bigcirc}$ equivalence.

(d) Divergence-sensitive stutter bisimulation equivalence is finer than $\mathrm{CTL}_{\setminus \mathsf{U}}$ equivalence.

(e) Stutter trace equivalence is finer than $\mathrm{CTL}_{\setminus \mathsf{U}}$ equivalence.

(f) For AP-deterministic transition systems, stutter trace equivalence is finer than trace-equivalence.

(g) For AP-deterministic transition systems, trace equivalence is finer than $\mathrm{CTL}^*_{\setminus \mathsf{U}}$ equivalence.

EXERCISE 7.28. Check the correctness of the following statement. If TS_1 and TS_2 are stutter-bisimilar transition systems over AP that are divergence-sensitive then $TS_1 \approx^{div} TS_2$. Provide either a proof or a counterexample.

EXERCISE 7.29. Let $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP, L_i)$, $i = 1, 2$, be two transition systems. A stutter simulation for (TS_1, TS_2) is a relation $\mathcal{R} \subseteq S_1 \times S_2$ such that

(A) Each initial state of TS_1 is $\mathcal{R}$-related to an initial state of TS_2, that is, $\forall s_1 \in I_1 \, \exists s_2 \in I_2 . \, (s_1, s_2) \in \mathcal{R}$.

(B) For all $(s_1, s_2) \in \mathcal{R}$ the following conditions (1) and (2) hold:

  (1) $L_1(s_1) = L_2(s_2)$.

  (2) If $s_1' \in \mathit{Post}(s_1)$ with $(s_1, s_1') \notin \mathcal{R}$, then there exists a finite path fragment $s_2 \, u_1 \, \ldots \, u_n \, s_2'$ with $n \geqslant 0$ and $(s_1, u_i) \in \mathcal{R}$, $i = 1, \ldots, n$ and $(s_1', s_2') \in \mathcal{R}$.

TS_1 is said to be stutter-simulated by TS_2, denoted $TS_1 \preceq_{st} TS_2$, iff there exists a stutter simulation for (TS_1, TS_2).

(a) Provide an example for transition systems TS_1, TS_2 such that TS_1 is not simulated by TS_2, but $TS_1 \preceq_{st} TS_2$.

(b) Provide an example for transition systems TS_1, TS_2 such that $TS_1 \trianglelefteq TS_2$ and $TS_1 \not\preceq_{st} TS_2$.

(c) Provide an example for transition systems TS_1, TS_2 such that $TS_1 \neg \trianglelefteq TS_2$ and $TS_1 \preceq_{st} TS_2$.

(d) Provide an example for transition systems TS_1, TS_2 such that $TS_1 \not\approx TS_2$, while $TS_1 \preceq_{st} TS_2$ and $TS_2 \preceq_{st} TS_1$.

A stutter simulation $\mathcal{R}$ for (TS_1, TS_2) is called divergence-sensitive if for all pairs $(s_1, s_2) \in \mathcal{R}$ and each infinite path fragment $\pi_1 = s_{0,1} \, s_{1,1} \, s_{2,1} \ldots$ in TS_1 with $s_{0,1} = s_1$ and $(s_{i,1}, s_2) \in \mathcal{R}$ for all $i \geqslant 0$ there exists a transition $s_2' \in \mathit{Post}(s_2)$ with $(s_{j,1}, s_2') \in \mathcal{R}$ for some $j \geqslant 1$. We write

$$TS_1 \preceq_{st}^{div} TS_2$$

iff there exists a divergence-sensitive stutter simulation $\mathcal{R}$ for (TS_1, TS_2).

(e) Provide an example for transition systems TS_1, TS_2 such that $TS_1 \not\approx^{div} TS_2$, while $TS_1 \preceq_{st}^{div} TS_2$ and $TS_2 \preceq_{st}^{div} TS_1$.

(f) Show that $TS_1 \approx^{div} TS_2$ if and only if there exists a divergence-sensitive stutter simulation $\mathcal{R}$ for (TS_1, TS_2) such that $\mathcal{R}^{-1} = \{ \, (s_2, s_1) \mid (s_1, s_2) \in \mathcal{R} \, \}$ is a divergence-sensitive stutter simulation for (TS_2, TS_1).

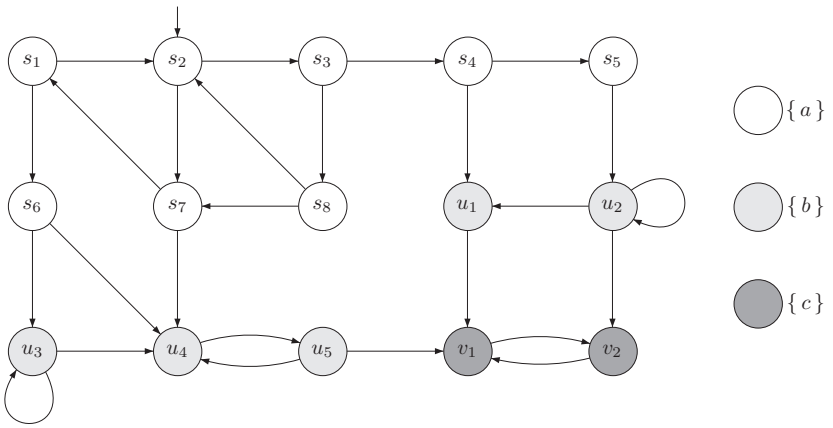
(g) Show that $TS_1 \approx^{div} TS_2$ implies divergence stutter simulation equivalence of TS_1 and TS_2, i.e., $TS_1 \preceq_{st}^{div} TS_2$ and $TS_2 \preceq_{st}^{div} TS_1$.

Show that the universal fragment of $\mathrm{CTL}^*_{\setminus\bigcirc}$ yields a logical characterization of the stutter simulation order with divergence. For this, provide proofs for the following statements (h) and (i) where TS_1 and TS_2 are supposed to be finite transition systems without terminal states.

(h) If $TS_1 \preceq_{st}^{div} TS_2$ and Φ is a $\forall\mathrm{CTL}^*_{\setminus\bigcirc}$ formula with $TS_2 \models \Phi$, then $TS_1 \models \Phi$.

(i) Assume that for all $\forall\mathrm{CTL}^*_{\setminus\bigcirc}$ formulae Φ we have $TS_2 \models \neg\Phi$ or $TS_1 \models \Phi$. Show that $TS_1 \preceq_{st}^{div} TS_2$.

Exercise 7.30. Consider the following transition system TS:



Questions:

(a) Give the divergence-sensitive expansion $\overline{TS}$.

(b) Determine the divergence stutter bisimulation quotient $(\overline{TS})/\approx$. Give for each iteration the partition of the state space.

(c) Give $TS/\approx^{div}$.

(d) Provide $\mathrm{CTL}_{\setminus\bigcirc}$ master formulae for each divergence stutter bisimulation equivalence class.

Chapter 8

Partial Order Reduction

Consider the parallel composition of a number of processes $\mathcal{P}_1$ through $\mathcal{P}_n$. The size of the state space of $\mathcal{P}_1 \| \mathcal{P}_2 \| \ldots \| \mathcal{P}_n$, where $\|$ denotes some parallel composition operator, is exponential in the number n of processes. To check the validity of a linear-time property of this system requires an inspection of all states in the underlying transition system. In the simple setting where there are no synchronizations between the individual processes—neither through shared variables nor via communication channels or the like—there are $n!$ different orderings of the interleaved execution of n local actions. The effect of concurrent actions, however, is often independent of their ordering. Consider, e.g., the assignments

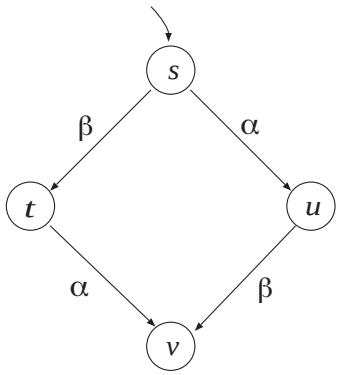


Figure 8.1: Interleaving diamond for $\alpha \,|||\, \beta$.

$x := x{+}1$ and $y := y{-}3$ in the concurrent system $\mathcal{P}_1 \,|||\, \mathcal{P}_2$, where x is a local variable of $\mathcal{P}_1$, say, and y of $\mathcal{P}_2$, and $|||$ denotes the interleaving operator. It is evident that regardless of the ordering of these assignments, the result will be the same. This is illustrated in Figure

8.1, where actions α and β denote the assignments of $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively. Instead of analyzing the 2! orderings of $x := x{+}1$ and $y := y{-}3$ it may suffice to check just a single ordering. This is correct as long as the intermediate states reached after the execution of either α or β (see states t and u in Figure 8.1), are irrelevant for the properties to be proved. Extending the simple example with a third process $\mathcal{P}_3$ that, e.g., resets its variable z to 0, yields following an analogous reasoning that it suffices to consider just one of the 3! possible orderings. This approach can be generalized for action sequences $\alpha_1\,\alpha_2\,\ldots\,\alpha_n$ and $\beta_1\,\beta_2\,\ldots\,\beta_m$ that are executed independently by processes $\mathcal{P}_1$ and $\mathcal{P}_2$. The transition system of $\mathcal{P}_1 \,|||\, \mathcal{P}_2$ represents all interleavings of these action sequences, whereas a single path fragment respecting the ordering in these sequences suffices, provided the intermediate states are irrelevant.

Put in a nutshell, the aim of partial order reduction, the technique that is treated in this chapter, is to reduce the number of possible orderings that need to be analyzed for checking formulae stated in a temporal logic such as LTL or CTL*. This main concept is to reduce the state space of the transition system that needs to be analyzed. Thus, the idea is to replace the full transition system for $\mathcal{P}_1 \,|||\, \mathcal{P}_2 \,|||\, \ldots \,|||\, \mathcal{P}_n$ by a small fragment. Figure 8.2 illustrates this for two processes that execute the action sequences $\alpha_1\,\alpha_2$ and $\beta_1\,\beta_2$, respectively. The transition system on the left contains all possible interleavings, while the reduced transition system on the right just consists a single path that might serve as a representative for all possible interleavings. On increasing the number of concurrent processes, this effect becomes even more drastic—the size of the full transition system grows exponentially in the number of processes, whereas the reduced system consists of a single path that grows linear in n.

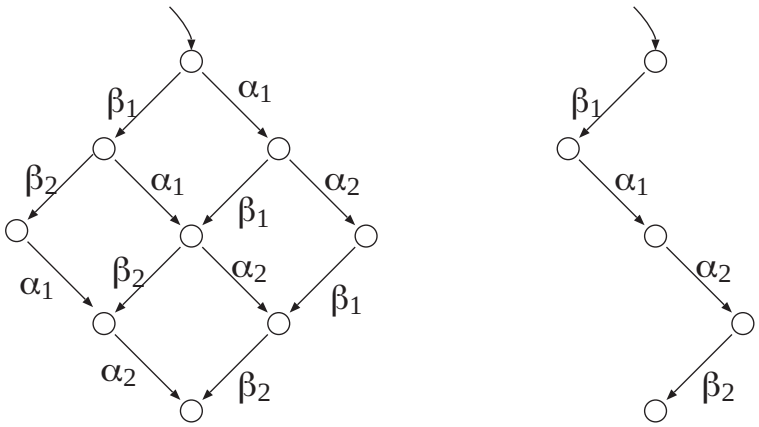


Figure 8.2: Parallel execution $(\alpha_1\,;\,\alpha_2) \,|||\, (\beta_1\,;\,\beta_2)$.

To avoid peak memory requirements, such a reduced transition system is obtained without

the necessity of ever generating the entire transition system that incorporates all possible orderings. This is typically done by a static analysis of the high-level description of the concurrent system, e.g., the channel systems as introduced in Chapter 2.

Clearly, the reduction technique crucially relies on the assumption that all processes are fully autonomous, i.e., no synchronizations are involved via, e.g., shared variables or communication channels. Moreover, it is assumed that the property of interest does not depend on the intermediate states. To treat realistic systems where the processes may communicate and thus depend on one another, the partial order reduction approach attempts to identify path fragments of the full transition system which only differ in the order of the concurrently executed activities. Since such path fragments represent the same behavior, it seems obvious to restrict the analysis of state space to one (or a few) representatives of every possible interleaving.

Central for the partial order reduction approach is the notion of independent actions which will be introduced in Section 8.1. Section 8.2 treats the partial order reduction approach for model-checking LT properties that are specified by $\mathrm{LTL}_{\setminus\bigcirc}$ formulae. Section 8.3 deals with partial order reduction for $\mathrm{CTL}_{\setminus\bigcirc}$ and $\mathrm{CTL}^*_{\setminus\bigcirc}$. The intuition behind the omission of the next step operator is that we have to abstract away from certain intermediate states, as we sketched for the reduction in Figure 8.2.

Throughout this chapter $TS = (S, Act, \rightarrow, I, AP, L)$ is a finite transition system without terminal states. As partial order reduction is most effective for concurrent systems that are "loosely" coupled, it is implicitly assumed that TS models an asynchronous concurrent system where processes interact, e.g., either by shared variables or channel communication. In a synchronous setting where concurrent processes evolve in a lockstep fashion, each global transition involves all processes and thus cannot be considered as independent. This assumption is not relevant for the theoretical considerations in this chapter, but is important when considering static analysis of concurrent programs to enable the detection of independent actions syntactically.

It is assumed that TS is *action-deterministic*. This entails that for any state $s \in S$ and any action $\alpha \in Act$, s has at most one outgoing transition with action label α. Formally, $s \xrightarrow{\alpha} s'$ and $s \xrightarrow{\alpha} s''$ implies $s' = s''$. (The reader should not confuse this notion with AP-determinism.) The assumption that TS is action-deterministic is not a severe restriction, since actions can always be renamed such that an action-deterministic transition system results. When, e.g., TS results from the parallel composition of several action-deterministic processes, indices may be used for the actions to indicate which process performs the action.

8.1 Independence of Actions

Let us first introduce some notations that are used throughout the remainder of this chapter.

Notation 8.1. The Set of Actions $Act(s)$

For state s in transition system TS with action set Act, let $Act(s) = \{\, \alpha \in Act \mid \exists s'.\, s \xrightarrow{\alpha} s' \,\}$. ■

Thus, $Act(s)$ denotes the set of actions that are *enabled* in state s. Since an action-deterministic transition system is assumed, for any $\alpha \in Act(s)$ there is a unique α-successor of s, denoted by $\alpha(s)$.

Notation 8.2. The States $\alpha(s)$

For action-deterministic transition system TS, s a state in TS and $\alpha \in Act(s)$, let $\alpha(s)$ denote the unique α-successor of s, i.e., $s \xrightarrow{\alpha} \alpha(s)$. For action sequence $\alpha_1 \ldots \alpha_n$ with $\alpha_1 \in Act(s)$ and $\alpha_{i+1} \in Act(s_i)$ where $s_i = \alpha_i(s_{i-1})$ for $1 < i \leqslant n$, $(\alpha_1 \ldots \alpha_n)(s)$ denotes s_n, the state that is reached from s by performing $\alpha_1 \ldots \alpha_n$. ■

In Figure 8.1, e.g., $Act(s) = \{\, \alpha, \beta \,\}$, $t = \beta(s)$ and $u = \alpha(s)$. Moreover, $v = (\beta\,\alpha)(s)$ and $v = (\alpha\,\beta)(s)$.

The notion of *independence of actions* plays a central role in partial order reduction. As we will describe later, the transition system TS is reduced by omitting the redundancies in TS that are caused by the different orderings of independent actions. Intuitively, the pair of actions α and β with $\alpha \neq \beta$ is independent when these actions access disjoint variables. In this case, the effect of executing $\alpha\,\beta$ or $\beta\,\alpha$ is the same. Actions α and β are also independent when they represent actions of different processes such that one of these actions only operates on local variables. Actions within a process may also be independent, viz. when their order of execution is irrelevant.

The characteristic feature of the independence of actions α and β is their *commutativity* which asserts that the effect of the execution orders $\alpha\,\beta$ and $\beta\,\alpha$ is the same. This means that if α and β are enabled in state s,

- the execution of action α cannot disable β, and vice versa, and
- the action sequences $\alpha\,\beta$ and $\beta\,\alpha$ executed in s yield the same state.

These properties are characteristic for the interleaving diamond (see Figure 8.1 on page 595).

Definition 8.3. Independence of Actions

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be an action-deterministic transition system with α, $\beta \in Act$, $\alpha \neq \beta$.

1. α and β are *independent* (in TS) if for any $s \in S$ with α, $\beta \in Act(s)$:

$$\beta \in Act(\alpha(s)) \quad \text{and} \quad \alpha \in Act(\beta(s)) \quad \text{and} \quad \alpha(\beta(s)) \;=\; \beta(\alpha(s)).$$

2. α and β are *dependent* (in TS) if α and β are not independent in TS.

■

As in most cases the transition system TS is fixed, the part (in TS) is often omitted. The notions of dependence and independence can be lifted to relations between actions and sets of actions as follows. For $A \subseteq Act$ and $\beta \in Act \setminus A$, β is independent of A (in TS) if for any $\alpha \in A$, β is independent of α (in TS). β is dependent on A in TS if $\beta \in Act \setminus A$ and β and α are dependent in TS for some $\alpha \in A$.

Example 8.4. Independence of Actions (Parallel Operator $\|_H$)

Let TS_1, TS_2 be two action-deterministic transition systems with action sets Act_1 and Act_2, respectively, and $H = Act_1 \cap Act_2$. The actions $\alpha \in Act_1 \setminus H$ and $\beta \in Act_2 \setminus H$ are independent in $TS_1 \| _H TS_2$. (The handshaking operator $\|_H$ has been defined on page 48.) When $H = \varnothing$, all actions of TS_1 are independent of the actions of TS_2. ■

Example 8.5. Independence of Actions (Program Graphs)

Consider the program graphs PG_1 and PG_2 and assume they do not communicate over channels. The actions α and β, by means of which at least one of the two processes accesses local variables only, are independent in $TS(PG_1 \,|||\, PG_2)$, the transition system underlying the parallel composition of PG_1 and PG_2. More precisely, we impose the following condition. Let α be an action only appearing in PG_1 such that for variable valuation η:

- $\mathit{Effect}(\alpha, \eta)(x) = \eta(x)$ for all variables x accessed by PG_2, and

- for any edge $\ell \overset{g:\alpha}{\hookrightarrow} \ell'$ in PG_1, guard g does not refer to the variables that appear in PG_2.

Under these conditions, α is independent of every action β in PG_2.

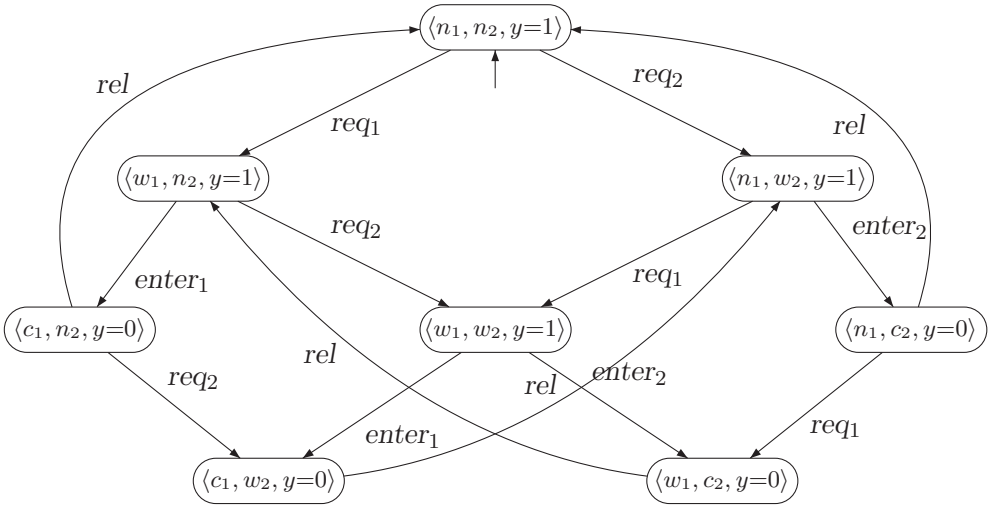


Figure 8.3: The semaphore-based mutual exclusion algorithm.

Let us illustrate this by means of the semaphore-based mutual exclusion program, see the transition system in Figure 8.3. The actions $\alpha = \mathit{request}_1$ and $\beta = \mathit{request}_2$ only appear in the following edges of the program graph:

$$\mathit{noncrit}_i \overset{\mathit{request}_i}{\hookrightarrow} \mathit{wait}_i.$$

In fact, α and β satisfy the above-mentioned conditions, and therefore are independent. This reflects the fact that the request operations—as well as all activities in the noncritical section—can be concurrently executed. On the other hand, the actions enter_i that are executed on entering the critical section,

$$\mathit{wait}_i \overset{y > 0:\ \mathit{enter}_i}{\hookrightarrow} \mathit{crit}_i,$$

access the shared variable y (the semaphore). Note that

$$\mathit{Effect}(\mathit{enter}_i, \eta)(y) \ = \ \eta(y) - 1 \quad \text{but} \quad \mathit{Effect}(\mathit{request}_i, \eta)(y) \ = \ \eta(y).$$

Therefore, the actions enter_1 and enter_2 are dependent, since e.g., for state $s = \langle w_1, w_2, y = 1\rangle$, we have

$$\mathit{enter}_1, \mathit{enter}_2 \in \mathit{Act}(s) \quad \text{and} \quad \mathit{enter}_2 \notin \mathit{Act}(\mathit{enter}_1(s)) = \mathit{Act}(\langle c_1, w_2, y = 0\rangle).$$

The pairs of actions $(request_1, enter_2)$, $(enter_1, request_2)$, $(rel, request_1)$ and $(rel, request_2)$ are independent. For instance, the only state in which both actions *rel* and $request_1$ are enabled is $s' = \langle n_1, c_2, y{=}0 \rangle$. We have

$$request_1 \in Act(rel(s')) \quad rel \in Act(request_1(s'))$$

and

$$rel(request_1(s')) = request_1(rel(s')) = \langle w_1, n_2, y{=}1 \rangle.$$

A similar reasoning applies to the other pairs of independent actions. ■

The following lemma is a central result for the partial order reduction approach. It relies on the successive exchange of independent actions β_i and α in the action sequence $\beta_1 \ldots \beta_n \alpha$.

Lemma 8.6. Permuting Independent Actions

Let TS be an action-deterministic transition system, s a state in TS, and let

$$s = s_0 \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} s_2 \xrightarrow{\beta_3} \ldots \xrightarrow{\beta_n} s_n$$

be an execution fragment from s with the action sequence $\beta_1 \ldots \beta_n$. Then, for any $\alpha \in Act(s)$ which is independent of $\{ \beta_1, \ldots, \beta_n \}$ we have $\alpha \in Act(s_i)$, and

$$s = s_0 \xrightarrow{\alpha} t_0 \xrightarrow{\beta_1} t_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_{n-1}} t_{n-1} \xrightarrow{\beta_n} t_n$$

is an execution fragment in TS with the action sequence $\alpha\, \beta_1 \ldots \beta_n$ such that $t_i = \alpha(s_i)$ for $0 \leqslant i \leqslant n$.

Proof: Let *TS* be action-deterministic transition system. By induction on $i \geqslant 1$ it is shown that

- α and β_{i+1} are enabled in state $s_i = (\beta_1 \ldots \beta_i)(s)$,
- β_i is enabled in state $t_{i-1} = (\alpha\, \beta_1 \ldots \beta_{i-1})(s)$, and
- $\alpha(s_i) = \beta_i(t_{i-1})$.

Base: $(i = 1)$. Let $\alpha, \beta_1 \in Act(s)$ and $s_1 = \beta_1(s)$ and $t_0 = \alpha(s)$. Since α and β_1 are independent, $\alpha \in Act(s_1)$ and $\beta_1 \in Act(t_0)$, and $\alpha(\beta_1(s)) = \beta_1(\alpha(s))$, i.e., $\alpha(s_1) = \beta_1(t_0)$.

Induction step: $(i{-}1 \Longrightarrow i$ for $1 < i \leqslant n)$. Assume α and β_i are enabled in state $s_{i-1} = (\beta_1 \ldots \beta_{i-1})(s)$, β_{i-1} is enabled in state $t_{i-2} = (\alpha\, \beta_1 \ldots \beta_{i-2})(s)$, and the α-successor of s_{i-1} and the β_{i-1}-successor of t_{i-2} agree, i.e.:

$$t_{i-1} = \alpha(s_{i-1}) = \beta_{i-1}(t_{i-2}).$$

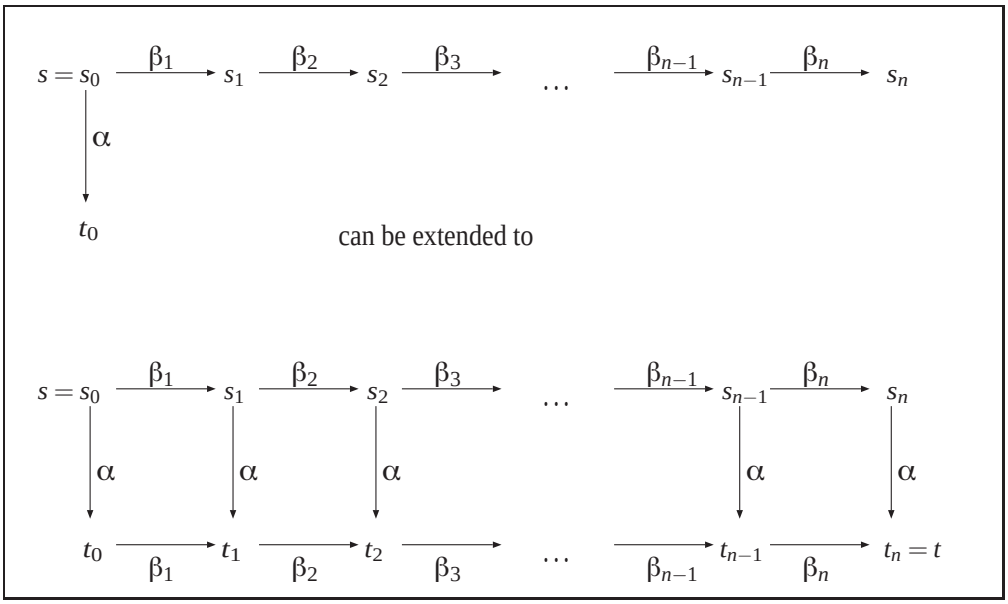


Figure 8.4: Permuting α with the independent actions β_1 through β_n.

The independence of α and β_i yields that α is enabled in state $\beta_i(s_{i-1}) = s_i$, β_i is enabled in state $\alpha(s_{i-1}) = \beta_{i-1}(t_{i-2}) = t_{i-1}$, and $\alpha(s_i) = \beta_i(t_{i-1}) = t_i$. ■

Lemma 8.6 is illustrated in Figure 8.4.

Consider the infinite execution fragment ρ starting in s with the action sequence $\beta_1\,\beta_2\,\beta_3\ldots$ and an action $\alpha \in Act(s)$ that is independent on all β_j's (in particular, $\alpha \neq \beta_j$ for all $j > 0$). Then Lemma 8.6 applied to the finite prefixes of ρ yields the existence of finite execution fragments from s with the action sequences $\alpha\,\beta_1\,\beta_2\ldots\beta_n$ for all $n > 0$. Since TS is action-deterministic, we obtain an infinite execution fragment which first executes action α and then the infinite action sequence $\beta_1\,\beta_2\,\beta_3\ldots$. More precisely, we have:

Lemma 8.7. Adding an Independent Action

Let TS be an action-deterministic transition system, s a state in TS, and let

$$s = s_0 \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} s_2 \xrightarrow{\beta_3} \ldots$$

be an infinite execution fragment from s with the action sequence $\beta_1\,\beta_2\ldots$. Then, for $\alpha \in Act(s)$ which is independent of $\{\,\beta_1, \beta_2, \ldots\,\}$ we have $\alpha \in Act(s_i)$ for all $i \geqslant 0$ and

$$s = s_0 \xrightarrow{\alpha} t_0 \xrightarrow{\beta_1} t_1 \xrightarrow{\beta_2} t_2 \xrightarrow{\beta_3} \ldots$$

is an infinite execution fragment in TS with the action sequence $\alpha\,\beta_1\,\beta_2\ldots$ *and* $t_i = \alpha(s_i)$ *for* $i \geqslant 0$.

If no further assumptions are made, the traces induced by the execution fragments

$$s_0 \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_n} s_n \xrightarrow{\alpha} t, \text{ and}$$
$$s_0 \xrightarrow{\alpha} t_0 \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_{n-1}} t_{n-1} \xrightarrow{\beta_n} t$$

will be distinct and not related via any form of (stutter) trace equivalence or (stutter) trace inclusion. However, if the action α which is moved from the "right" to the "left" (by successive swapping the order $\beta_i\,\alpha$ into $\alpha\,\beta_i$) does not affect the state labeling, then the execution fragments are stutter-equivalent. Recall that execution fragments ϱ, ϱ' are stutter-equivalent, denoted $\varrho \triangleq \varrho'$, if their traces only differ in the number of repetitions of state labels; see Section 7.7.1. These actions are called stutter actions or invisible actions.

Definition 8.8. Stutter Action

The action $\alpha \in Act$ is a *stutter action* if for each transition $s \xrightarrow{\alpha} s'$ in transition system TS we have $L(s) = L(s')$. ■

Since action-deterministic transition systems are considered, α is a stutter action in TS if and only if $L(s) = L(\alpha(s))$ for all states s in TS with $\alpha \in Act(s)$. For example, the actions β and γ in the transition system in Figure 8.5 are stutter actions, while α is not a stutter action.

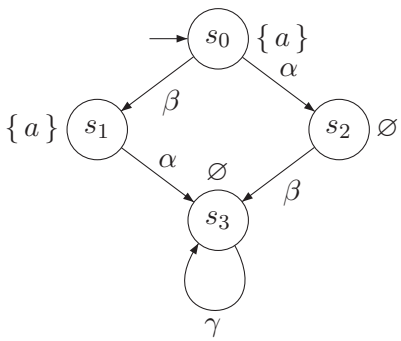


Figure 8.5: Actions β and γ are stutter actions.

Remark 8.9. Stutter Steps vs. Stutter Actions

Let us briefly explain the difference between stutter steps and stutter actions. Recall that

a stutter step is a transition $s \to t$ such that $L(s) = L(t)$. The fact whether or not an α-labeled transition is a stutter step depends on the state in which α is executed. For example, let α be the action corresponding to the assignment $x := 2 * x$ for some integer variable x. In all states in which x evaluates to 0, α does not affect the value of x. If the atomic propositions only consider the values of x (and other program variables), but not the program locations, then the transitions

$$\langle \ldots, x = 0 \rangle \xrightarrow{x:=2*x} \langle \ldots, x = 0 \rangle$$

are stutter steps, whereas the transitions

$$\langle \ldots, x = v \rangle \xrightarrow{x:=2*x} \langle \ldots, x = 2v \rangle$$

for $v \neq 0$ are not. Action α is a stutter action whenever all transitions $s \xrightarrow{\alpha} s'$ are stutter steps. ■

Lemma 8.10. Permuting Independent Stutter Actions

Let TS be an action-deterministic transition system, s a state in TS, and ϱ and ϱ' be finite execution fragments starting in s with action sequences $\beta_1 \ldots \beta_n \, \alpha$ and $\alpha \, \beta_1 \ldots \beta_n$, respectively, such that α is a stutter action which is independent of $\{ \beta_1, \ldots, \beta_n \}$. Then $\varrho \triangleq \varrho'$.

Proof: Let

$$\begin{array}{lllllllllll}
\varrho = s_0 & \xrightarrow{\beta_1} & s_1 & \xrightarrow{\beta_2} & s_2 & \xrightarrow{\beta_3} & \ldots & \xrightarrow{\beta_n} & s_n & \xrightarrow{\alpha} & v \\
\varrho' = s_0 & \xrightarrow{\alpha} & t_0 & \xrightarrow{\beta_1} & t_1 & \xrightarrow{\beta_2} & \ldots & \xrightarrow{\beta_{n-1}} & t_{n-1} & \xrightarrow{\beta_n} & t_n,
\end{array}$$

where $s = s_0$. By Lemma 8.6, ϱ and ϱ' end in the same state, i.e., $t_n = v$, and $\alpha(s_i) = t_i$ for $0 \leqslant i \leqslant n$. Since α is a stutter action:

$$L(s_i) = L(\alpha(s_i)) = L(t_i) \quad \text{for} \quad 0 \leqslant i \leqslant n.$$

Let $A_i = L(s_i)$, $0 \leqslant i \leqslant n$. Then:

$$\begin{array}{rcccl}
\mathit{trace}(\varrho) & = & L(s_0)\, L(s_1) \ldots L(s_n)\, L(t_n) & = & A_0\, A_1 \ldots A_n\, A_n \text{ and} \\
\mathit{trace}(\varrho') & = & L(s_0)\, L(t_0)\, L(t_1) \ldots L(t_n) & = & A_0\, A_0\, A_1 \ldots A_n.
\end{array}$$

Thus, both traces have the form $A_0^+ A_1 \ldots A_{n-1} A_n^+$. Hence, $\varrho \triangleq \varrho'$. ■

The following lemma describes a transformation of an infinite execution fragment with the action sequence $\beta_1 \, \beta_2 \, \beta_3 \ldots$ into a stutter-equivalent execution fragment with the action sequence $\alpha \, \beta_1 \, \beta_2 \, \beta_3 \ldots$ where α is a stutter action that is independent of any β_i.

Lemma 8.11. Adding an Independent Stutter Action

Let TS be an action-deterministic transition system, s a state in TS, and ρ and ρ' be infinite execution fragments starting in s with the action sequences $\beta_1\,\beta_2\,\beta_3\ldots$ and $\alpha\,\beta_1\,\beta_2\,\beta_3\ldots$, respectively, such that α is a stutter action which is independent of $\{\,\beta_1,\beta_2,\beta_3,\ldots\,\}$. Then $\rho \stackrel{\triangle}{=} \rho'$.

Proof: Let $\rho = s_0 \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} s_2 \xrightarrow{\beta_3} \ldots$ and $\rho' = s_0 \xrightarrow{\alpha} t_0 \xrightarrow{\beta_1} t_1 \xrightarrow{\beta_2} t_2 \xrightarrow{\beta_3} \ldots$ where $s = s_0$. Then, $s_i = \alpha(t_i)$ for all $i \geqslant 0$. Since α is a stutter action we have $L(s_i) = L(t_i)$ for all $i \geqslant 0$. With $A_i = L(s_i)$ we get

$$\begin{array}{lcll} \mathit{trace}(\rho) & = & L(s_0)\,L(s_1)\,L(s_2)\,\ldots & = \quad A_0\,A_1\,A_2\ldots \\ \mathit{trace}(\rho') & = & L(s_0)\,L(t_0)\,L(t_1)\,L(t_2)\ldots & = \quad A_0\;\,A_0\,A_1\,A_2\ldots \end{array}$$

Thus, both traces have the form $A_0^+ A_1 A_2 \ldots$ which yields $\rho \stackrel{\triangle}{=} \rho'$. ■

Lemmas 8.10 and 8.11 yield the basis of the partial order reduction approach. During partial order reduction, any stutter equivalence class of executions in the full system *TS* is represented by at least one execution in the reduced system $\hat{TS}$. (One might say that partial order reduction amounts to model checking using representative executions.) The representatives in $\hat{TS}$ of *TS*'s stutter equivalence classes arise by permuting independent actions and adding independent stutter actions.

8.2 The Linear-Time Ample Set Approach

We consider partial order reduction for LTL using so-called *ample sets*. The basic idea is the following. Consider a high-level specification of an asynchronous system. Using traditional state space generation, for each encountered states all direct successors are explored. That is, for each action $\alpha \in \mathit{Act}(s)$, the successor state $\alpha(s)$ is determined, and when encountered for the first time, generated. With partial order reduction using ample sets, the set $\mathit{ample}(s) \subseteq \mathit{Act}(s)$ will be explored instead of the entire set $\mathit{Act}(s)$. That is, all direct successors in $\mathit{Act}(s) \setminus \mathit{ample}(s)$ are not explored, and possibly not generated at all. By choosing appropriate action sets $\mathit{ample}(\cdot)$, this approach yields a—hopefully small—fragment of the full transition system $TS = (S, \mathit{Act}, \rightarrow, I, AP, L)$. As *TS* will never be generated, the peak memory requirements are determined by the size of the fragment $\hat{TS}$ rather than by *TS*. The reduced transition system $\hat{TS}$ results from the transition relation $\Longrightarrow$ which is defined by

$$\frac{s \xrightarrow{\alpha} s' \;\wedge\; \alpha \in \mathit{ample}(s)}{s \overset{\alpha}{\Longrightarrow} s'}.$$

More precisely, $\hat{TS} = (\hat{S}, Act, \Longrightarrow, I, AP, \widehat{L})$ where the state space $\hat{S}$ consists of those states that are reachable (under $\Longrightarrow$) from some initial state $s_0 \in I$ and $\widehat{L}(s) = L(s)$ for any $s \in \hat{S}$. Thus, $\hat{S}$ might be a proper subset of the original state space S. The following correctness criterion needs to be established:

(1) $\hat{TS}$ and TS are equivalent with respect to the formulae to be checked.

It turns out that stutter trace equivalence is an appropriate notion of equivalence for partial order reduction when checking LT properties; for branching-time properties, divergence-sensitive stutter bisimulation is convenient. Besides this formal soundness criterion, the following more informal requirements are of importance:

(2) $\hat{TS}$ should be considerably smaller (and therefore more efficient to analyze) than TS.

(3) The effort to generate $\hat{TS}$ should be relatively low compared to the verification of TS.

To fulfill the last constraint, we aim at an algorithm to generate $\hat{TS}$ with a time complexity that is linear in the size of $\hat{TS}$. Typically, $\hat{TS}$ is obtained by a static analysis of a high-level description of TS, e.g., as a channel system or sequential program graph. If $\hat{TS}$ is significantly smaller than TS, then such an approach can be viewed as efficient.

The following subsections treat the ample-set method for verifying LT properties specified as LTL formulae. It is assumed that the original transition system $TS = (S, Act, \rightarrow, I, AP, L)$ is finite, action-deterministic, and does not have terminal states. The aim is to replace the verification whether $TS \models \varphi$, for φ an LTL formula over AP, by checking whether $\hat{TS} \models \varphi$. $\hat{TS}$ is obtained from TS by choosing appropriate ample sets. We first deal with LT properties; in Section 8.3, this approach will be extended to the verification of branching-time properties stated in CTL*.

8.2.1 Ample Set Constraints

To ensure that TS and $\hat{TS}$ are equivalent, the ample sets have to fulfill a number of conditions. As we will see, these constraints ensure that $TS \triangleq \hat{TS}$, i.e., TS and $\hat{TS}$ are stutter trace equivalent (see Section 7.7.1, see page 530 and further). Since stutter trace equivalence preserves all *stutter-insensitive* LT properties, the reduction of TS to $\hat{TS}$ is sound for such LT properties. The fragment of LTL that does not contain the next step

operator, $\mathrm{LTL}_{\setminus\bigcirc}$, is an appropriate logical formalism to specify stutter-insensitive LT properties, and thus

$$\hat{TS} \models \varphi \quad \text{if and only if} \quad TS \models \varphi$$

for any $\mathrm{LTL}_{\setminus\bigcirc}$-formulae φ, see Corollary 7.93 (page 535). (As the next step operator is not very useful for asynchronous systems, the kind of systems that partial order reduction is aimed at, the absence of the $\bigcirc$ operator is not a severe restriction.)

Given a technique to generate the ample sets, such that $TS \triangleq \hat{TS}$, standard LTL model-checking techniques may be applied to check $\hat{TS} \models \varphi$ (and thus $TS \models \varphi$). There are two main approaches to the computation of the ample sets, i.e., for determining $\hat{TS}$: *static* vs. *dynamic* partial order reduction. In dynamic (or on-the-fly) partial order reduction, the reduced transition system $\hat{TS}$ is generated *during* the LTL model checking of $\hat{TS}$. The main advantage of this approach is that $\hat{TS} \not\models \varphi$ might be established without the need for generating the entire transition system $\hat{TS}$—only the relevant part of the product $\hat{TS} \otimes \mathcal{A}_{\neg\varphi}$, where $\mathcal{A}_{\neg\varphi}$ is the Büchi automaton for $\neg\varphi$, is needed to show the refutation of φ. This approach is treated in Section 8.2.3. In the static approach, a symbolic representation (e.g., in terms of program graphs) of $\hat{TS}$ is generated *prior to* the verification. The reduction of the transition system may be viewed as a preprocessing phase of model checking. The static approach is treated in Section 8.2.4.

In the remainder of this section, conditions for the ample sets are established that ensure $TS \triangleq \hat{TS}$. Subsequently, techniques will be discussed to algorithmically determine appropriate ample sets.

Since every execution in $\hat{TS}$ is an execution in TS, we establish sufficient conditions to assign to each execution ρ_0 in TS a stutter-equivalent execution $\hat{\rho}$ in $\hat{TS}$. In order to avoid any confusion, we stress that this assignment of executions in TS to executions in $\hat{TS}$ is *not* part of the algorithm for generating $\hat{TS}$, but is only needed to prove the stutter trace equivalence of TS and $\hat{TS}$. The basic idea for the transformation

$$\text{“execution } \rho_0 \text{ in } TS \mapsto \hat{\rho}_0 \text{ in } \hat{TS} \text{ with } \rho_0 \triangleq \hat{\rho}_0 \text{”}$$

is the following. Let ρ_0 be an infinite execution in TS that is not an execution in $\hat{TS}$. A stutter-equivalent execution $\hat{\rho}_0$ in $\hat{TS}$ will be obtained from ρ_0 by successively permuting the order of independent actions and possibly adding stutter steps, according to the transformations described in cases 1 and 2 (see below). The correctness of these transformations is ensured by Lemmas 8.10 and 8.11. These transformations allow the replacement of ρ_0 through a possibly infinite *series of transformations*:

$$\rho_0 \mapsto \rho_1 \mapsto \rho_2 \mapsto \ldots \mapsto \hat{\rho}_0 \quad \text{where } \rho_i \triangleq \rho_0 \text{ for all } i \geqslant 0$$

such that at least the first i steps in ρ_i are transitions (i.e., according to $\Longrightarrow$) in $\hat{TS}$ and

agree with the first i transitions of ρ_j for all $j > i$. That is, $\rho_i[..i] = \rho_j[..i]$ for all $j > i$. In this way, an execution $\hat{\rho}_0$ in $\hat{TS}$ is obtained which, for any $i \geqslant 0$, agrees with ρ_i for the first i transitions. It can be regarded as the "limit" of the sequence ρ_0, ρ_1, ρ_2, and so on.

Let us consider the transformation of ρ_i to ρ_{i+1} in more detail. For simplicity consider the case i=0. Let m be the minimal index in ρ_0 such that $s = \rho_0[m] \xrightarrow{\alpha} \rho_0[m{+}1]$ for $\alpha \notin \mathit{ample}(s)$. Execution ρ_0 thus consists of a finite prefix ϱ_0 containing transitions in $\hat{TS}$ that ends in state s, and the infinite execution fragment ρ starting in s with action sequence $\beta_1\,\beta_2\ldots$, say, such that $\beta_1 \notin \mathit{ample}(s)$:

$$\rho_0 \;=\; \underbrace{u \xRightarrow{\gamma_1} \ldots \xRightarrow{\gamma_m} s}_{\text{prefix } \varrho_0} \qquad \underbrace{s \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} s_2 \ldots}_{\text{suffix } \rho \text{ with } \beta_1 \notin \mathit{ample}(s)} \qquad \text{for } m \geqslant 0.$$

Execution ρ_1 then starts with the prefix ϱ_0 and continues with the infinite execution fragment ρ' that is obtained from ρ according to one of the transformations described below.

Case 1: There exists $n > 0$ such that $\alpha = \beta_{n+1} \in \mathit{ample}(s)$ and $\beta_1, \ldots, \beta_n \notin \mathit{ample}(s)$, i.e., some action of the suffix $\rho = s \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \ldots$ of ρ_0 belongs to $\mathit{ample}(s)$. The conditions imposed on the ample sets will ensure that α is a stutter action that is independent of $\{\,\beta_1, \ldots, \beta_n\,\}$. The execution fragment ρ' (i.e., the suffix of ρ_1) results from ρ by replacing the action sequence $\beta_1 \ldots \beta_n\,\alpha$ with $\alpha\,\beta_1 \ldots \beta_n$. That is, the action α is shifted to occur prior to any β_i ($0 < i \leqslant n$). This transformation is described by Lemma 8.6 (page 601). Pictorially this amounts to

$$\begin{aligned}
\rho_0 = \;& u \xRightarrow{\gamma_1} \ldots \xRightarrow{\gamma_m} \quad s \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_n} s_n \xrightarrow{\alpha} \quad t \xrightarrow{\beta_{n+2}} s_{n+2} \xrightarrow{\beta_{n+3}} \ldots \\
\rho_1 = \;& u \xRightarrow{\gamma_1} \ldots \xRightarrow{\gamma_m} \quad s \xRightarrow{\alpha} t_0 \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_n} \quad t \xrightarrow{\beta_{n+2}} s_{n+2} \xrightarrow{\beta_{n+3}} \ldots
\end{aligned}$$

common prefix ϱ_0 — stutter-equivalent execution fragments — common suffix

Since α is a stutter action, $\rho \triangleq \rho'$ (see Lemma 8.10 on page 604), and thus $\rho_0 \triangleq \rho_1$.

Case 2: For all $i > 0$, $\beta_i \notin \mathit{ample}(s)$, i.e., none of the actions occurring in the suffix ρ belongs to $\mathit{ample}(s)$. The conditions on the ample sets will ensure the independence of β_i and $\mathit{ample}(s)$ for any i, and that any $\alpha \in \mathit{ample}(s)$ is a stutter action. The execution fragment ρ' then results by replacing ρ by the execution fragment that starts in s and successively executes the actions $\alpha\,\beta_1\,\beta_2\,\beta_3\ldots$, for some $\alpha \in \mathit{ample}(s)$.

Schematically:

$$\begin{array}{lcl}
\rho_0 & = & u \xRightarrow{\gamma_1} \ldots \xRightarrow{\gamma_m} s \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} s_2 \xrightarrow{\beta_3} \ldots \\
\rho_1 & = & u \xRightarrow{\gamma_1} \ldots \xRightarrow{\gamma_m} s \xRightarrow{\alpha} t_0 \xrightarrow{\beta_1} t_1 \xrightarrow{\beta_2} t_2 \xrightarrow{\beta_3} \ldots
\end{array}$$

common prefix ϱ_0 — stutter-equivalent execution fragments

Since α is a stutter action, Lemma 8.11 yields $\rho \triangleq \rho'$ and thus $\rho_0 \triangleq \rho_1$.

In both cases, an execution fragment ρ_1 is obtained that starts in the same state as ρ_0 such that the first transition that is not a transition in $\hat{TS}$ occurs at some position $\geqslant 2$. This recipe is applied iteratively to $\rho_j \mapsto \rho_{j+1}$ for $j \geqslant 1$. This yields a stutter-equivalent execution fragment ρ_{j+1} such that the first transition that is not a transition in $\hat{TS}$ (i.e., not an ample transition) occurs at some position $> j$ and $\rho_j[..j{+}1] = \rho_{j+1}[..j{+}1]$. Continuing in this way, we finally obtain an execution fragment in $\hat{TS}$.

To ensure that the above-described transformations (i.e., cases 1 and 2) are applicable and yield an execution $\hat{\rho}_0$ in $\hat{TS}$ such that $\rho_0 \triangleq \hat{\rho}_0$ with ρ_0 in TS, four conditions are imposed on ample sets referred to as (A1) through (A4). Conditions (A1) through (A3) are imposed on each state s in $\hat{S}$; no restrictions are imposed on states that are unreachable via $\Longrightarrow$. Condition (A4) is imposed on all cycles in $\hat{TS}$.

Let us consider the constraints in more detail.

(A1) Nonemptiness condition

$\varnothing \neq \mathit{ample}(s) \subseteq \mathit{Act}(s)$

The first condition asserts that if a state has at least one direct successor in TS, then it has least one direct successor in $\hat{TS}$. As TS does not have terminal states, condition (A1) ensures that $\hat{TS}$ does not have any terminal states.

(A2) Dependency condition

Let $s \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_n} s_n \xrightarrow{\alpha} t$ be a finite execution fragment in TS. If α depends on $\mathit{ample}(s)$, then $\beta_i \in \mathit{ample}(s)$ for some $0 < i \leqslant n$.

The key condition for the correctness is the dependency condition (A2). It asserts that in every (!) finite execution fragment of TS, an action depending on $\mathit{ample}(s)$ cannot occur before some action from $\mathit{ample}(s)$ is occurring first. Note that this condition is imposed

on every (finite) execution of the original transition system TS. We will see later that this condition ensures that for any state s which is not fully expanded (i.e., $\mathit{ample}(s)$ is a proper subset of $\mathit{Act}(s)$), all ample actions $\alpha \in \mathit{ample}(s)$ are independent of $\mathit{Act}(s) \setminus \mathit{ample}(s)$. Note that for $n{=}0$, condition (A2) is false, as the existential quantification (over i) ranges over an empty domain.

Condition (A2) guarantees that any finite execution in TS is of the form

$$\varrho \;=\; s \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_n} s_n \xrightarrow{\alpha} t \quad \text{with} \quad \alpha \in \mathit{ample}(s)$$

and β_i independent of $\mathit{ample}(s)$ for $0 < i \leqslant n$. If α is a stutter action—which will be guaranteed by constraint (A3)—the execution obtained by shifting α to the beginning is also an execution of TS; see the transformation as described under case 1 above. That is to say, if ϱ is pruned in TS, i.e., does not occur in $\hat{TS}$ as $\beta_1 \notin \mathit{ample}(s)$, then a stutter-equivalent execution can be constructed by performing α in s. Infinite executions in TS are of the form

$$s_1 \xrightarrow{\beta_1} s_2 \xrightarrow{\beta_2} \ldots \quad \text{with } \beta_i \text{ independent of } \mathit{ample}(s) \text{ for } 0 < i \leqslant n.$$

For stutter action $\alpha \in \mathit{ample}(s)$, inserting the action α at the beginning of this execution yields another execution of TS; see the transformation as described under case 2 above.

(A3) Stutter condition

If $\mathit{ample}(s) \neq \mathit{Act}(s)$ then any $\alpha \in \mathit{ample}(s)$ is a stutter action.

Stutter condition (A3) ensures that the transformations (cases 1 and 2) generate stutter-equivalent executions; see Lemma 8.10 (page 604) and Lemma 8.11 (page 604). To be more precise, condition (A3) ensures that the switch from action sequence $\beta_1 \ldots \beta_n \alpha$ to $\alpha \beta_1 \ldots \beta_n$, and from $\beta_1 \beta_2 \beta_3 \ldots$ to $\alpha \beta_1 \beta_2 \beta_3 \ldots$ in state s with $\alpha \in \mathit{ample}(s)$ yields stutter-equivalent executions. Ample actions may thus be executed first.

(A4) Cycle condition

For any cycle $s_0\, s_1 \ldots s_n$ in $\hat{TS}$ and $\alpha \in \mathit{Act}(s_i)$, for some $0 < i \leqslant n$, there exists $j \in \{1, \ldots, n\}$ such that $\alpha \in \mathit{ample}(s_j)$.

The cycle condition (A4) is the final condition that is needed to ensure that TS and $\hat{TS}$ are stutter-equivalent. The justification of this condition is provided later.

The conditions imposed on ample sets are summarized in Figure 8.6.

(A1) **Nonemptiness condition**

$\varnothing \neq \mathit{ample}(s) \subseteq \mathit{Act}(s)$

(A2) **Dependency condition**

Let $s \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_n} s_n \xrightarrow{\alpha} t$ be a finite execution fragment in TS.
If α depends on $\mathit{ample}(s)$, then $\beta_i \in \mathit{ample}(s)$ for some $0 < i \leqslant n$.

(A3) **Stutter condition**

If $\mathit{ample}(s) \neq \mathit{Act}(s)$ then any $\alpha \in \mathit{ample}(s)$ is a stutter action.

(A4) **Cycle condition**

For any cycle $s_0\, s_1\, \ldots\, s_n$ in $\hat{TS}$ and $\alpha \in \mathit{Act}(s_i)$, for some $0 < i \leqslant n$, there exists $j \in \{1, \ldots, n\}$ such that $\alpha \in \mathit{ample}(s_j)$.

Figure 8.6: Requirements on the ample set of state s.

Example 8.12. Ample Set Conditions

Consider the transition system TS in Figure 8.7 (left part) over $AP = \{a\}$. Action β is a stutter action, and is independent of $\{\alpha, \gamma, \delta\}$. Let $\mathit{ample}(s_0) = \{\beta\}$. This choice satisfies constraints (A1) through (A3). Consider now state s_2. The choice $\mathit{ample}(s_2) = \{\alpha\}$ violates (A3), as α is not a stutter action. $\mathit{ample}(s_2) = \{\delta\}$ violates the cycle condition (A4): the reduced transition system $\hat{TS}$ would contain the cycle $s_0\, s_2\, s_2$ with $\alpha \in \mathit{Act}(s_2)$, but $\alpha \notin \mathit{ample}(s_2)$. Thus, we select $\mathit{ample}(s_2) = \{\alpha, \delta\}$. The nonemptiness condition (A1) then leaves no freedom for s_3: $\mathit{ample}(s_3) = \{\gamma\}$. The resulting reduced transition system $\hat{TS}$ is depicted in Figure 8.7 (right part).

The traces of TS and $\hat{TS}$ are either of the form $(\varnothing^+\{a\}^+)^\omega$ or $(\varnothing^+\{a\}^+)^*\varnothing^\omega$. Hence, $TS \triangleq \hat{TS}$. ■

We now state the main result of this section. The proof of this result is provided by a series of lemmas, presented in the remainder of this section.

Theorem 8.13. Correctness of the Ample Set Approach

Let TS be an action-deterministic, finite transition system without terminal states. Then

$$\textit{if conditions (A1) through (A4) are satisfied, then } \hat{TS} \triangleq TS.$$

This theorem asserts that whenever $\hat{TS}$ is constructed from TS using ample sets that

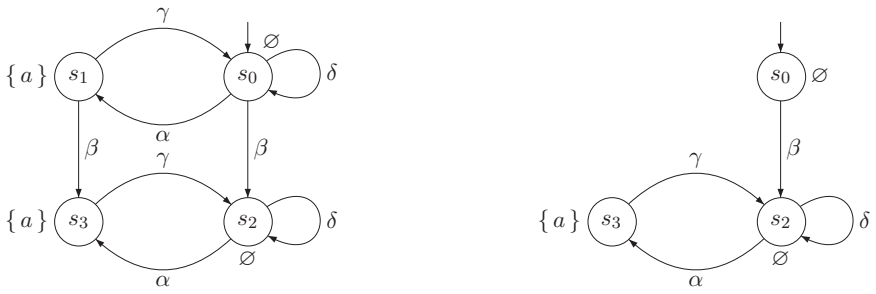


Figure 8.7: Transition sytem TS (left) and $\hat{TS}$ (right)

satisfy all constraints (A1) through (A4), then $\hat{TS}$ and TS are stutter trace equivalent. Since stutter trace equivalence is finer than $\mathrm{LTL}_{\setminus\bigcirc}$ equivalence, conditions (A1) through (A4) guarantee that $\hat{TS}$ and TS satisfy the same $\mathrm{LTL}_{\setminus\bigcirc}$ formulae. The remainder of this subsection is devoted to the proof of Theorem 8.13. Since $\hat{TS}$ is a fragment of TS, every execution of $\hat{TS}$ is an execution of TS. It remains to show that, conversely, for every execution in TS there exists a stutter-equivalent execution in $\hat{TS}$. The proof for this statement proceeds in a number of steps, and results from Lemmas 8.14 through 8.21.

The first two lemmas follow from simple observations that are based on the dependency condition (A2).

Lemma 8.14. Independence of Ample and Other Enabled Actions
Let $s \in Reach(TS)$ and $\alpha \in ample(s)$. Then

(A2) implies that α is independent of $Act(s) \setminus ample(s)$.

Proof: Let $s \in Reach(TS)$, $\alpha \in ample(s)$, and $\beta \in Act(s) \setminus ample(s)$. Then there exists an execution fragment starting in s which begins with action β:

$$\rho \;=\; s \xrightarrow{\beta} s_1 \xrightarrow{\gamma_1} s_2 \xrightarrow{\gamma_2} s_3 \xrightarrow{\gamma_3} \ldots$$

Assume $ample(s)$ satisfies (A2) and assume α and β are dependent. The existence of the execution ρ, however, violates condition (A2) since β is not preceded by an action in $ample(s)$. Contradiction. ∎

(In Remark 8.19 (page 614) it is shown that just imposing that any $\alpha \in ample(s)$ is independent of $Act(s) \setminus ample(s)$ is too weak to establish that $TS \triangleq \hat{TS}$.)

The following lemma asserts that for any execution fragment that starts in s it holds that any action in $ample(s)$ remains enabled as long as no action in $ample(s)$ has been executed.

Lemma 8.15. Enabledness of Ample Actions

Let $s \in Reach(TS)$ and $s = s_0 \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_n} s_n$ be a finite execution fragment in TS. If $ample(s)$ satisfies (A2) and $\{\beta_1, \ldots, \beta_n\} \cap ample(s) = \varnothing$, then for all actions $\alpha \in ample(s)$ are independent of $\{\beta_1, \ldots, \beta_n\}$. In adddition, we have $\alpha \in Act(s_i)$ for $0 < i \leqslant n$.

Proof: Let $s \in Reach(TS)$ and $s = s_0 \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_n} s_n$ be a finite execution fragment, ρ say, in TS. Consider $\alpha \in ample(s)$ and assume $ample(s)$ satisfies constraints (A1) and (A2). The proof is by induction on i.

Basis: (i=1). By Lemma 8.14, α and β_1 are independent. But then also, $\alpha \in Act(\beta(s_0)) = Act(s_1)$.

Induction step: Suppose the claim holds for $0 < i < n$. Consider the execution fragment $\rho_i = s_0 \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_i} s_i$. As ρ is an execution fragment in TS, $\beta_{i+1} \in Act(s_i)$. So, $\rho_{i+1} = s_0 \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_i} s_i \xrightarrow{\beta_{i+1}} s_{i+1}$ is an execution fragment in TS. Then α and β_{i+1} are independent, as otherwise ρ_{i+1} would violate (A2). The induction hypothesis yields $\alpha \in Act(s_i)$. As α and β_{i+1} are independent, we get $\alpha \in Act(\beta_{i+1}(s_i)) = Act(s_{i+1})$. ■

Notation 8.16. Fully Expanded State

A state s of TS is *fully expanded* if $ample(s) = Act(s)$. ■

The following two lemmas show that the conditions (A1) through (A3) are necessary conditions to guarantee that for each execution in TS a stutter-equivalent execution in $\hat{TS}$ can be determined.

Lemma 8.17. Constructing Stutter-Equivalent Executions (Case 1)

Let ϱ be a finite execution fragment in $Reach(TS)$ of the form

$$s \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_n} s_n \xrightarrow{\alpha} t$$

where $\beta_i \notin ample(s)$, for $0 < i \leqslant n$, and $\alpha \in ample(s)$. If $ample(s)$ satisfies (A1) through (A3), then there exists an execution fragment ϱ' of the form

$$s \overset{\alpha}{\Longrightarrow} t_0 \xrightarrow{\beta_1} t_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_{n-1}} t_{n-1} \xrightarrow{\beta_n} t$$

and $\varrho \triangleq \varrho'$.

Proof: Let $\varrho = s \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_n} s_n \xrightarrow{\alpha} t$ be a finite execution fragment in $\mathit{Reach}(TS)$ where $\beta_i \notin \mathit{ample}(s)$, for $0 < i \leqslant n$, and $\alpha \in \mathit{ample}(s)$. By Lemma 8.15, α is independent of $\{ \beta_1, \ldots, \beta_n \}$. State s is not fully expanded, since $\beta_1 \in \mathit{Act}(s) \setminus \mathit{ample}(s)$. By condition (A3), α is a stutter action. The claim now follows from Lemma 8.10 (page 604). ■

Provided $\beta_1, \ldots, \beta_n \notin \mathit{ample}(s)$ and $\alpha \in \mathit{ample}(s)$, replacing the action sequence $\beta_1 \ldots \beta_n \, \alpha$ in TS by $\alpha \, \beta_1 \ldots \beta_n$ (as described in case 1) results in a stutter-equivalent execution fragment. The following lemma is similar in flavor but deals with infinitely many actions that are independent of $\alpha \in \mathit{ample}(s)$.

Lemma 8.18. Constructing Stutter-Equivalent Executions (Case 2)

Let $\rho = s \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} s_2 \xrightarrow{\beta_3} \ldots$ be an infinite execution fragment in $\mathit{Reach}(TS)$ where $\beta_i \notin \mathit{ample}(s)$, for $i > 0$. If $\mathit{ample}(s)$ satisfies (A1) through (A3), then there exists an execution fragment ρ' of the form

$$s \overset{\alpha}{\Longrightarrow} t_0 \xrightarrow{\beta_1} t_1 \xrightarrow{\beta_2} t_2 \xrightarrow{\beta_3} \ldots$$

where $\alpha \in \mathit{ample}(s)$ and $\rho \triangleq \rho'$.

Proof: Let $\rho = s \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} s_2 \xrightarrow{\beta_3} \ldots$ be an infinite execution fragment in $\mathit{Reach}(TS)$ where $\beta_i \notin \mathit{ample}(s)$, for $i > 0$. As $\beta_1 \in \mathit{Act}(s) \setminus \mathit{ample}(s)$, state s is not fully expanded. By condition (A3), all actions $\alpha \in \mathit{ample}(s)$ are stutter actions. Condition (A1) yields the existence of an action $\alpha \in \mathit{ample}(s)$. The claim now follows directly from Lemma 8.7 (page 602) and Lemma 8.11 (page 605). ■

Remark 8.19. An Alternative Dependency Condition (A2′)

Consider the following variant of (A2): for any $s \in \hat{S}$ with $\mathit{ample}(s) \neq \mathit{Act}(s)$, any $\alpha \in \mathit{ample}(s)$ is independent of $\mathit{Act}(s) \setminus \mathit{ample}(s)$. This variant, referred to as (A2′) in the sequel, seems—at first sight—a reasonable variant of (A2). It simply requires that any ample action of a nonfully expanded state s is independent of all enabled actions in s that are not in its ample set. The following example shows, however, that (A2′) does not guarantee that $TS \triangleq \hat{TS}$.

Consider the transition system depicted in Figure 8.8 (left part). Actions α and β are independent and α and δ are stutter actions. The following ample sets satisfy the conditions (A1), (A2′), (A3) and (A4):

$$\mathit{ample}(s_0) = \{ \alpha \} \quad \text{and} \quad \mathit{ample}(s_2) = \{ \beta \} \quad \text{and} \quad \mathit{ample}(s_3) = \{ \delta \}.$$

The resulting reduced transition system $\hat{TS}$ is depicted in Figure 8.8 (right part). Conditions (A1) and (A3) are obvious. Note that $s_1 \notin \hat{S}$, and thus no requirements are imposed

on $\mathit{ample}(s_1)$. Cycle condition (A4) is fulfilled, as there is only one cycle (the self-loop at s_3), and $\mathit{ample}(s_3) = \mathit{Act}(s_3)$. As state s_0 is the only nonfully expanded state in $\hat{TS}$, condition (A2′) has only to be checked for s_0. It holds as α and β are independent.

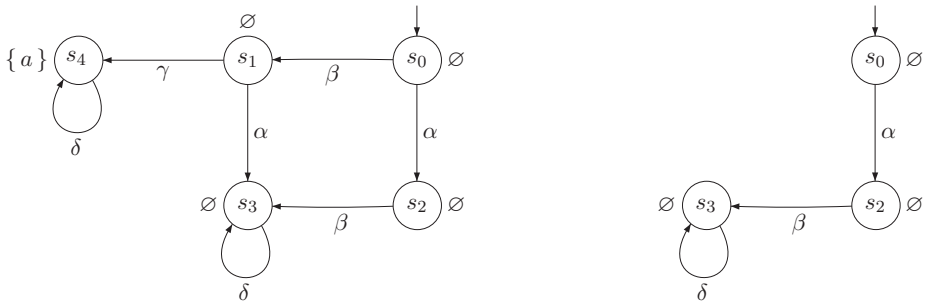


Figure 8.8: A possible, though unsound reduction that satisfies (A2′) but not (A2).

However, $TS \stackrel{\triangle}{\not=} \hat{TS}$ since, e.g., $\hat{TS} \models \Box\neg a$ (as it does not contain any a-state), while $TS \not\models \Box\neg a$. Thus, condition (A2′) does not guarantee $TS \stackrel{\triangle}{=} \hat{TS}$. The ample sets defined above violate condition (A2). This can be seen as follows. Action γ depends on α, since both actions are enabled in state s_1, but α is not enabled in $\gamma(s_1) = s_4$. Thus, the execution fragment $s_0 \xrightarrow{\beta} s_1 \xrightarrow{\gamma} s_4$ violates (A2) as α should occur before γ. So, (A2) does not allow the reduction from TS to $\hat{TS}$ as depicted in Figure 8.8. ■

Now consider the cycle condition (A4). The results we established so far ensure that the series of transformations $\rho_0 \mapsto \rho_1 \mapsto \rho_2 \mapsto \ldots$ (according to case 1 and 2) generate executions ρ_i that are stutter-equivalent to the execution ρ_0 in TS. Moreover, at least the first i transitions in ρ_i are transitions in $\hat{TS}$ and agree with the first i transitions in ρ_j for all $j \geqslant i$. The idea is now to consider the "limit" $\hat{\rho}_0$ of the executions $(\rho_i)_{i \geqslant 0}$. That is, for all $i > 0$, the ith transition in $\hat{\rho}_0$ is the ith transition in ρ_i (and ρ_j for all $j \geqslant i$). Thus, $\hat{\rho}_0$ is an execution in $\hat{TS}$. Unfortunately, without the cycle condition (A4) it cannot be guaranteed that $\rho_i \stackrel{\triangle}{=} \hat{\rho}_0$. The intuitive reason is that some (nonstutter) action β might be delayed ad infinitum in the successive construction of $\hat{\rho}_0$.

Let us consider this effect more in detail. Assume α and β are independent, enabled in all states, and α is a stutter action while β is not. The ample sets $\mathit{ample}(s) = \{\, \alpha \,\}$ for any state s fulfill conditions (A1) through (A3). Let ρ_0 be an execution with the action sequence $\beta\,\alpha^\omega$. Then the above transformation $\rho_0 \mapsto \rho_1 \mapsto \rho_2 \mapsto \ldots$ yields the executions ρ_i with the action sequence $\alpha^i\,\beta\,\alpha^\omega$ (see case 1). Their limit is the execution $\hat{\rho}_0$ with the action sequence $\alpha^\omega\,\beta \;\equiv\; \alpha^\omega$. However, it is not guaranteed that $\hat{\rho}_0 \stackrel{\triangle}{=} \rho_i$ as action β is

ignored forever in $\hat{\rho}_0$.

In a similar way, in absence of condition (A4), the transformation according to case 2 may generate an execution in $\hat{TS}$ that is not stutter-equivalent to the original execution ρ_0. Let, e.g., ρ_0 have the action sequence $\beta_1\,\beta_2\ldots$. Then case 2 might generate a series of executions $\rho_1 \mapsto \rho_2 \mapsto \ldots$ with action sequences

$$\begin{array}{cccccccc} \alpha & \beta_1 & \beta_2 & \beta_3 & \beta_4 & \beta_5 & \ldots \\ \alpha & \alpha & \beta_1 & \beta_2 & \beta_3 & \beta_4 & \ldots \\ \alpha & \alpha & \alpha & \beta_1 & \beta_2 & \beta_3 & \ldots \\ & & \vdots & & & & \end{array}$$

such that in the limit, action β_1 is never performed.

The following example illustrates the necessity of cycle condition (A4).

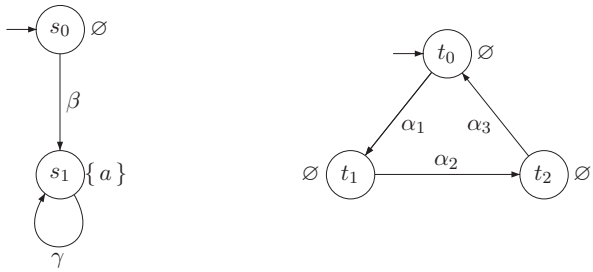


Figure 8.9: Transition systems TS_1 (left) and TS_2 (right).

Example 8.20. Necessity of Cycle Condition (A4)

Consider the transition systems TS_1 and TS_2 depicted in Figure 8.9. The transition system $TS = TS_1 \,|||\, TS_2$ is depicted in the left part of Figure 8.10. The reduced transition system $\hat{TS}$, depicted on the right of Figure 8.10, results from choosing $\mathit{ample}(\langle s_0, t_i\rangle) = \{\,\alpha_{i+1}\,\}$, for i=1, 2, 3. Conditions (A1), (A2), and (A3) are fulfilled since β is independent of $\{\,\alpha_1, \alpha_2, \alpha_3\,\}$. Consider the action sequence $\beta\,(\alpha_1\,\alpha_2\,\alpha_3)^\omega$ and the associated execution ρ in TS. The associated trace is

$$\mathit{trace}(\rho) \;=\; \varnothing\,\{\,a\,\}\,\{\,a\,\}\,\{\,a\,\}\ldots \;=\; \varnothing\,\{\,a\,\}^\omega \in \mathit{Traces}(TS).$$

In $\hat{TS}$, however, there does *not* exist an execution that is stutter-equivalent to ρ, since $\hat{TS}$ has no states labeled with a. In fact, $\mathit{Traces}(\hat{TS}) = \{\,\varnothing^\omega\,\}$. Hence, $TS \not\triangleq \hat{TS}$. (This can also be seen by considering the $\mathrm{LTL}_{\setminus\bigcirc}$ formula $\Box\neg a$. Hence $TS \not\models \Box\neg a$ and $\hat{TS} \models \Box\neg a$.)

Let us explain why the above-mentioned replacement process, which should transform the executions of TS into a stutter-equivalent execution of $\hat{TS}$, fails. In this example, case 1

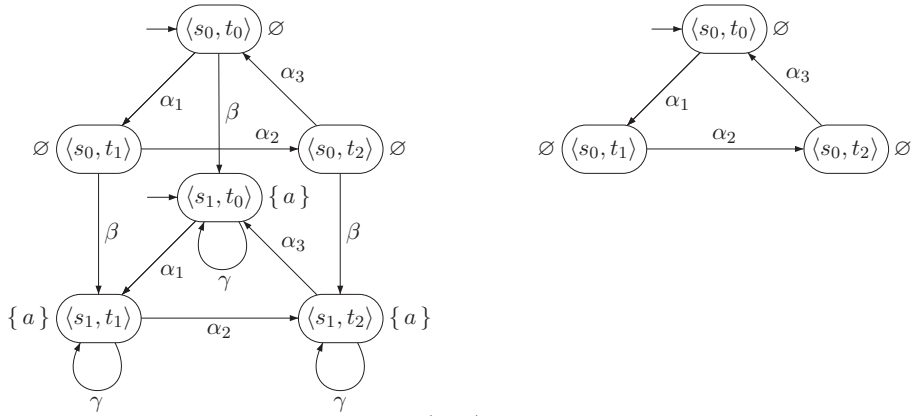


Figure 8.10: Transition system TS (left) and unsound reduced transition system $\hat{TS}$ (right).

continuously applies, and ρ_0 is successively replaced with executions that are associated to the action sequences:

$$\begin{aligned}
&\alpha_1\,\beta\,\alpha_2\,\alpha_3\,\alpha_1\,\alpha_2\,\alpha_3\ldots\\
&\alpha_1\,\alpha_2\,\beta\,\alpha_3\,\alpha_1\,\alpha_2\,\alpha_3\ldots\\
&\alpha_1\,\alpha_2\,\alpha_3\,\beta\,\alpha_1\,\alpha_2\,\alpha_3\ldots\\
&\alpha_1\,\alpha_2\,\alpha_3\,\alpha_1\,\beta\,\alpha_2\,\alpha_3\ldots\\
&\quad\vdots
\end{aligned}$$

But the action β is never performed, since β does not occur in $\hat{TS}$. Actually, the cycle con-

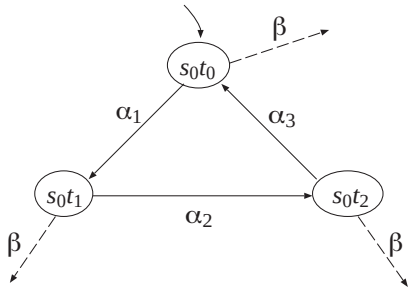


Figure 8.11: Cycle condition (A4) is violated.

dition (A4) is violated, since β is continuously enabled on the cycle $\langle s_0, t_0\rangle\,\langle s_0, t_1\rangle\,\langle s_0, t_2\rangle\,\langle s_0, t_0\rangle$, see Figure 8.11, but not in any ample sets of these states. ■

The above considerations demonstrate that conditions (A1) through (A3) cannot guarantee $TS \triangleq \hat{TS}$. The goal is now to show that (A1), (A2), and (A3) together with the cycle

condition (A4) suffice. In fact, with the cycle condition (A4) a situation described above is impossible.

Lemma 8.21. Stutter-Trace Inclusion of TS in $\hat{TS}$

If (A1) through (A4) are satisfied, then $TS \trianglelefteq \hat{TS}$.

Proof: Let ρ_0 be an execution in TS which starts in state s and is induced by the action sequence $\beta_1\,\beta_2\,\beta_3\,\ldots$ where $\beta_1 \notin \mathit{ample}(s)$. The execution ρ_0 is successively replaced with stutter-equivalent executions ρ_m, $m = 1, 2, 3, \ldots$, by means of the transformations indicated in Lemmas 8.17 and 8.18. Each of these executions ρ_m starts in state s and is based on an action sequence of the form

$$\alpha_1\,\ldots\,\alpha_m\,\beta_1\,\gamma_1\,\gamma_2\,\gamma_3\ldots$$

The action sequence $\alpha_1\,\alpha_2\ldots\alpha_m$ contains the actions of the ample sets, which are newly inserted according to Lemma 8.18, and all actions β_n, which were shifted forward according to Lemma 8.17. $\gamma_1\,\gamma_2\,\gamma_3\ldots$ denotes the remaining subsequence of $\beta_1, \beta_2, \beta_3\ldots$. Thus, ρ_m is of the form

$$s \overset{\alpha_1}{\Longrightarrow} t_1 \overset{\alpha_2}{\Longrightarrow} \ldots \overset{\alpha_m}{\Longrightarrow} t_m \xrightarrow{\beta_1} t_0^m \xrightarrow{\gamma_1} t_1^m \xrightarrow{\gamma_2} t_2^m \xrightarrow{\gamma_3} \ldots$$

where $\alpha_1, \ldots, \alpha_m$ are stutter actions.

Actually, the case $\beta_1 \notin \mathit{ample}(t_m)$ for all $m \in \mathbb{N}$ is impossible, since—due to the finiteness of TS—the path fragment $s\,t_1\,t_2\ldots t_m$ contains a cycle for sufficiently large m. Moreover, $\beta_1 \in \mathit{Act}(t_m)$ for all m by Lemma 8.15 on page 613. This contradicts condition (A4). Hence, $\beta_1 \in \mathit{ample}(t_m)$ for some $m \geqslant 1$. But then

$$s \overset{\alpha_1}{\Longrightarrow} t_1 \overset{\alpha_2}{\Longrightarrow} \ldots \overset{\alpha_m}{\Longrightarrow} t_m \overset{\beta_1}{\Longrightarrow} t_0^m$$

is an execution fragment in $\hat{TS}$ which is a prefix of ρ_{m+1} and all executions ρ_j for $j > m$.

According to the outlined recipe, we obtain an execution $\hat{\rho}_0$ in $\hat{TS}$ (as the "limit" of ρ_m, $\rho_{m+1}\ldots$), where the induced action sequence contains all actions that occur in ρ_0 (in TS). Let us assume that ρ_0 has the form $s_0 \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \ldots$ and let $0 = k_0 < k_1 < k_2 < \ldots$ such that $\beta_{k_1}\,\beta_{k_2}\,\ldots$ results from $\beta_1\,\beta_2\ldots$ by omitting all stutter actions in $\beta_1\,\beta_2, \ldots$. (Note that the subsequence $\beta_{k_1}\,\beta_{k_2}\,\ldots$ may be finite.) Then, $\mathit{trace}(\rho_0)$ has the form $A_0^+\,A_1^+\,A_2^+\ldots$, where A_i is the label $L(s_k)$ of all states s_k with $k_i \leqslant k < k_{i+1}$. Since each of the nonstutter actions β_{k_i} is eventually "processed" when generating the executions $\rho_1, \rho_2, \rho_3, \ldots$, for each index k_i there is some finite word w_i of the form $A_0^+\,A_1^+\ldots A_i^+$ and some index ℓ_i such that the traces of the executions ρ_j for all $j \geqslant \ell_i$ start with w_i. In particular, w_i is a proper prefix of w_{i+1} and the words w_i are prefixes of the trace associated with the "limit" execution $\hat{\rho}_0$. Hence, $\mathit{trace}(\hat{\rho}_0)$ has the form $A_0^+\,A_1^+\,A_2^+\ldots$, and $\rho_0 \triangleq \hat{\rho}_0$. ■

As $Traces(\hat{TS}) \subseteq Traces(TS)$, the above lemma yields that $TS \triangleq \hat{TS}$. This completes the proof of Theorem 8.13 (page 611).

Remark 8.22. Nonemptiness Condition (A1) and Terminal States

Throughout this chapter, we assume a transition system without terminal states. This assumption has been made to be consistent with Chapter 5 where LTL formulae are interpreted as languages over *infinite* words. However, all concepts presented here (as well as those in Chapter 5) can also be applied to transition systems that have terminal states. The only difference from the approach presented here is that the nonemptiness condition has to be replaced with the following requirements (A1.1) and (A1.2):

(A1.1) $ample(s) \subseteq Act(s)$.

(A1.2) $ample(s) = \varnothing$ if and only if $Act(s) = \varnothing$.

Condition (A1.1) ensures that $\hat{TS}$ is a subtransition system of TS, while (A2.1) guarantees that any terminal state in $\hat{TS}$ is a terminal state in TS. Theorem 8.13 holds for transition systems with terminal states, i.e., conditions (A1.1), (A1.2), and (A2) through (A4) yield $TS \triangleq \hat{TS}$ ■

8.2.2 Dynamic Partial Order Reduction

We now consider the integration of partial order reduction as part of LTL model checking. The basic strategy is to generate $\hat{TS}$ during model checking. This is in contrast to static partial order reduction in which $\hat{TS}$ is constructed prior to the verification. To simplify matters, we first treat on-the-fly, or dynamic, partial order reduction during invariant checking. As invariants can be checked by a depth-first search (DFS) technique, this boils down to integrating the ample set technique in a depth-first search. Subsequently, we show how nested depth-first search—the algorithm to find cycles containing an accept state in $\hat{TS} \otimes \mathcal{A}_{\neg\varphi}$—can be adapted such that partial order reduction is employed.

Partial Order Reduction in Depth-First Search Consider the invariant $\Box\Phi$ where Φ is a propositional formula (i.e., not an LTL formula). In order to check whether $TS \models \Box\Phi$, a depth-first search algorithm can be applied, as explained in Chapter 3, that checks in every state of TS whether Φ holds. This is done during the state-space generation. On exploring a state s, all outgoing transitions of s are considered, i.e., all actions in $Act(s)$ are considered. When integrating partial order reduction into this procedure, only

the α-successor states of s are considered for which $\alpha \in \mathit{ample}(s)$. The resulting depth-first search algorithm is provided in Algorithm 38 on page 622. It generates the ample sets for any fresh encountered state of $\hat{TS}$ and explores the successor states under $\Longrightarrow$ according to a depth-first search strategy. The set $\mathit{mark}(s)$ keeps track of the actions in $\mathit{ample}(s)$ that have been considered during the search. As soon as this set equals $\mathit{ample}(s)$, all ample successors of s are considered, state s is popped from the stack, and Φ is checked. In case an action α is considered whose successor has not been encountered before, the state is marked as being reachable (under $\Longrightarrow$) and pushed on the stack. For any freshly encountered state, an approximation of its ample set is generated that satisfies (A1) through (A3), but possibly not (A4). Later on, we will see that establishing these constraints can be done by means of local criteria. Roughly speaking, it is first attempted to determine $\mathit{ample}(s)$ as the set of actions of a single process that are enabled in state s. To ensure the cycle condition (A4), the ample set is enlarged on demand. This is indicated by the statement $\mathit{ample}(s') := \mathit{Act}(s')$. This technique relies on replacing (A4) by the stronger condition:

(A4′) Strong cycle condition

Any cycle in $\hat{TS}$ contains at least one state s with $\mathit{ample}(s) = \mathit{Act}(s)$.

It is not difficult to assess that (A4′) is a sufficient criterion for (A4), provided the other ample set conditions (A1) through (A3) hold. This is shown in the following lemma.

Lemma 8.23. Strong Cycle Condition

If (A1) through (A3) hold, then condition (A4′) implies (A4).

Proof: By contraposition. Assume $s_0 \ldots s_n$ (with $n > 0$) is a cycle in $\hat{TS}$ with $\mathit{Act}(s_j) = \mathit{ample}(s_j)$ for some $0 < j \leqslant n$, i.e., state s_j is fully expanded. Assume (A4) is violated, i.e., for some $i \neq j$ it holds that $\beta \in \mathit{Act}(s_i)$ and $\beta \notin \mathit{ample}(s_k)$ for all $0 < k \leqslant n$. Consider the transition $s_i \xrightarrow{\alpha_i} s_{i+1}$ that is part of the cycle $s_0 \ldots s_n$. (Here, $i{+}1$ should be read as $(i{+}1) \bmod n$.) As $s_0 \ldots s_n$ is a cycle in $\hat{TS}$, it follows $\alpha_i \in \mathit{ample}(s_i)$. Condition (A2) yields that all actions in $\mathit{ample}(s_i)$ are independent of those in $\mathit{Act}(s_i) \setminus \mathit{ample}(s_i)$. In particular, α_i is independent of $\beta \notin \mathit{ample}(s_i)$. But then, $\beta \in \mathit{Act}(\alpha_i(s_i)) = \mathit{Act}(s_{i+1})$. As $\beta \notin \mathit{ample}(s_{i+1})$, it follows by a similar reasoning that $\beta \in \mathit{Act}(s_{i+2})$. Continuing this reasoning, we obtain $\beta \in \mathit{Act}(s_j) \setminus \mathit{ample}(s_j)$. This, however, contradicts that $\mathit{ample}(s_j) = \mathit{Act}(s_j)$. ■

The advantage of (A4′) is that it can easily be integrated in the depth-first search algorithm. If the depth-first search finds a backward edge $s' \xrightarrow{\alpha} s''$, i.e., s' is the current state and $s'' = \alpha(s')$ for the current action $\alpha \in \mathit{ample}(s')$ and s'' is on the stack U, then a cycle

$s'' \ldots s' \ldots s''$ has been found that contains s'. Thus, we may simply enlarge $\mathit{ample}(s')$ by adding all actions $\beta \in \mathit{Act}(s')$ that are not in the current ample set of s. This turns s' into a fully expanded state and ensures (A4′). Alternatively, we may fully expand $\alpha(s')$, as it suffices to fully expand an arbitrary state on the cycle. (The check whether a state is on the stack U can be realized in constant time by, e.g., organizing R by a hash table where each entry is equipped with a bit indicating whether the state is in U or not.)

Example 8.24. POR (Reachability Analysis)

Consider the following concurrent program consisting of two parallel processes:

```
Process 0:                          Process 1:
        while true {                        while true {
ℓ0 :      skip;                     ℓ1 :      skip;
m0 :      wait until (¬b) {         m1 :      wait until (b) {
n0 :        ... critical section ...}   n1 :      ... critical section ...}
          b := true;                          b := false;
        }                                   }
```

The atomic proposition a holds in the states of $\mathit{TS}(\mathit{PG}_0 \,|||\, \mathit{PG}_1)$, see Figure 8.12, for which at least one of the processes is in location n_0 or n_1. Action δ_i denotes the execution of the skip statement by process i, α_i denotes the action that corresponds to waiting at program location m_i; β_i is the action denoting the exit of the busy-wait cycle by process i, and γ_i the action to return to the beginning of the loop. The initial value of the shared boolean variable b is unspecified.

We apply Algorithm 38 with $\Phi = \text{true}$—just a state-space generation—to $\mathit{TS}(\mathit{PG}_0 \,|||\, \mathit{PG}_1)$. We start with state $s_0 = \langle \ell_0, \ell_1, \neg b \rangle$. The actions δ_0 and $\delta_1 \in \mathit{Act}(s_0)$ are stutter actions and independent. The possibilities are

$$\mathit{ample}(s_0) \;=\; \{\, \delta_0 \,\}, \quad \mathit{ample}(s_0) \;=\; \{\, \delta_1 \,\} \;\text{ or }\; \mathit{ample}(s_0) \;=\; \{\, \delta_0, \delta_1 \,\}.$$

The first two alternatives are equivalent and lead to a better reduction than the last alternative. Let

$$\mathit{ample}(s_0) \;=\; \{\, \delta_0 \,\}.$$

Intuitively, this corresponds to assigning a higher priority to process zero. The next encountered state is $\delta_0(s_0) = s_1 = \langle m_0, \ell_1, \neg b \rangle$. Actions δ_1 and β_0 belong to $\mathit{Act}(s_1)$ and are independent. β_0 (and β_1) are not stutter actions. Let

$$\mathit{ample}(s_1) \;=\; \{\, \delta_1 \,\}.$$

The next encountered state $\delta_1(s_1) = s_2 = \langle m_0, m_1, \neg b \rangle$. We have $\mathit{Act}(s_2) = \{\, \alpha_1, \beta_0 \,\}$. The choice $\mathit{ample}(s_2) = \{\, \alpha_1 \,\}$ violates condition (A4) as it closes the cycle $s_1\, s_1$ on which the

Algorithm 38 Invariant checking using partial order reduction

Input: finite transition system TS and propositional formula Φ
Output: "yes" if $TS \models \Box\Phi$", otherwise "no" plus a counterexample

```
set of states R := ∅;                                   (* the set of reachable states *)
stack of states U := ε;                                 (* the empty stack *)
bool b := true;                                         (* all states in R satisfy Φ *)
while (I \ R ≠ ∅ ∧ b) do
  let s ∈ I \ R;                                        (* choose an arbitrary initial state not in R *)
  visit(s);                                             (* perform a DFS for each unvisited initial state *)
od
if b then
  return("yes")                                         (* TS ⊨ "always Φ" *)
else
  return("no", reverse(U))                              (* counterexample arises from the stack content *)
fi
```

```
procedure visit (state s)
  push(s, U);                                           (* push s on the stack *)
  R := R ∪ { s };                                       (* mark s as reachable *)
  compute ample(s) satisfying (A1)–(A3);                (* see Section 8.2.3 *)
  mark(s) := ∅;                                         (* taken actions in s *)
  repeat
    s' := top(U);
    if ample(s') = mark(s') then
      pop(U);                                           (* all ample actions have been taken *)
      b := b ∧ (s' ⊨ Φ);                                (* check validity of Φ in s' *)
    else
      let α ∈ ample(s') \ mark(s');
      mark(s') := mark(s') ∪ { α };                     (* mark α as taken *)
      if α(s') ∉ R then
        push(α(s'), U);
        R := R ∪ { α(s') };                             (* α(s') is a new reachable state *)
        compute ample(α(s')) satisfying (A1)–(A3);      (* see Section 8.2.3 *)
        mark(α(s')) := ∅;
      else
        if α(s') ∈ U then ample(s') := Act(s'); fi      (* establish (A4') *)
      fi
    fi
  until ((U = ε) ∨ ¬b)
endproc
```

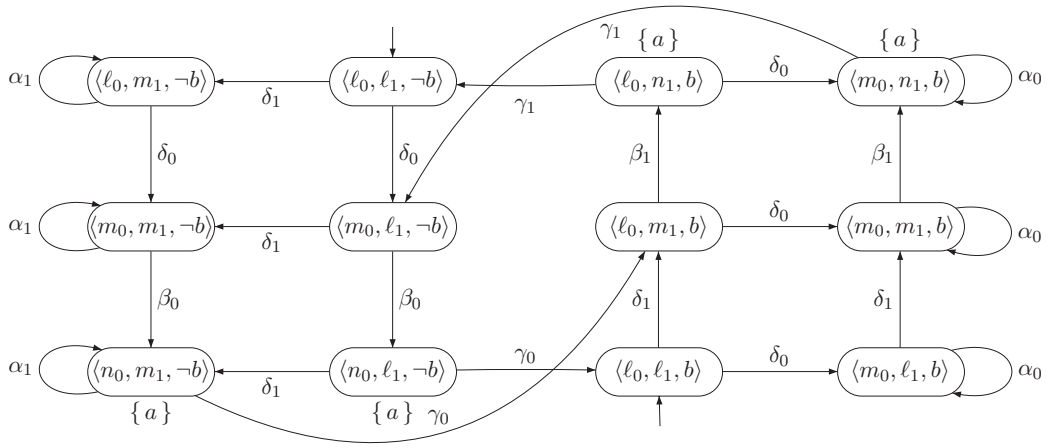


Figure 8.12: Transition system for the program $PG_1 \,|||\, PG_2$.

action β_0 is enabled but is never chosen. As β_0 is not a stutter action, the only reasonable choice is

$$ample(s_2) = \{ \alpha_1, \beta_0 \}.$$

By continuing such arguments the reduced transition system $\hat{TS}$ depicted in Figure 8.13 is obtained. Thus, eight of the reachable 12 states in TS are obtained by partial order reduction, provided the ample sets are chosen as indicated above. ■

Partial Order Reduction in Nested Depth-First Search The next issue is to integrate partial order reduction in the LTL$_{\setminus\bigcirc}$ model-checking procedure. Rather than checking $TS \models \varphi$ it is verified whether $\hat{TS} \models \varphi$. The rough idea for verifying $\hat{TS} \models \varphi$ is as follows. As usual, a Büchi automaton $\mathcal{A}_{\neg\varphi}$ is constructed for the LTL$_{\setminus\bigcirc}$ formula φ. The reachable states of the product transition system $\hat{TS} \otimes \mathcal{A}_{\neg\varphi}$ are generated and during this state-space generation phase, the persistence property "eventually forever no accept state" is checked by means of the nested depth-first search described in Section 5.2. In the outer depth-first search, a reachable accept state is searched for, whereas in the inner depth-first search, it is checked whether a reachable accept state lies on a cycle. In order to obtain correct results it is evident that in both depth-first searches, the same ample sets need to be used. Otherwise, different transition systems are considered.

As described in Section 5.2, a cycle with an accept state in $\mathcal{A}_{\neg\varphi}$ is found if the cycle check—the inner DFS which attempts to find a backward edge to an accept state—is started for state s once the outer DFS for s is completed, i.e., when all successors of s have been explored. When adopting the same approach as in Algorithm 38 for the nested

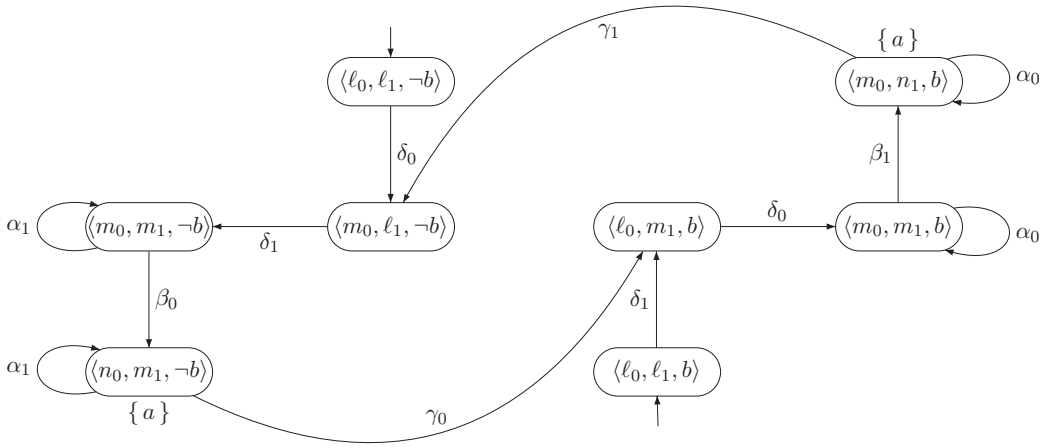


Figure 8.13: The reduced transition system $\hat{TS}$ for $PG_1 \,|||\, PG_2$.

depth-first search, the inner and outer depth-first search may dynamically change the ample set. This is due to the extension of the ample set to accomplish (A4′). Hence, we have to ensure that the inner DFS is started only if *ample*(s) cannot change anymore. To that end, a slightly modified version of the nested DFS procedure (indicated at the end of Section 5.2) is exploited. In this variant, the nested DFS is aborted as soon as the inner DFS visits a state t which is on the stack U for the outer DFS.

The main steps are outlined in Algorithm 39. The procedure cycle_check_por is a slight variant of the cycle check as given in Algorithm 7 (see page 210). This algorithm aborts as soon as a state t is visited which is on the stack U for the outer DFS. That is, this variant seeks an arbitrary backward edge rather than a backward edge from s' to s (as is done in Algorithm 40 on page 626). The test whether the current state t of the inner DFS is on the stack U for the outer DFS can be performed in (expected) constant time if the elements in R (organized as a hash table) are augmented with a bit that yields the information whether or not a state is in U.

To ensure that the inner DFS uses the same ample sets *ample*(s) as the outer DFS (and to avoid the recomputation of *ample*(s) in the inner DFS) one might use one bit for any enabled action α which indicates whether α belongs to *ample*(s). A simple solution is to use hashing for representing T and R, bits for U as well as the action bits. Thus, the entries in this hash table are tuples $\langle s, b_T, b_U, \langle \alpha_1, b_1 \rangle, \ldots, \langle \alpha_n, b_n \rangle \rangle$ consisting of a state $s \in R$, bit b_T indicating whether $s \in T$, bit b_U indicating whether $s \in U$, and a list $\langle \alpha_1, b_1 \rangle, \ldots, \langle \alpha_n, b_n \rangle$ for all actions $\alpha_i \in Act(s)$ where bit b_i equals one iff $\alpha_i \in$ *ample*(s).

Algorithm 39 Nested depth-first search with partial order reduction

Input: finite transition system TS and propositional formula Φ
Output: "yes" if $TS \models \Box\Phi$", otherwise "no" plus a counterexample

```
set of states R := ∅;                                  (* set of visited states in the outer DFS *)
stack of states U := ε;                                (* stack for the outer DFS *)
set of states T := ∅;                                  (* set of visited states in the inner DFS *)
stack of states V := ε;                                (* stack for the inner DFS *)
boolean cycle_found := false;

while (I \ R ≠ ∅ ∧ ¬cycle_found) do
  let s ∈ I \ R;
  reachable_cycle(s);
od
if ¬cycle_found then
  return ("yes")                                       (* TS ⊨ "eventually for ever Φ" *)
else
  return ("no", reverse(V.U))                          (* stack contents yield a counterexample *)
fi
```

```
procedure reachable_cycle (state s)
  push(s, U);                                          (* push s on the stack *)
  R := R ∪ { s };                                      (* mark s as reachable *)
  compute ample(s) satisfying (A1)–(A3);               (* see Section 8.2.3 *)
  mark(s) := ∅;                                        (* taken actions in s *)
  repeat
    s' := top(U);
    if ample(s') = mark(s') then
      if s ⊭ Φ then
        cycle_found := cycle_check_por(s');            (* start the inner DFS in s' *)
      fi
      pop(U);
    else
      let α ∈ mark(s') \ ample(s');
      mark(s') := mark(s') ∪ { α };                    (* mark α as taken *)
      if α(s') ∉ R then
        push(α(s'), U);
        R := R ∪ { α(s') };                            (* α(s') is a new reachable state *)
        compute ample(α(s')) satisfying (A1)–(A3);     (* see Section 8.2.3 *)
        mark(α(s')) := ∅;
      else
        if α(s') ∈ U then ample(s') := Act(s'); fi     (* establish (A4') *)
      fi
    fi
  until ((U = ε) ∨ ¬b)
endproc
```

Algorithm 40 Cycle detection (inner DFS) using ample sets

Input: state s in $\hat{TS}$ with $s \not\models \Phi$
Output: true if s belongs to a cycle in $\hat{TS}$; otherwise false

```
(* T organizes the set of states that have been visited in previous calls of      *)
(* cycle_check_por(·). V serves as DFS stack for cycle_check_por(·),             *)
(* U as DFS stack for the outer DFS (Algorithm 39).                              *)

procedure boolean cycle_check_por(state s)
  boolean cycle_found := false;                          (* no cycle found yet *)
  push(s, V);
  T := T ∪ { s };
  repeat
    t := top(V);          (* check whether t is still on the stack of the outer DFS *)
    if t ∈ U then
      cycle_found := true;                          (* there is a cycle t ... s ... t *)
      push(t, V);
    else
      if ample(t) = mark(t) then
        pop(V);                  (* all successors of t in TS^ have been explored *)
      else
        let α ∈ ample(t) \ mark(t);
        mark(t) := mark(t) ∪ { α };
        push(α(t), V);
        T := T ∪ { α(t) };
      fi
    fi
  until ((V = ε) ∨ cycle_found)
  return cycle_found
endproc
```

8.2.3 Computing Ample Sets

This section is devoted to techniques to determine the ample sets by means of a static analysis of channel systems. The aim is to find criteria for selecting ample sets that can be checked efficiently by a *syntactic* analysis of the high-level formal description of the system provided as a channel system. As the cycle condition (A4) can be established in the way described before, the focus is on satisfying conditions (A1) through (A3). Recall that a channel system CS consists of a number of concurrent processes, $\mathcal{P}_1$ through $\mathcal{P}_n$ say, given as program graphs PG_1 through PG_n that may have shared variables and may communicate with each other via channels, i.e., first-in-first-out buffers that can store messages. Communication via a channel of capacity zero corresponds to handshaking plus the exchange of some data. The transition system of CS is denoted as TS, i.e., $TS = TS(CS)$ where $CS = [PG_1 \mid \ldots \mid PG_n]$. A detailed introduction and formalization of channel systems is given in Chapter 2. When considering channel c as buffer, the communication action $c!v$ puts value v (at the rear of) the buffer whereas $c?x$ retrieves an element from (the front of) the buffer while assigning it to x.

Let Act_i and Loc_i denote the action set and set of locations of PG_i, respectively. Assume that any action $\alpha \in Act_i$ occurs in exactly one edge $\ell \overset{g:\alpha}{\hookrightarrow} \ell'$ in program graph PG_i and that the action sets $Act_1, \ldots, Act_n$ are pairwise disjoint. (This can always be ensured by renaming of actions.) In order to treat all edges in a uniform manner, we assume communication actions to be preceded by guards that equal true, e.g., $\ell \overset{c?x}{\hookrightarrow} \ell'$ is written as $\ell \overset{g:c?x}{\hookrightarrow} \ell'$ where g equals true.

Let us first introduce some notations. For action α, let $Var(\alpha)$ denote the set of variables occurring in α, and $Modify(\alpha) \subseteq Var(\alpha)$ the set of variables that are modified by α. For example:

- $Var(x := x + y) = \{ x, y \}$ and $Modify(x := x + y) = \{ x \}$,
- $Var(c?x) = Modify(c?x) = \{ x \}$, and
- $Var(c!v) = Modify(c!v) = \varnothing$ if v is a value (in the domain of channel c).

Variable x is *local* to process $\mathcal{P}_i$ if no other process refers to x, i.e., if $x \notin Var(\alpha)$ for any $\alpha \in \bigcup_{\substack{1 \leqslant j \leqslant n \\ j \neq i}} Act_j$.

Let $Act_i(s) = Act(s) \cap Act_i$ denote the set of actions of process $\mathcal{P}_i$ which are enabled in (global) state s in $TS = TS(CS)$. State s has the form $\langle \ell_1, \ldots, \ell_n, \eta, \xi \rangle$ where ℓ_i denotes

the current location (control point) of PG_i, η is the variable evaluation, and ξ the channel evaluation. As before, TS is assumed to have no terminal states, i.e., $Act_i(s) \neq \varnothing$ for some process $\mathcal{P}_i$, for any state s.[1]

As a first step, we partition the set of processes $\mathcal{P}_1$ through $\mathcal{P}_n$ into two blocks. The first block contains processes $\mathcal{P}_{i_1}$ through $\mathcal{P}_{i_k}$; the remaining processes constitute the other block. A possible criterion to obtain such partitioning is to exploit the communication pattern of the processes, e.g., no process $\mathcal{P}_{i_j}$ $(0 < j \leqslant k)$ is able to communicate to processes outside this set. The intuition is to let $\mathit{ample}(s) = Act_{i_1}(s) \cup \ldots \cup Act_{i_k}(s)$, for (global) state s in $TS(CS)$. This guarantees that by executing actions not in $\mathit{ample}(s)$, it is impossible that an action β that depends on $\mathit{ample}(s)$ will become enabled in a global state, and therefore would possibly be executed before an action in $\mathit{ample}(s)$—violating (A2). For simplicity, suppose k=1 and let $\mathit{ample}(s) = Act_i(s)$, for some i. If no other process $\mathcal{P}_j$ with $j \neq i$ can perform an action in s, then $Act_i(s) = Act(s)$, and s is fully expanded (and evidently satisfying (A1) through (A3)).

Consider that $Act_i(s)$ is a proper subset of $Act(s)$. Checking the nonemptiness condition (A1) is trivial; it just amounts checking whether process $\mathcal{P}_i$ can perform an action in state s, i.e., $Act_i(s) \neq \varnothing$. Checking the stutter condition (A3) amounts to check whether all actions in $Act_i(s)$ are stutter actions. This step can be facilitated by determining the stutter actions by a static analysis—α is a stutter action if the atomic propositions in $TS(CS)$ refer neither to a variable in $\mathit{Modify}(\alpha)$ nor to the source or target location of edges of the form $\ell \overset{g:\alpha}{\hookrightarrow} \ell'$ nor to the content of channel c if α is a receive or send action on c.

Conditions (A1) and (A3) are thus relatively easy to deal with. This does not hold for condition (A2) as this imposes a requirement on *any* execution in the full transition system $TS(CS)$. (The cycle condition (A4) is also a global property, but refers to the reduced transition system $\hat{TS}$ that needs to be analyzed anyway. For this reason the cycle condition is easier to handle.) It turns out that checking (A2) is as hard as checking a reachability property in the full transition system TS.

Theorem 8.25. Algorithmic Difficulty of Checking (A2)

The worst-case time complexity of checking (A2) in finite, action-deterministic transition system TS equals that of checking $TS' \models \exists\Diamond\, a$, for some $a \in AP$, where TS' is a finite, action-deterministic transition system of the same size as TS.

Proof: Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a finite, action-deterministic transition system

[1] In case $TS = TS(CS)$ has terminal states, condition (A1) has to be replaced by (A1.1) and (A1.2), see Remark 8.22 on page 619.

and $a \in AP$. Without loss of generality, assume $I = \{ s_0 \}$. (In case there is more than one initial state, a new state s_0 can be introduced such that s_0 has no incoming transitions, and $s_0 \xrightarrow{\alpha} s'$, if $s \xrightarrow{\alpha} s'$ for initial state s.) Assume there is some (possibly unreachable) state $t \in S$ with $t \models a$. (If such a state does not occur in TS, then this state t may just be added.)

The idea of the proof is to define transition system TS' and ample sets for the states in TS' such that $t \in \mathit{Reach}(TS)$ if and only if the ample sets defined for TS' violate condition (A2). TS' results from TS by adding the dependent actions α and β to the actions of TS, such that β is enabled in any state of TS and independent of Act while α is only enabled in the a-states (and results in the new state *trap*). Formally:

$$TS' \;=\; (S \cup \{ \mathit{trap} \}, \mathit{Act}', \rightarrow', \{ s_0 \}, AP, L')$$

where $\mathit{trap} \notin S$, $\mathit{Act}' = \mathit{Act} \cup \{ \alpha, \beta, \tau \}$ for $\alpha, \beta, \tau \notin \mathit{Act}$. The labeling function L' is irrelevant. The transition relation $\rightarrow'$ is defined by

$$\frac{s \xrightarrow{\gamma} s' \quad \gamma \in \mathit{Act}}{s \xrightarrow{\gamma}' s'} \quad \text{and} \quad \frac{s \models a}{s \xrightarrow{\alpha}' \mathit{trap}} \quad \text{and} \quad \frac{s \in S}{s \xrightarrow{\beta}' s} \quad \text{and} \quad \frac{}{\mathit{trap} \xrightarrow{\tau}' \mathit{trap}}$$

Accordingly, the transitions in TS are extended such that $\alpha \in \mathit{Act}(s)$ if $s \models a$, $\beta \in \mathit{Act}(s)$ for all $s \in S$, and *trap* is equipped with a self-loop labeled with the special action τ.

Let the ample sets for TS' be defined by

$$\mathit{ample}(s) = \mathit{Act}'(s) \text{ for all } s \in (S \cup \{ \mathit{trap} \}) \setminus \{ s_0 \} \text{ and } \mathit{ample}(s_0) = \{ \beta \}.$$

That is, all states are fully expanded except for the initial state s_0. Actions α and β are dependent in TS', since α and β are both enabled in any a-state s, but β cannot be executed in state $\alpha(s) = \mathit{trap}$. Due to the β-loops inserted at all states $s \in S$, β is independent of Act.

Claim: $TS \models \exists \Diamond a$ if and only if the ample sets in TS' violate (A2).

1. ($\Rightarrow$) Let $t \in \mathit{Reach}(TS)$ for some $t \in S$ with $t \models a$. Then there exists an initial execution fragment in TS' of the form

$$\varrho \;=\; \underbrace{s_0 \longrightarrow \;\ldots\; \longrightarrow t}_{\substack{\text{initial execution} \\ \text{fragment in } TS}} \xrightarrow{\alpha} \mathit{trap}.$$

   Since $\beta \in \mathit{Act}(s_0)$ and α and β are dependent in TS', action α depends on $\mathit{ample}(s_0) = \{ \beta \}$ in TS'. But since α is only enabled in state t and not in any state visited in ϱ prior to t, ϱ violates (A2). (Note that β cannot occur in the execution fragment $s_0 \rightarrow \ldots \rightarrow t$, as β is not an action of TS.)

2. ($\Leftarrow$) Assume (A2) is violated in TS'. Then there exists a state $v \in \mathit{Reach}(TS')$ and an execution fragment

$$v \xrightarrow{\gamma_1} s_1 \xrightarrow{\gamma_2} \ldots \xrightarrow{\gamma_n} s_n \xrightarrow{\gamma} s'$$

in TS' such that (i) γ depends on $\mathit{ample}(v)$ and (ii) $\gamma_1, \ldots, \gamma_n \notin \mathit{ample}(v)$. Since s_0 is the only state that is not fully expanded, it follows from (ii) that $v = s_0$. Due to $\mathit{ample}(s_0) = \{\, \beta \,\}$ and since β is independent of all actions in TS, (i) yields $\gamma = \alpha$. Since α is only enabled in the a-states, it follows $s_n \models a$. Thus, $TS' \models \exists\Diamond\, a$. By construction of TS', we get $v, s_1, \ldots, s_n \in \mathit{Reach}(TS)$—the only state in TS' that is not in TS is the state *trap* that has no outgoing transitions. Hence, $TS \models \exists\Diamond\, a$.

■

Remark 8.26. A Technical Remark on the Proof of Theorem 8.25

The above proof suggests that TS' needs to be explicitly generated. This is, however, not true. Suppose $TS = TS(PG_1 \,|||\, \ldots \,|||\, PG_n)$ for program graphs $PG_1, \ldots, PG_n$ and a is the atomic proposition stating that variable y evaluates to 0. Introduce a new Boolean variable x with initial value 0, strengthen all guards in any edge in $PG_1, \ldots, PG_n$ with the conjunct $x{=}0$, and introduce the program graphs PG_α, PG_β and PG_τ that are defined by

- the command "**if** $x = 0$ **then** $\underbrace{\text{skip}}_{\beta}$ **fi**" in an infinite loop (PG_β),
- the command "**if** $\underbrace{y = 0}_{a}$ **then** $\underbrace{x := 1}_{\alpha}$ **fi**" (PG_α),
- the skip-command in an infinite loop (PG_τ).

Program graphs PG_β and PG_τ consist of a single location with a self-loop (labeled with β and τ, respectively), while PG_α consists of two locations connected by an edge with the guard $y = 0$ and the action α for modeling the assignment $x := 1$. We then consider

$$TS' = TS(\underbrace{PG_1 \,|||\, \ldots \,|||\, PG_n}_{\text{original program}} \,|||\, \underbrace{PG_\alpha \,|||\, PG_\beta \,|||\, PG_\tau}_{\text{extension}}).$$

In this way, a representation of TS' is obtained from the representation of TS in constant time. ■

As a result of Theorem 8.25, (A2) is not checked for an arbitrary set of transitions, but the structure of the program graph will be exploited to determine ample sets for which

condition (A2) is guaranteed to hold. In fact, by means of a static analysis of the program graphs it will be determined which actions will be dependent. Let us consider determining these dependencies in more detail.

Overapproximating Dependencies Let $D \subseteq Act \times Act$ be a binary relation on actions, that is intended to be an overapproximation of the *dependency* relation. That is, if $(\alpha, \beta) \in D$, α and β are considered to be dependent. As D is an overapproximation, it follows that if $(\alpha, \beta) \notin D$, then α and β are independent; on the other hand, $(\alpha, \beta) \in D$ does not exclude that α and β might be independent. It is evident that if (A2) holds for D, then it also holds for the proper dependency relation. Why do we not consider the proper dependency relation, i.e., the complement of the independence relation? The problem is that the notion of independency is a *global* property. For instance, it might seem reasonable to consider the actions $x := z + y$ and $x := z$ to be dependent as they modify the same variable. This is a local criterion. However, if there is no state in TS with $y \neq 0$ in which both actions are enabled, then they could be considered to be independent. The latter is, however, a global property. Since the aim is to avoid the construction of TS, and to determine ample sets prior to the state-space generation, we have to rely on criteria that can be considered during a *static analysis* of the program graphs.

Let us briefly sketch which simple syntactic criteria can be employed to determine D. It seems reasonable that all pairs of actions belonging to the same process are dependent. A simple, though rather conservative strategy would be to consider all pairs of actions that refer to the same variable to be dependent. This is somewhat restrictive as, e.g., the assignments $x := y + 1$ and $z := y + z$ are in fact independent since they do not modify the shared variable y. A more refined strategy would consider the modified variables. A simple, conservative strategy could be to consider all communication actions acting on the same channel to be dependent (i.e., in D). This is somewhat restrictive as, e.g., actions $c!v$ and $c?x$ for channel c with capacity one can never be enabled in the same global state: either the channel is full or empty. Any action involving a synchronous channel (a channel with capacity zero) is a joint action of two processes, and may thus be considered to be dependent on all actions in both involved processes.

Local Criteria for (A2) Local criteria which ensure that the choice $ample(s) = Act_i(s)$ satisfies condition (A2) can be obtained as follows. (Recall that $Act_i(s) = Act(s) \cap Act_i$.) As before, we assume all actions for the processes $\mathcal{P}_1, \ldots, \mathcal{P}_n$ to be pairwise disjoint. To ensure condition (A2), we check whether the following two (somewhat conservative) conditions hold in global state s:

(A2.1) Any $\beta \in Act_j$ is independent of $Act_i(s)$ for $i \neq j$.

(A2.2) Any $\beta \in Act_i \setminus Act(s)$ may not become enabled through the activities of some process $\mathcal{P}_j$ with $i \neq j$.

Condition (A2.1) can be established by inspecting the program graphs of all processes $\mathcal{P}_j$ and checking whether for any $\alpha \in Act_i$ and $\beta \in Act_j$, $(\alpha, \beta) \notin D$. Note that all actions local to process $\mathcal{P}_i$ are considered to be dependent by (A2.1). Condition (A2.2) considers a global state $s = \langle \ell_1, \ldots, \ell_i, \ldots, \ell_n, \eta, \xi \rangle$ and action β of process $\mathcal{P}_i$ that might be enabled in ℓ_i, i.e., $\ell_i \overset{g:\beta}{\hookrightarrow} \ell_i'$ occurs in PG_i, but is disabled in s. This may be due to the fact that the guard g does not hold, or the action β might be blocked (e.g., a send action to a full channel). In order to ensure that β will not become enabled by activities of some other process $\mathcal{P}_j$, it should be the case that neither:

- the guard g is violated in s (as g may refer to shared variables that can be modified by other processes), nor that
- β is a disabled communication action for a channel of nonzero capacity (which may become enabled as another process either enters or removes a message from the channel), nor that
- β is a disabled handshaking action (which may become enabled due to an activity of another process),

nor that for some $i \neq j$, there is an edge $\ell_j' \overset{h:\gamma}{\hookrightarrow} \ell_j''$ in PG_j such that

- ℓ_j' is reachable (via $\hookrightarrow^*$) from the current location ℓ_j (in state s) in PG_j, and action γ modifies some variable that occurs in the guard g, or γ and β are complementary communication actions, i.e., send and receive actions on the same channel.

Thus, (A2.2) ensures that for global state $s = \langle \ell_1, \ldots, \ell_i, \ldots, \ell_n, \eta, \xi \rangle$ there do not exist actions $\beta \in Act_i \setminus Act(s)$ and $\gamma \in \bigcup_{j \neq i} Act_j$ and states

$$s' = \langle \ldots, \ell_j', \ldots, \ell_i, \ldots, \eta', \xi' \rangle, \text{ and } s'' = \langle \ldots, \ell_j'', \ldots, \ell_i, \ldots, \eta'', \xi'' \rangle$$

where process $\mathcal{P}_i$ resides in location ℓ_i (as in state s) such that:

$$\beta \notin Act(s') \text{ and } \underbrace{s \longrightarrow \ldots \longrightarrow s'}_{\beta \text{ is not enabled}} \xrightarrow{\gamma} s'' \xrightarrow{\beta} \ldots$$

Let us consider a small example. Condition (A2.2) is violated in state $s = \langle \ldots, \ell_i, \ldots, x = 0 \rangle$ if there are edges

$$\ell_i \xleftarrow{x>0:\beta} \ell_i' \quad \text{and} \quad \ell_j' \xleftarrow{x:=1} \ell_j''$$

in PG_i and PG_j with $i \neq j$ such that ℓ'_j is reachable (via $\hookrightarrow^*$) from ℓ_j. Note that the action pairs (β, γ) referred to in (A2.2), as well as the reachability relation $\hookrightarrow^*$ in the program graphs $PG_1, \ldots, PG_n$, can be determined by a static analysis. The same applies to (A2.1). As the size of program graphs is relatively small compared to their underlying transition system, the cost of analyzing the program graphs for checking (A2.1) and (A2.2) is negligible compared to the analysis of their transition systems.

The following lemma shows that conditions (A2.1) and (A2.2) are sufficient (though not necessary) for (A2):

Lemma 8.27. Sufficient Local Criteria for (A2)

If (A2.1) and (A2.2) hold, then $ample(s) = Act_i(s)$ *satisfies (A2) for all execution fragments in TS that start in state* s*.*

Proof: By contraposition. Suppose that (A2.1) and (A2.2) hold for $ample(s)$. Assume (A2) does not hold and global state s is of the form $\langle \ldots, \ell_i, \ldots \rangle$, i.e., process $\mathcal{P}_i$ is at location ℓ_i. Then, there exists a finite execution fragment

$$s \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_n} s_n \xrightarrow{\beta_{n+1}} \ldots$$

where $\beta_1, \ldots, \beta_n \notin Act_i(s)$ and β_{n+1} depends on $ample(s) = Act_i(s)$. Since (A2.1) holds, actions depending on $Act_i(s)$ are actions of process i:

$$\beta_{n+1} \in Act_i \setminus Act_i(s).$$

Let m be the largest index in $\{1, \ldots, n\}$ such that $\beta_1, \ldots, \beta_{m-1}$ are actions of other processes, i.e.:

$$\beta_1, \ldots, \beta_{m-1} \in \bigcup_{j \neq i} Act_j \setminus Act_i \quad \text{and} \quad \beta_m \in Act_i$$

Since the actions $\beta_1, \ldots, \beta_{m-1}$ cannot affect the location of process $\mathcal{P}_i$, location ℓ_i does not change in the first $m-1$ steps, i.e., states $s_1, \ldots, s_{m-1}$ are also of the form $\langle \ldots, \ell_i, \ldots \rangle$. As $\beta_m \notin Act_i(s)$ and $\beta_m \in Act_i(s_{m-1})$, action β_m becomes enabled in location ℓ_i by executing one of the actions in the set $\{\beta_1, \ldots, \beta_{m-1}\}$. Since $\beta_1, \ldots, \beta_{m-1}$ are actions of processes different from $\mathcal{P}_i$, this contradicts (A2.2). ■

Algorithm 41 on page 634 summarizes the main steps for computing the ample set of a given state s such that conditions (A1), (A2), and (A3) are fulfilled. We consider here the case where the candidates are the action sets $Act_i(s)$, i.e., the actions of $\mathcal{P}_i$ that are enabled in s.

Algorithm 41 Computation of *ample*(s)

Input: state $s = \langle \ell_1, \ldots, \ell_n, \eta, \xi \rangle$ in $\hat{TS}$
Output: *ample*$(s) \subseteq Act(s)$ such that (A1)-(A3) are fulfilled

(* let $D \subseteq Act \times Act$ such that $(\alpha, \beta) \in D$ if α and β are dependent *)

determine $Act_i(s)$ for all $0 < i \leqslant n$;

if $(\exists i.\, Act_i(s) = Act(s))$ **then return** $Act(s)$ **fi**;
for $i = 1$ **to** n **do** (* check whether *ample*$(s) = Act_i(s)$ is possible *)
if ($Act_i \neq \varnothing$ and $Act_i(s)$ only contains stutter actions) **then**
  if $(\exists j \neq i.\, Act_i(s) \times Act_j(s) \cap D = \varnothing)$ **then**
    $b :=$ true; (* (A2.1) holds *)
    if $\exists \ell_i \overset{g:\beta}{\hookrightarrow} \ell_i'$ in PG_i where β is a handshaking action **then**
      $b :=$ false; (* (A2.2) violated *)
    else
      for all $\ell_i \overset{g:\beta}{\hookrightarrow} \ell_i'$ in PG_i and $\ell_j' \overset{h:\gamma}{\hookrightarrow} \ell_j''$ in PG_j with $j \neq i$ and $\ell_j \hookrightarrow^* \ell_j'$ **do**
        if ($\eta \not\models g$ and γ modifies some variable that occurs in g) or
            (β and γ are complementary communication actions) **then**
          $b :=$ false; (* (A2.2) violated *)
        fi
      od
    fi
    if (b) **then return** $Act_i(s)$ **fi** (* (A1)-(A3) hold *)
  fi
fi
od
(* *ample*(s) cannot be defined as the action set of one process *)
return $Act(s)$ (* *ample*$(s) := Act(s)$ *)

8.2.4 Static Partial Order Reduction

Rather than realizing the partial order reduction criteria during the model-checking process, in static partial order reduction the reduced transition system $\hat{TS}$, or preferably, a high-level formal description of $\hat{TS}$, is constructed prior to the verification. The advantage of this approach is that it may be combined with other state-space reduction techniques such as symbolic approaches using binary decision diagrams, or bisimulation minimization. This section treats static partial order reduction starting from the high-level description $PG_1 \,|||\, \ldots \,|||\, PG_n$, where, for the sake of simplicity, it is assumed that the program graphs have shared variables but do not communicate via channels.

The main quest for static partial order reduction is to determine ample sets that satisfy the conditions (A1) through (A4). As described in Section 8.2.3, the conditions (A1), (A2), and (A3) can be established by means of local criteria of transition systems. It turns out that these criteria can also be easily reformulated for program graphs and thus can be checked prior to model checking. In contrast to on-the-fly partial order reduction, however, (A4) cannot be checked by establishing the strong cycle condition (A4′), as this condition is tailored to the state space generation by means of a depth-first search, a step that is absent in the static approach. Instead, (A4) is established by inspection of the program graphs PG_1 through PG_n by fixing a set A_{sticky} of *sticky* actions:

$$A_{\text{sticky}} \subseteq Act$$

The purpose of actions in A_{sticky} is to serve to "break" any cycle in $\hat{TS}$. For state s, action $\alpha \in A_{\text{sticky}} \cap \mathit{ample}(s)$ "sticks", so to speak, all other enabled actions in s and forces their exploration.

Notation 8.28. Visible Action

An action $\alpha \in Act$ is *visible* if α is not a stutter action, i.e., if there exists a state s in TS with $\alpha \in Act(s)$ and $L(s) \neq L(\alpha(s))$. Let Vis denote the set of visible actions. ■

The following requirements are imposed on A_{sticky}.

(S1) Visibility condition

$Vis \subseteq A_{\text{sticky}}$

The visibility condition (S1) asserts that all visible actions are sticky, or equivalently, that all actions in $Act \setminus A_{\text{sticky}}$ are stutter actions.

(S2) Cycle-breaking condition

For any cycle $s_0 \stackrel{\beta_1}{\Longrightarrow} \ldots \stackrel{\beta_n}{\Longrightarrow} s_n$ in $\hat{TS}$, $A_{\text{sticky}} \cap \{ \beta_1, \ldots, \beta_n \} \neq \varnothing$.

Condition (S2) thus asserts that any cycle in $\hat{TS}$ contains at least one sticky action.

Consider now the following new condition on ample sets:

(A3/4) Sticky condition

If $\mathit{ample}(s) \neq \mathit{Act}(s)$, then $\mathit{ample}(s) \cap A_{\text{sticky}} = \varnothing$.

Condition (A3/4) asserts that ample actions of nonfully expanded states are not sticky. The condition (A3/4) implies both the stutter condition (A3) and the strong cycle condition (A4′). This is stated by the following lemma.

Lemma 8.29. Sticky vs. Stutter and Cycle Condition

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a finite, action-deterministic transition system and $A_{sticky} \subseteq Act$ satisfying (S1) and (S2). If the ample sets for $s \in S$ satisfy (A3/4), then (A3) and (A4′) hold.

Proof: Let $s \in S$ with $\mathit{ample}(s) \neq \mathit{Act}(s)$ and suppose (A3/4) holds.

1. The sticky condition (A3/4) ensures that $\mathit{ample}(s) \cap A_{\text{sticky}} = \varnothing$. By the visibility condition (S1), any $\alpha \in \mathit{ample}(s)$ is a stutter action. Thus, (A3) holds.

2. Consider the cycle $s_0 \stackrel{\beta_1}{\Longrightarrow} s_1 \ldots \stackrel{\beta_n}{\Longrightarrow} s_n$ in $\hat{TS}$. By the cycle-breaking condition (S2), we have $\beta_i \in A_{\text{sticky}}$ for some $0 < i \leqslant n$. So $\beta_i \in \mathit{ample}(s_{i-1}) \cap A_{\text{sticky}}$. The sticky condition (A3/4) yields that s_{i-1} is fully expanded, i.e., $\mathit{ample}(s_{i-1}) = \mathit{Act}(s_{i-1})$. Hence, (A4′) holds.

■

As a consequence we have the following result. If A_{sticky} satisfies (S1) and (S2), and the ample sets satisfy (A1), (A2), and (A3/4), then $TS \triangleq \hat{TS}$. In the sequel of this section, we will discuss in detail how to check that all conditions (S1), (S2), (A1), (A2) and (A3/4) are fulfilled by inspecting the program graphs of the concurrent processes involved.

Program Graph Reduction. First, we discuss how to establish the ample set conditions by a static analysis of the program graphs. It is assumed that A_{sticky} satisfies (S1) and (S2); later on it is explained how this can be guaranteed. For the sake of simplicity, processes cannot communicate via channels (but may have shared variables). That is, the full transition system TS has the form $TS = TS(PG)$ where $PG = PG_1 \,|||\, \ldots \,|||\, PG_n$ for program graphs $PG_1, \ldots, PG_n$ that do not contain any communication actions on channels. As before, Act_i denotes the action set of PG_i and $Act_i \cap Act_j = \varnothing$ if $i \neq j$, and that no $\alpha \in Act_i$ occurs on more than one edge in PG_i. Thus, there is a one-to-one correspondence between the actions in PG_i and the edges in PG_i. It is assumed that an overapproximation D of the dependency relation has been computed (see page 631). Finally, we assume that PG_i has a single starting location, denoted $\ell_{0,i}$, and an initial condition $g_{0,i}$.

The idea is to transform $PG = PG_1 \,|||\, \ldots \,|||\, PG_n$ into $\widehat{PG} = \widehat{PG_1} \,|||\, \ldots \,|||\, \widehat{PG_n}$ such that $TS(PG) \triangleq TS(\widehat{PG})$. The transformation of each program graph PG_i proceeds in a number of steps. First, the edges in PG_i are marked with the labels *good* and *sticky* and locations with the label *ample* in the following way:

1. For any action $\alpha \in A_{\text{sticky}} \cap Act_i$, the (unique) edge $\ell \overset{g:\alpha}{\hookrightarrow} \ell'$ in PG_i is marked as *sticky*.
2. An edge $\ell \overset{g:\alpha}{\hookrightarrow} \ell'$ in PG_i is marked *good* if and only if for any action β in some program graph PG_j, $i{\neq}j$, $(\alpha, \beta) \notin D$ and variables modified by β do not occur in guard g.
3. Location ℓ of PG_i is marked as *ample* if all its outgoing edges $\ell \hookrightarrow \ell'$ are marked *good*, but not *sticky*, and the disjunction of the guards of all outgoing edges of ℓ is equivalent to true.

After this (partial) labeling phase, the program graphs PG_i are modified as follows. Let $ample_1, \ldots, ample_n$ be new Boolean variables. These variables are set to true on edges that lead to locations that are marked as ample, and are set to false on all other edges:

4. Edge $\ell \overset{g:\alpha}{\hookrightarrow} \ell'$ in PG_i is replaced by $\ell \overset{g:\hat{\alpha}}{\hookrightarrow} \ell'$, where $\hat{\alpha}$ is the atomic execution of α followed by $ample_i :=$ true whenever ℓ' is marked as *ample*, and α followed by $ample_i :=$ false (atomically) otherwise.

In the next step, the guards of the edges in program graph PG_i are strengthened with propositions about the $ample_i$ boolean variables. Let

$$\begin{aligned} h_i &= \bigwedge_{1 \leqslant j < i} \neg \mathit{ample}_j \wedge \mathit{ample}_i, \quad i = 1, \ldots, n \text{ and} \\ f &= \bigwedge_{1 \leqslant j \leqslant n} \neg \mathit{ample}_j. \end{aligned}$$

The intuitive meaning of proposition h_i is that the enabled actions of process $\mathcal{P}_i$ will serve as an ample set, while f indicates that the state corresponding to the current location is fully expanded. The guards in PG_i are adapted in the following way:

5. Any edge $\ell \overset{g:\alpha}{\hookrightarrow} \ell'$ in PG_i is replaced with the edge $\ell \overset{\widehat{g}:\hat{\alpha}}{\hookrightarrow} \ell'$ where $\widehat{g} = g \ \wedge \ h_i$ if the source location ℓ is marked as *ample*, and $\widehat{g} = g \ \wedge \ f$ otherwise.

The initial condition of $\hat{PG}_i$ is defined as follows:

6. $\hat{g}_{0,i} \ = \ g_{0,i} \wedge \mathit{ample}_i$ if the starting location $\ell_{0,i}$ in PG_i is marked as *ample*, and $\hat{g}_{0,i} = g_{0,i} \wedge \neg \mathit{ample}_i$, otherwise.

These transformations of PG_i yield the program graph $\hat{PG}_i$. The reduced transition system $\hat{TS}$ is obtained as follows:

$$\hat{TS} = TS(\hat{PG}) \quad \text{where} \quad \hat{PG} = \hat{PG}_1 \,|||\, \ldots \,|||\, \hat{PG}_n.$$

Some remarks are in order. First, observe that $\hat{PG}_i$ is not a reduced variant of PG_i, i.e., it contains the locations of PG_i and extends its variables with the auxiliary variables ample_i. State $\hat{s}$ in $\hat{TS}$ has the form $\langle \ell_1, \ldots, \ell_n, \widehat{\eta} \rangle$ where ℓ_i is a location in $\hat{PG}_i$ (and PG_i) and $\widehat{\eta}$ is a variable evaluation for the variables in $PG_1, \ldots, PG_n$ and the auxiliary variables $\mathit{ample}_1, \ldots, \mathit{ample}_n$. In contrast, state s in TS has the form $\langle \ell_1, \ldots, \ell_n, \eta \rangle$ where η assigns values to the variables in $PG_1, \ldots, PG_n$. Nevertheless, $\hat{TS}$ can be considered as a subsystem of TS that results from the transition relation $\Longrightarrow$ defined by the ample sets: on entering state $\langle \ell_1, \ldots, \ell_n, \widehat{\eta} \rangle$ in $\hat{TS}$ it holds $\widehat{\eta}(\mathit{ample}_i) = \text{true}$ if and only if location ℓ_i of PG_i is marked as *ample*. This also applies to the initial state. In particular, the reachable fragment in $\hat{TS}$ will not be larger than the reachable fragment in TS.

Lemma 8.30. Invariant for $\hat{TS}$

For any state $\langle \ell_1, \ldots, \ell_n, \widehat{\eta} \rangle$ *in* $\mathit{Reach}(\hat{TS})$ *and* $0 < i \leqslant n$*:*

$$\widehat{\eta} \models \mathit{ample}_i \ \ \textit{if and only if} \ \ \ell_i \ \textit{is marked as ample.}$$

The above lemma allows us to identify state $\hat{s} = \langle \ell_1, \ldots, \ell_n, \widehat{\eta} \rangle$ in $\hat{TS}$ with state $s = \langle \ell_1, \ldots, \ell_n, \eta \rangle$ in TS where η results from $\widehat{\eta}$ by ignoring the vaulation of the auxiliary

variables $ample_1, \ldots, ample_n$. The action $\hat{\alpha}$ in $\hat{TS}$ can be identified with action α in TS. Due to the transformation of PG_i, $ample(s)$ for global state s in TS consists of actions $\widehat{\alpha}$ for $\alpha \in Act(s)$. This permits treating $ample(\hat{s})$ as a subset of $Act(s)$.

Example 8.31. Program Graph Transformation

Consider the program graphs PG_1 and PG_2 in Figure 8.14. Intuitively, PG_1 and PG_2 both describe a process which counts iteratively from 0 to N where N is a fixed positive integer. Initially, $x = y = 0$ and $b = c =$ false, where these variables are all local.

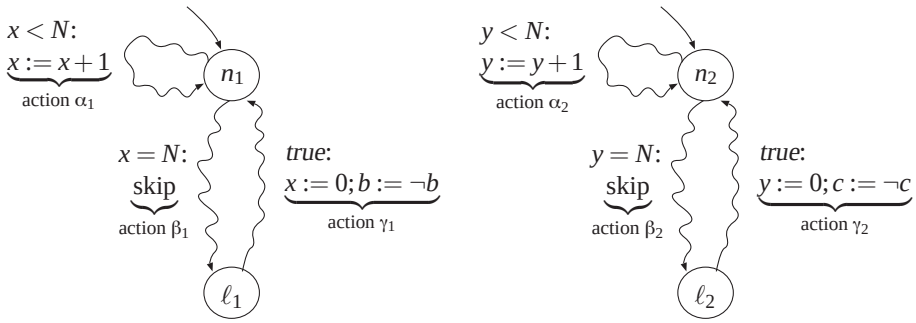


Figure 8.14: Two counting processes as program graph PG_1 (left) and PG_2 (right).

Let $AP = \{ b, c \}$, i.e., the only observable part of the program graphs is the value of b and c. It follows that the actions α_1, β_1, α_2, and β_2 are stutter actions, whereas γ_1 and γ_2 are visible. As all visible actions need to be sticky actions, a first choice for A_{sticky} is:

$$A_{\text{sticky}} \;=\; \{ \gamma_1, \gamma_2 \}.$$

This choice also satisfies (S2), since any cycle in $\hat{TS}$ contains γ_1 or γ_2—a cycle can only occur if both x and y have the same value, and this can only occur by first counting until N and resetting to zero (via a γ action). Actions α_1 and β_1 are independent of $Act_2 = \{ \alpha_2, \beta_2, \gamma_2 \}$, and symmetrically, α_2 and β_2 are independent of $Act_1 = \{ \alpha_1, \beta_1, \gamma_1 \}$.

Consider the transformation from PG_1 and PG_2 into $\hat{PG}_1$ and $\hat{PG}_2$, respectively. This goes along the steps described before. We start by marking the α_i- and β_i-edges as *good* as these actions are independent. The γ_i-edges are marked as *sticky*. Accordingly, the initial locations n_1 and n_2 are marked as *ample* (they only have edges marked as *ample*), while locations ℓ_1 and ℓ_2 are not. Completing steps (4) through (6) yields the program graphs $\hat{PG}_1$ and $\hat{PG}_2$ in Figure 8.15. The initial condition for the modified program graphs is

$$x = y = 0 \;\wedge\; b = c = \text{false} \;\wedge\; ample_1 = ample_2 = \text{true}.$$

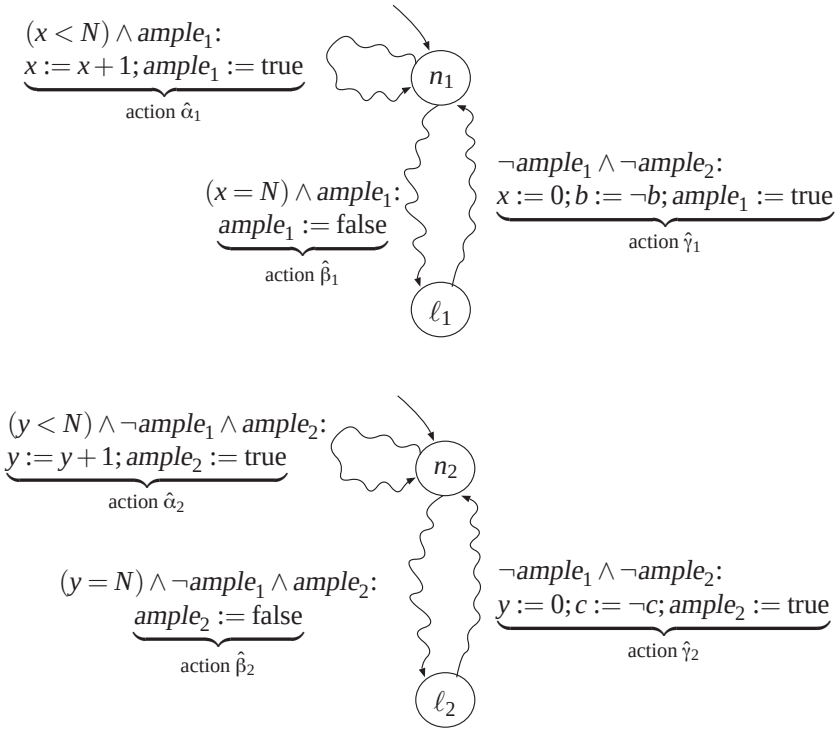


Figure 8.15: Transformed program graphs $\hat{PG}_1$ (top) and $\hat{PG}_2$ (bottom).

We now consider the reduced transition system $\hat{TS} = TS(\hat{PG}_1 \,|||\, \hat{PG}_2)$. Observe that $\hat{TS}$ is smaller than $TS = TS(PG_1 \,|||\, PG_2)$ since it avoids the interleavings of the action sequences α_1^N and α_2^N. Instead, if both processes are in their initial location n_i, then the auxiliary variables $ample_1$ and $ample_2$ evaluate to true. The added conjuncts $h_1 = ample_1$ on the edge $n_1 \hookrightarrow n_1$ and $h_2 = \neg ample_1 \wedge ample_2$ on the edge $n_2 \hookrightarrow n_2$ give a higher priority to $\hat{PG}_1$ and thus enforce performing the action sequence $\widehat{\alpha}_1^N \, \beta_1$ prior to performing $\widehat{\alpha}_2^N$. This can be seen as follows.

For the initial state of $\hat{TS}$,

$$\hat{s} = \langle n_1, n_2, x = y = 0, b = c = \text{false}, ample_1 = ample_2 = \text{true} \rangle,$$

we have

$$Act(\hat{s}) = \{ \, \widehat{\alpha}_1 \, \} \quad \text{while} \quad Act(s) = \{ \, \alpha_1, \alpha_2 \, \}$$

where $s = \langle n_1, n_2, x = y = 0, b = c = \text{false} \rangle$ is the state corresponding to $\hat{s}$ in TS. The same applies to all reachable states s of TS where both processes are in their initial location and $x < N$ and $y < N$. Thus, in all these states, a significant reduction results. Consider

now a state t of the form

$$t = \langle n_1, n_2, x = N, y = k, b = \ldots, c = \ldots \rangle \quad \text{where } 0 \leqslant k < N.$$

in TS. The corresponding state in $\hat{TS}$ is

$$\hat{t} = \langle n_1, n_2, x = N, y = k, b = \ldots, c = \ldots, \mathit{ample}_1 = \mathit{ample}_2 = \text{true} \rangle.$$

We have $Act(\hat{t}) = \{ \widehat{\beta}_1 \}$ which is a proper subset of $Act(t) = \{ \beta_1, \alpha_2 \}$. By symmetry, similar observations apply to the state where both program graphs are in their initial location, $y{=}N$ and $x < N$. State

$$u = \langle \ell_1, n_2, x = N, y = k, b = \ldots, c = \ldots \rangle \quad \text{with} \quad Act(u) = \{ \gamma_1, \alpha_2 \}$$

in TS where $0 \leqslant k < N$ corresponds to the following state in $\hat{TS}$:

$$\hat{u} = \langle \ell_1, n_2, x = N, y = k, b = \ldots, c = \ldots, \mathit{ample}_1 = \text{false}, \mathit{ample}_2 = \text{true} \rangle$$

with $Act(\hat{u}) = \{ \widehat{\alpha}_2 \} \subset Act(u)$. The only fully expanded states are the states of the form

$$v = \langle \ell_1, \ell_2, x = y = N, b = \ldots, c = \ldots \rangle$$

that correspond to

$$\hat{v} = \langle \ell_1, \ell_2, x = y = N, b = \ldots, c = \ldots, \mathit{ample}_1 = \mathit{ample}_2 = \text{false} \rangle.$$

Hence $Act(\hat{v}) = Act(v) = \{ \gamma_1, \gamma_2 \}$. ■

Theorem 8.32. Correctness of the Sticky Set Approach

Let $PG = PG_1 \,|||\, \ldots \,|||\, PG_n$, *and* $\hat{PG} = \hat{PG}_1 \,|||\, \ldots \,|||\, \hat{PG}_n$, $TS = TS(PG)$ *and* $\hat{TS} = TS(\hat{PG})$. *Then*

if A_{sticky} *satisfies (S1) and (S2). then* $\hat{TS} \triangleq TS$.

Proof: Let $PG = PG_1 \,|||\, \ldots \,|||\, PG_n$ and $\hat{PG} = \hat{PG}_1 \,|||\, \ldots \,|||\, \hat{PG}_n$. Assume A_{sticky} satisfies (S1) and (S2). We prove that conditions (A1), (A2), and (A3/4) hold for all states in $\mathit{Reach}(\hat{TS})$. Since (A3/4) implies (A3) and (A4′), and (A4′) implies (A4), it follows by Theorem 8.13 that $TS \triangleq \hat{TS}$.

Let $\hat{s} = \langle \ell_1, \ldots, \ell_n, \widehat{\eta} \rangle$ be a reachable state in $\hat{TS}$ with corresponding state $s = \langle \ell_1, \ldots, \ell_n, \eta \rangle$ in TS. Consider two cases.

1. None of the locations ℓ_i is marked as *ample*. Then PG and $\hat{PG}$ have the same outgoing edges in (global) location $\langle \ell_1, \ldots, \ell_n \rangle$. Moreover, $\eta \models g$ if and only if $\widehat{\eta} \models g$ for any guard g that occurs in PG. (Recall that η and $\widehat{\eta}$ agree on the original program variables.) Hence, state $\hat{s}$ is fully expanded, i.e.,

$$Act(\hat{s}) = \{ \widehat{\alpha} \mid \alpha \in Act(s) \}.$$

   It follows that conditions (A1), (A2), and (A3/4) are then fulfilled.

2. ℓ_i is marked as *ample* and $\ell_1, \ldots, \ell_{i-1}$ are not marked as *ample*. By Lemma 8.30, $\widehat{\eta} \models h_i$. For $1 \leqslant j < i$, any outgoing edge from ℓ_j in $\hat{PG}_j$ is either guarded by h_j or f. As $\widehat{\eta}(ample_j) = \text{true}$, we get $\widehat{\eta} \not\models h_j$. Similarly, as $\widehat{\eta}(ample_i) = \text{true}$, we have $\widehat{\eta} \not\models f$. Thus, for $1 \leqslant j < i$, none of the outgoing edges from ℓ_j in $\hat{PG}_j$ is enabled in state $\hat{s}$. Similarly, for $i < j \leqslant n$ none of the outgoing edges from ℓ_j in PG_j is enabled since these edges are guarded by h_j or f as both require $ample_i$ to be false. Thus:

$$ample(\hat{s}) \subseteq \{ \widehat{\alpha} \mid \alpha \in Act_i(s) \}.$$

   Since ℓ_i is marked as *ample*, any action α of an edge $\ell_i \overset{g:\alpha}{\hookrightarrow} \ell_i'$ in PG_i is marked as *good*, but not marked as *sticky*. In particular, $Act_i(s) \cap A_{\text{sticky}} = \varnothing$ which yields:

$$ample(\hat{s}) \cap A_{\text{sticky}} = \varnothing$$

   (where we identify α and $\widehat{\alpha}$). Thus, condition (A3/4) is fulfilled. Since all actions in $ample(\hat{s})$ are marked as *good*, conditions (A2.1) and (A2.2) are fulfilled. By Lemma 8.27 (page 633) it follows that the dependency condition (A2) holds. The nonemptiness condition (A1) is fulfilled since the mark *ample* ensures that the disjunction of the guards of all outgoing edges of ℓ_i is true.

■

Computing Sticky Sets It remains to explain how a set A_{sticky} satisfying the conditions (S1) and (S2) can be determined by a static analysis of the program graph. The obvious choice $A_{\text{sticky}} = Act$ would evidently suffice, but then (A3/4) would not allow for any reduction of the state space. The goal is to obtain a set A_{sticky} satisfying (S1) and (S2) that is as small as possible. Initially, we set $A_{\text{sticky}} = Vis$, the set of visible actions. This ensures the visibility condition (S1). In order to establish (S2), actions are added until any cycle in $\hat{TS}$ contains an action in A_{sticky}. This is done by a static analysis of the program graph, attempting to keep the cardinality of A_{sticky} minimal.

Notation 8.33. Control Path and Control Cycle

A *control path* of PG_i is a sequence of edges in PG_i of the form

$$\ell_0 \xleftarrow{g_1:\alpha_1} \ell_1 \xleftarrow{g_2:\alpha_2} \ldots \xleftarrow{g_k:\alpha_k} \ell_k.$$

A control path in PG_i is a *control cycle* if $\ell_k = \ell_0$ and $k > 0$. ■

Notation 8.34. PG_i**-Projection**

Let $\varrho = s_0 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_k} s_k$ be an execution fragment in TS where $s_j = \langle \ell_1^j, \ldots, \ell_n^j, \eta^j \rangle$ for $0 \leqslant j \leqslant k$. The PG_i*-projection* of ϱ is the control path of PG_i resulting from ϱ by (1) removing all transitions $s_{j-1} \xrightarrow{\alpha_j} s_j$ where $\alpha_j \notin Act_i$ and (2) replacing the transitions $s_{i-1} \xrightarrow{\alpha_i} s_i$ where $\alpha_j \in Act_i$ with the corresponding edge $\ell_i^{j-1} \xleftarrow{g_j:\alpha_j} \ell_i^j$. ■

The PG_i-projection of an execution fragment in TS is the control path in PG_i that has been taken in ϱ. Vice versa, any execution fragment in TS arises by the combination of control paths in some of the program graphs $PG_1, \ldots, PG_n$, possibly in an interleaved way.

Example 8.35. Projections

Consider $TS = TS(PG_1 \,|||\, PG_2)$ where PG_1, PG_2 are depicted in Figure 8.14 (page 639). For $N = 2$, the execution fragment

$$\begin{aligned}
&\langle n_1, n_2, x = 0, y = 0, b = c = \text{false} \rangle \quad \xrightarrow{\alpha_1} \\
&\langle n_1, n_2, x = 1, y = 0, b = c = \text{false} \rangle \quad \xrightarrow{\alpha_1} \\
&\langle n_1, n_2, x = 2, y = 0, b = c = \text{false} \rangle \quad \xrightarrow{\alpha_2} \\
&\langle n_1, n_2, x = 2, y = 1, b = c = \text{false} \rangle \quad \xrightarrow{\beta_1} \\
&\langle \ell_1, n_2, x = 2, y = 0, b = c = \text{false} \rangle
\end{aligned}$$

yields the PG_1-projection $n_1 \xleftarrow{x<2:\alpha_1} n_1 \xleftarrow{x<2:\alpha_1} n_1 \xleftarrow{x=2:\beta_1} \ell_1$ and the PG_2-projection $n_2 \xleftarrow{y<2:\alpha_1} n_2$. Note that the PG_i-projection of ϱ might have length 0 (i.e., solely consists of the location ℓ_i^0) in which case ϱ does not take any edge of PG_i. ■

The following lemma is obvious:

Lemma 8.36. Global Cycles vs. Control Cycle

For each cycle $s_0 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_m} s_m$ *in TS and for any* $0 < j \leqslant n$*:*

$$\textit{if } \{ \alpha_1, \ldots, \alpha_m \} \cap Act_j \neq \varnothing, \textit{ then the } PG_j\textit{-projection is a control cycle.}$$

The converse of Lemma 8.36 does not hold, since, e.g., the control cycle $n_1 \xrightarrow{x<N:\alpha_1} n_1$ in the program graph PG_1 of Figure 8.14 does not induce a cycle in TS since the value of x is strictly increasing in each iteration.

By Lemma 8.36, the cycle-breaking condition (S2) is obviously fulfilled if A_{sticky} contains at least one action of any control cycle in $PG_1, \ldots, PG_n$. This is guaranteed as follows. Let $A_{\text{sticky}} = \mathit{Vis}$, the set of visible actions. This ensures (S1). As each visible action is in A_{sticky}, it suffices for establishing (S2) that any cycle that does not contain a visible action is considered. To that end, all edges from PG_i with a visible action are removed. By means of a depth-first search of the resulting program graph PG_i', say, control cycles are determined. Whenever a backward edge $\ell \overset{g:\alpha}{\hookrightarrow} \ell'$ is found, α is added to the sticky set. Applying this DFS-based approach to each program graph PG_i yields A_{sticky}. From Lemma 8.36 together with the fact that for any control cycle in PG_i' the depth-first search will find a backward edge on this cycle, it follows that A_{sticky} satisfies (S2). Hence, A_{sticky} contains at least one action of each control cycle in PG_i'. The remaining control cycles in PG_i contain at least one visible action, and thus, an action in $\mathit{Vis} \subseteq A_{\text{sticky}}$.

This strategy often generates a rather large A_{sticky} set. For instance, for the program graphs in Figure 8.14, the set

$$A_{\text{sticky}} = \underbrace{\{\, \gamma_1, \gamma_2 \,\}}_{=\mathit{Vis}} \cup \{\, \alpha_1, \alpha_2 \,\}$$

is obtained because of the control cycles $n_1 \xrightarrow{x<N:\alpha_1} n_1$ and $n_2 \xrightarrow{y<N:\alpha_2} n_2$. However, these control cycles do not correspond to cycles in TS since each execution of α_1 strictly increases the value of x. The same applies to α_2 and variable y. In order to find control cycles that do not correspond to a *global cycle*, i.e., a cycle in TS, we impose for each program variable x:

a transitive, irreflexive relation $\prec_x$ on the domain of x.

For example, for integer variable x, the relation $\prec_x$ could be the natural order $<$ or the reverse natural order $>$. For Boolean variable x, the order with false $\prec_x$ true and true $\not\prec_x$ false is appropriate. The relation $\prec_x$ should be chosen such that the classification of the actions into incrementing, decrementing, neutral, or complex actions (see below) is algorithmically simple. For variable x and order $\prec_x$, the effect of action $\alpha \in \mathit{Act}$ is said to be

- *incrementing* on x if $\eta(x) \prec_x \mathit{Effect}(\eta, \alpha)(x)$ for each valuation η,
- *decrementing* on x if $\mathit{Effect}(\eta, \alpha)(x) \prec_x \eta(x)$ for each valuation η,

- *neutral* on x if $\mathit{Effect}(\eta,\alpha)(x) \not\prec_x \eta(x)$ and $\eta(x) \not\prec_x \mathit{Effect}(\eta,\alpha)(x)$, for each valuation η.
- *complex* in all other cases.

For instance, for integer variable x where $\prec_x$ is the natural order $<$ we have: $x := x+2$ has an incrementing effect on x, action $x := x-5$ has a decrementing effect on x, $y := y-5$ is neutral on x, and $x := y$ has a complex effect on x.

A control path $\ell_0 \xrightarrow{g_1:\alpha_1} \ldots \xrightarrow{g_k:\alpha_k} \ell_k$ has incrementing effect on x if (1) the effect of α_j on x is incrementing for some $j \in \{1,\ldots,k\}$ and (2) for each $j \in \{1,\ldots,r\}$, $j \neq k$, the effect of α_j on x is either incrementing or neutral. Control paths with a decrementing effect on x are defined analogously. For an execution fragment

$$s_0 \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_m} s_m$$

such that for some $i \in \{1,\ldots,n\}$

- the PG_i-projection is a control path $\ell_0 \xrightarrow{g_1:\alpha_1} \ell_1 \ldots \xrightarrow{g_r:\alpha_r} \ell_r$ which has an incrementing effect on x, and
- the PG_j-projection for $j \neq i$ does not contain an action which has either a decrementing or complex effect on x,

the value of x is strictly increasing with respect to the order $\prec_x$. The transitivity of $\prec_x$ yields $v_0 \prec_x v_m$, where v_0 and v_m are the value of x in the state s_0 and s_m, respectively. Since $\prec_x$ is irreflexive, $v_0 \neq v_m$. Thus, the given execution fragment cannot be a cycle in TS, i.e., $s_0 \neq s_m$. A similar observation holds for control cycles with a decrementing effect on x.

Example 8.37. Control Cycles vs. Global Cycles

Consider the program graphs in Figure 8.16 where $N > 1$ is a fixed integer, x and y are nonnegative integer variables, and b, c Boolean variables. Program graph PG_1 has the control cycle

$$n_1 \xrightarrow{x<N:x:=x+1} \ell_1 \xrightarrow{b:=\neg b} n_1.$$

For $\prec_x = <$, this control cycle has an incrementing effect on x. None of the actions $\alpha_2, \beta_2, \gamma_2$ in PG_2 modify the value of x, i.e., these actions have a neutral effect on x. Thus, there is no cycle in TS for which the PG_1-projection coincides with the above control cycle.

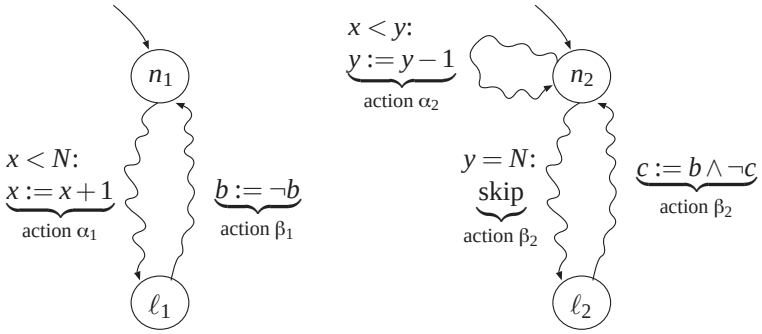


Figure 8.16: Program graphs PG_1 (left) and PG_2 (right).

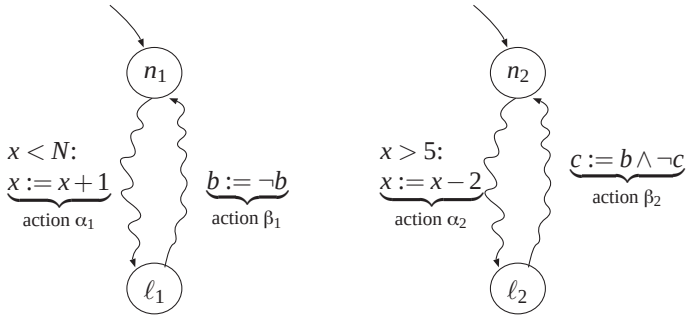


Figure 8.17: Program graphs PG_1 (left) and PG_2 (right).

Now consider the program graphs in Figure 8.17. The control cycle

$$n_1 \xrightarrow{x<N:x:=x+1} \ell_1 \xrightarrow{b:=\neg b} n_1$$

in PG_1 has an incrementing effect on x. However, action α_2 in PG_2 has a decreasing effect on x. In fact, there is a global cycle in $TS(PG_1 \,|||\, PG_2)$ of the form

$$\begin{aligned}
s \;=\; & \langle n_1, n_2, x=0, y=0, b=\text{false}, c=\text{false}\rangle \xrightarrow{\alpha_1} \\
& \langle \ell_1, n_2, x=1, y=0, b=\text{false}, c=\text{false}\rangle \xrightarrow{\beta_1} \\
& \langle n_1, n_2, x=1, y=0, b=\text{true}, c=\text{false}\rangle \xrightarrow{\alpha_1} \\
& \langle \ell_1, n_2, x=2, y=0, b=\text{true}, c=\text{false}\rangle \xrightarrow{\beta_1} \\
& \langle n_1, n_2, x=2, y=0, b=\text{false}, c=\text{false}\rangle \xrightarrow{\alpha_2} \\
& \langle n_1, \ell_2, x=0, y=0, b=\text{false}, c=\text{false}\rangle \xrightarrow{\beta_2} \\
& \langle n_1, n_2, x=0, y=0, b=\text{false}, c=\text{false}\rangle \;=\; s
\end{aligned}$$

where the PG_1-projection yields the (duplication of the) above control cycle. ■

The goal is now to compute A_{sticky} by starting with $A = \mathit{Vis}$ (as before) and adding further actions to break any control cycle of program graph PG_i, say, that only contains stutter actions, except for actions that have an incrementing or a decrementing effect on some variable x and for which there is no action in PG_j, $i \neq j$ with an opposite effect on x.

Notation 8.38. Opposite Actions

Actions α and $\beta \in \mathit{Act}$ are *opposite* if there exists a variable x such that α has either an incrementing or complex effect on x and β has either a decrementing or complex effect on x, or vice versa. Let

$$\mathit{Opp}(\alpha) = \{\, \beta \in \mathit{Act} \mid \alpha \text{ and } \beta \text{ are opposite} \,\}.$$

Note that any action α with complex effect on some variable is opposite to itself, in which case $\alpha \in \mathit{Opp}(\alpha)$. ■

The computation of A_{sticky} proceeds as outlined in Algorithm 42 on page 648. Here, an action α is called *monotonic* on x if it has either an incrementing or decrementing effect on x.

Example 8.39. Computing A_{sticky}

Reconsider the program graphs PG_1 and PG_2 in Figure 8.16 (page 646). Let $\prec_x = <= \prec_y$ and false $\prec$ true for the Boolean variables b and c. Moreover, we assume that AP only refers to variable y, but not to x, b, or c. Then, action α_2 is visible and all other actions are stutter actions. Hence, Algorithm 42 starts with $A_{\text{sticky}} = \mathit{Vis} = \{\, \alpha_2 \,\}$. Removal of all visible edges yields the program graphs in Figure 8.18.

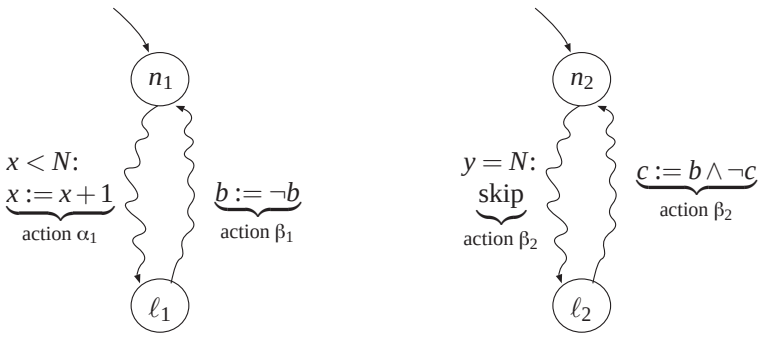


Figure 8.18: Program graphs PG_1 (left) and PG_2 (right) after removal of visible edges.

The algorithm obtains $M_1 = \{\, \alpha_1 \,\}$ since α_1 has an incrementing effect on x, but there

Algorithm 42 Computation of A_{sticky}

Input: program graphs $PG_1, \ldots PG_n$
Output: $A_{\text{sticky}} \subseteq Act$ satisfying (S1) and (S2)

$A_{\text{sticky}} := Vis$; (* establish (S1) *)

for $i = 1$ **to** n **do**
  $PG_i := PG_i$ where any edge $\ell \overset{g:\alpha}{\hookrightarrow} \ell'$ with $\alpha \in Vis$ is removed;
od

for $i = 1$ **to** n **do**
  $M_i := \{\, \alpha \in Act_i \mid \alpha \text{ is monotonic } \wedge\, Opp(\alpha) \subseteq \bigcup_{j<k\leqslant n} Act_k \,\}$;
  $PG_i := PG_i$ where any edge $\ell \overset{g:\alpha}{\hookrightarrow} \ell'$ with $\alpha \in M_j$ is removed;
od

for $i = 1$ **to** n **do**
  perform a DFS on PG_i;
  $A_{\text{sticky}} := A_{\text{sticky}} \cup \{\, \alpha \mid \ell \overset{g:\alpha}{\hookrightarrow} \ell' \text{is a backward edge in } PG_i \,\}$;
od

return A_{sticky} (* A_{sticky} fulfills (S1) and (S2) *)

is no opposite action. Hence, the α_1-edge from n_1 is ℓ_1 will be removed. The resulting program graph is acyclic; no backward edge is found.

We obtain $M_2 = \varnothing$ as there is no monotonic action in the modified PG_2. Hence, one of the actions β_2 or γ_2 becomes sticky (i.e., is added to A_{sticky}), depending on the starting location for the depth-first search. The result of Algorithm 42 is thus either $A_{\text{sticky}} = \{ \alpha_2, \beta_2 \}$ or $A_{\text{sticky}} = \{ \alpha_2, \gamma_2 \}$. ■

Lemma 8.40. Soundness of Algorithm 42

The set $A_{sticky} \subseteq Act$ returned by Algorithm 42 satisfies (S1) and (S2).

Proof: The visibility condition (S1) obviously holds since initially $A_{\text{sticky}} = \mathit{Vis}$, and no actions are removed from A_{sticky}. Now consider (S2). Let ϱ be a global cycle of the form

$$s_0 \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_m} s_m = s_0.$$

If some of the actions $\beta_1, \ldots, \beta_m$ are visible, then the cycle-breaking condition (S2) holds since $\mathit{Vis} \subseteq A_{\text{sticky}}$. Assume $\beta_1, \ldots, \beta_m$ are stutter actions. We prove that there exists some j such that (some of) the actions in $(\{\beta_1, \ldots, \beta_m\} \cap \mathit{Act}_j) \setminus M_i$ yield a control cycle ϑ in PG''_j (the variant of PG_j obtained after removing all edges with action in M_j). From this result, it follows that the depth-first search in PG''_j will classify one of the edges in ϑ as a backward edge, and hence, adds at least one of the actions in $\{\beta_1, \ldots, \beta_m\} \cap \mathit{Act}_j$ to A_{sticky}. This yields $A_{\text{sticky}} \cap \{\beta_1, \ldots, \beta_m\} \neq \varnothing$.

Let J be the set of indices $j \in \{ 1, \ldots, n \}$ with $\{ \beta_1, \ldots, \beta_m \} \cap \mathit{Act}_j \neq \varnothing$. Clearly, $J \neq \varnothing$. Let $j = \max J$ and assume there is no control cycle in PG''_j where the underlying action set is contained in

$$(\{\beta_1, \ldots, \beta_m\} \cap \mathit{Act}_j) \setminus M_j.$$

Since the PG_j-projection of the given global cycle ϱ is a control cycle ϑ in PG'_j (Lemma 8.36 on page 643), the control cycle ϑ in PG'_j must contain an action $\beta_i \in M_j$. Otherwise, ϑ is a control cycle in PG''_j built by actions in $(\{\beta_1, \ldots, \beta_m\} \cap \mathit{Act}_j) \setminus M_j$.

By definition of M_j, action β_i is monotonic. Since $\beta_1, \ldots, \beta_m$ forms a global cycle in TS, there must be another action β_h, $h \neq i$, on this cycle with an opposite effect:

$$\beta_h \in \mathit{Opp}(\beta_i).$$

Again, by definition of M_j, the actions that are opposite to β_i belong to Act_k for some $k > j$. Thus, $\beta_h \in \mathit{Act}_k$ for some $k > j$. But then, $k \in J$ and $k > j$, which contradicts $j = \max J$. ■

8.3 The Branching-Time Ample Set Approach

The previous section presented conditions for the ample sets that ensure the stutter trace equivalence of TS and $\hat{TS}$. Hence, these conditions are sound for verifying $\mathrm{LTL}_{\setminus\bigcirc}$. This section treats partial order reduction for the branching-time temporal logics $\mathrm{CTL}_{\setminus\bigcirc}$ and $\mathrm{CTL}^*_{\setminus\bigcirc}$. The aim is to adopt the ample set approach, and adapt it such that rather than establishing $TS \triangleq \hat{TS}$ we obtain that $TS \approx^{div} \hat{TS}$. As $\approx^{div}$ coincides with the logical equivalence of $\mathrm{CTL}_{\setminus\bigcirc}$ and $\mathrm{CTL}^*_{\setminus\bigcirc}$ (see Section 7.8.3 on page 560), this yields a sound approach for verifying these logics. We first argue by means of an example that the conditions (A1) through (A4) do not suffice for $\mathrm{CTL}_{\setminus\bigcirc}$.

Example 8.41. Conditions (A1)-(A4) Are Insufficient for $CTL_{\setminus\bigcirc}$

Consider the transition system TS in Figure 8.19 (left) over $AP = \{a, b, c\}$. Actions α and δ are independent of $\{\beta, \gamma\}$ and β, γ are stutter actions. Let $ample(s_0) = \{\beta, \gamma\}$, and $ample(s) = Act(s)$ for all states $s \neq s_0$; see the reduced transition system $\hat{TS}$ in Figure 8.19 (right).

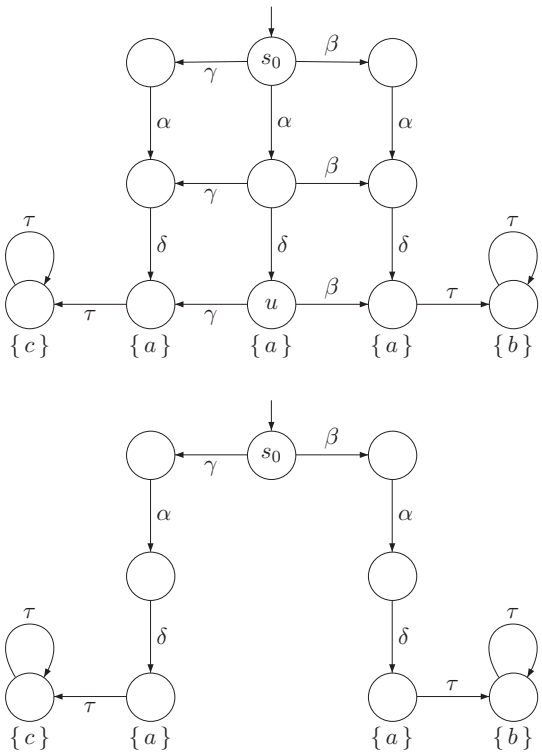


Figure 8.19: Reduction satisfies (A1)-(A4), but $TS \not\equiv_{CTL_{\setminus\bigcirc}} \hat{TS}$.

Consider the $\mathrm{CTL}_{\setminus\bigcirc}$ formula:

$$\Phi = \forall\Box\Big(a \rightarrow (\forall\Diamond b \vee \forall\Diamond c)\Big).$$

Formula Φ asserts that for each reachable a-state s either any path from s eventually reaches a b-state or any path from s eventually reaches a c-state. Hence $\hat{TS} \models \Phi$ as any a-state in TS has either a direct b- or c-successor, but $TS \not\models \Phi$ as state $u \not\models \forall\Diamond b \vee \forall\Diamond c$. The latter follows from the fact that

$$\mathit{Traces}_{TS}(u) = \{\, \{a\}\{a\}\{b\}^\omega, \{a\}\{a\}\{c\}^\omega \,\}.$$

As $\hat{TS} \models \Phi$ and $TS \not\models \Phi$, we get $TS \not\approx^{div} \hat{TS}$. In fact, we have

$$\beta(s_0) \not\approx^{div} \gamma(s_0) \not\approx^{div} \alpha(s_0)$$

as $\beta(s_0)$ can reach a b-state, while $\gamma(s_0)$ cannot, and vice versa $\gamma(s_0)$ can reach a c-state, while $\beta(s_0)$ cannot. Moreover, $\alpha(s_0)$ (and s_0) can reach both a b- and a c-state. The same holds for the α-successor s_1 of s_0. Since $\hat{TS}$ does not contain s (or any equivalent state), $TS \not\approx^{div} \hat{TS}$.

Since $s_0 \approx^{div} \alpha(s_0)$, the reduction obtained by $\mathit{ample}(s_0) = \{\, \alpha \,\}$, and $\mathit{ample}(s) = \mathit{Act}(s)$ for any state $s \neq s_0$ yields the reduced transition system $\hat{TS}$ in Figure 8.20). Now we have $\hat{TS} \approx^{div} TS$. ■

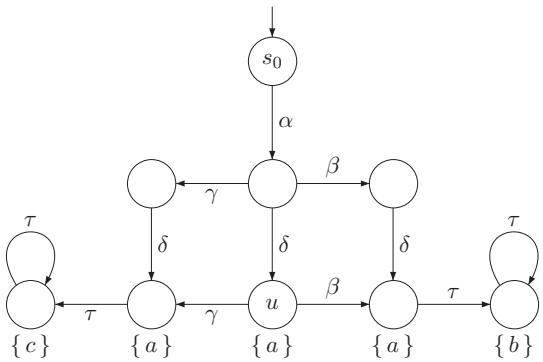


Figure 8.20: A sound partial order reduction for $\mathrm{CTL}^*_{\setminus\bigcirc}$.

The task is now to provide sufficient ample set conditions that ensure $TS \approx^{div} \hat{TS}$. Since $\triangleq$ is strictly coarser than $\approx^{div}$, the conditions (A1) though (A4) are *extended* with a condition ensuring that the branching structure of TS is preserved.

In Example 8.41, $\mathit{ample}(s_0) = \{\,\alpha\,\}$ resulted in a correct reduction. The soundness of this reduction follows from the fact that $s_0 \approx^{div} \alpha(s_0)$. The γ- and β-transitions in state s_0 can be mimicked in the reduced transition system by first performing the stutter step $s_0 \rightarrow \alpha(s_0)$ followed by γ or β, respectively. It turns out that this strategy is generally applicable. Whenever (A1)-(A4) allow for choosing a singleton action set $\mathit{ample}(s) = \{\,\alpha\,\}$, then state $s \approx^{div} \alpha(s)$. A formal proof of this statement will be provided later. An intuitive explanation is that all other enabled actions in s (i.e., all actions in $\mathit{Act}(s) \setminus \{\,\alpha\,\}$) are enabled in $\alpha(s)$ and lead to states that are equivalent to the corresponding successors of α. This suggests extending conditions (A1) through (A4) by

(A5) Branching condition

If $\mathit{ample}(s) \neq \mathit{Act}(s)$, then $|\mathit{ample}(s)| = 1$.

Example 8.42.

Consider again the transition system TS in Figure 8.19 (left). The reduction shown in Figure 8.20 fulfills conditions (A1) through (A5). The nonemptiness condition (A1) and the branching condition (A5) are obviously fulfilled, since $\mathit{ample}(s_0) = \{\,\alpha\,\}$ is a singleton and all other states are fully expanded. The stutter condition (A3) holds since α is a stutter action. The cycle condition (A4) is obvious since all states on a cycle (the states equipped with a self-loop) are fully expanded. Finally, the dependency condition (A2) is satisfied since α is independent of $\{\,\beta, \gamma\,\}$.

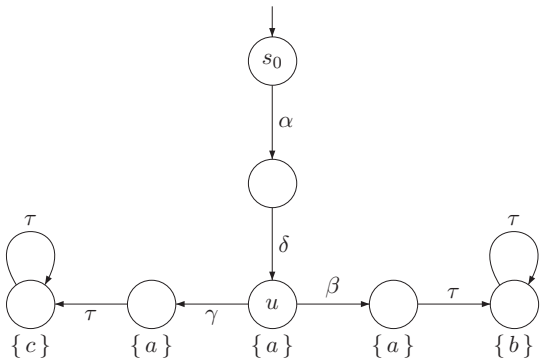


Figure 8.21: Partial order reduction that violates (A3).

Now consider the alternative selection of ample sets: $\mathit{ample}(s_0) = \{\,\alpha\,\}$ and $\mathit{ample}(\alpha(s_0)) = \{\,\delta\,\}$, while all other states are fully expanded. These ample sets yield the reduced system $\hat{TS}$ shown in Figure 8.21. Observe that $TS \not\approx^{div} \hat{TS}$ since $\hat{TS}$ does not contain a state that is equivalent to $\beta(s_0)$—there is no state labeled with $\varnothing$ that can reach a c-state but

(A1) **Nonemptiness condition**

$\varnothing \neq \mathit{ample}(s) \subseteq \mathit{Act}(s)$

(A2) **Dependency condition**

Let $s \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_n} s_n \xrightarrow{\alpha} t$ be a finite execution fragment in TS.
If α depends on $\mathit{ample}(s)$, then $\beta_i \in \mathit{ample}(s)$ for some $0 < i \leqslant n$.

(A3) **Stutter condition**

If $\mathit{ample}(s) \neq \mathit{Act}(s)$, then any $\alpha \in \mathit{ample}(s)$ is a stutter action.

(A4) **Cycle condition**

For any cycle $s_0\, s_1\, \ldots\, s_n$ in $\hat{TS}$ and $\alpha \in \mathit{Act}(s_i)$, for some $0 < i \leqslant n$, there exists $j \in \{1, \ldots, n\}$ such that $\alpha \in \mathit{ample}(s_j)$.

(A5) **Branching condition**

If $\mathit{ample}(s) \neq \mathit{Act}(s)$, then $|\mathit{ample}(s)| = 1$.

Figure 8.22: Requirements on the ample set of state s for $\mathrm{CTL}^*_{\setminus\bigcirc}$.

not a b-state. The same applies to $\gamma(s_0)$. In fact, TS and $\hat{TS}$ can be distinguished by the $\mathrm{CTL}_{\setminus\bigcirc}$ formula

$$\Phi = \forall\Diamond\,(a \ \wedge\ \exists\Diamond\, b \ \wedge\ \exists\Diamond\, c)$$

which holds for $\hat{TS}$, but not for TS. In fact, stutter condition (A3) is violated since the chosen action δ in state $\alpha(s_0)$ is not a stutter action. ■

The ample set conditions are summarized in Figure 8.22. The remainder of this section is concerned with the proof of the correctness theorem:

Theorem 8.43. Correctness of the Ample Set Approach for CTL$^*_{\setminus\bigcirc}$

For action-deterministic, finite transition system TS without terminal states:

If conditions (A1) through (A5) are satisfied, then $TS \approx^{div} \hat{TS}$.

Since $\approx^{div}$ coincides with $\mathrm{CTL}^*_{\setminus\bigcirc}$ equivalence (see Theorem 7.128, page 561), TS and $\hat{TS}$ are $\mathrm{CTL}^*_{\setminus\bigcirc}$-equivalent, provided that (A1)-(A5) hold.

For the remainder of this section, let $TS = (S, Act, \rightarrow, I, AP, L)$ be an action-deterministic, finite transition system without terminal states and assume that the ample sets satisfy (A1)-(A5). As before, $\hat{TS}$ denotes the reduced transition system that arises from the reachable fragment of TS under the transition relation $\Longrightarrow$ given by

$$\frac{s \xrightarrow{\alpha} s' \ \wedge \ \alpha \in \mathit{ample}(s)}{s \overset{\alpha}{\Longrightarrow} s'}.$$

The initial states of $\hat{TS}$ are the initial states in TS. $\hat{S}$ denotes the state space of $\hat{TS}$, i.e., all states in S that are reachable (via $\Longrightarrow$) from some initial state $s_0 \in I$.

It will be shown that (A1) through (A5) ensure that there exists a *normed bisimulation* that relates TS and $\hat{TS}$. The concept of normed bisimulations has been introduced in Definition 7.120 (page 552). The correctness theorem then follows from the fact that normed bisimulation is strictly finer than $\approx^{div}$.

Let us briefly recall the main concepts of normed bisimulations. A normed bisimulation for $(TS, \hat{TS})$ is a triple $(\mathcal{R}, \nu_1, \nu_2)$ where $\mathcal{R} \subseteq S \times \hat{S}$ and $\nu_1, \nu_2 : S \times \hat{S} \to \mathbb{N}$ are functions such that for all pairs $(s_1, s_2) \in \mathcal{R}$ the following conditions are fulfilled:

(NI) $L(s_1) = L(s_2)$, and

(NII) For all $s_1' \in \mathit{Post}(s_1)$, at least one of the following three conditions holds:

(N1) there exists $s_2' \in \mathit{Post}(s_2)$ with $(s_1', s_2') \in \mathcal{R}$, or

(N2) $(s_1', s_2) \in \mathcal{R}$ and $\nu_1(s_1', s_2) < \nu_1(s_1, s_2)$, or

(N3) there exists $s_2' \in \mathit{Post}(s_2)$ with $(s_1, s_2') \in \mathcal{R}$ and $\nu_2(s_1, s_2') < \nu_2(s_1, s_2)$,

and the analogous conditions for all $s_2' \in \mathit{Post}(s_2)$ with the "swapped" norm functions ν_2^- (instead of ν_1 in (N2)) and ν_1^- (instead of ν_2 in (N3)) given by $\nu_i^-(u, v) = \nu_i(v, u)$ for all $u \in \hat{S}$ and $v \in S$.

In addition, it is required that each initial state of TS is related by $\mathcal{R}$ to some initial state of $\hat{TS}$, and vice versa. Intuitively, $\nu_1(s, \hat{s})$ is an upper bound on the "allowed" number of stutter steps from s that cannot be mimicked by transitions of $\hat{s}$, Natural $\nu_2(s, \hat{s})$ serves as a counter for the number of stutter steps that $\hat{s}$ may perform to reach a state where the visible (nonstutter) steps of s can be simulated.

To define $\mathcal{R}$ and the norm functions ν_1, ν_2 we need some further notations.

Definition 8.44. Forming Path, Relation $\lhd$

Let TS be a finite, action-deterministic transition system and $s, s' \in S$. A *forming path* from s to s' is a finite execution fragment ϱ in TS of the form $s_0 \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_n} s_n$ where

- $s = s_0$ and $s_n = s'$,
- $\beta_1, \ldots, \beta_n$ are stutter actions, and
- for $1 \leqslant i < n$, the singleton action set $\{\beta_{i+1}\}$ fulfills the dependence condition (A2) for state s_i. That is, for any finite execution fragment $s_i \xrightarrow{\alpha_1} t_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_m} t_m \xrightarrow{\gamma} \ldots$ where γ is dependent on β_{i+1}, there exists $j \in \{1, \ldots, m\}$ such that $\alpha_j = \beta_{i+1}$.

We write $s \lhd s'$ iff there exists a forming path from s to s'. ■

The following simple properties of forming paths and the relation $\lhd$ will be used quite often in the following argumentation. First, the relation $\lhd$ is transitive and reflexive (even though, in general, nonsymmetric). The reflexivity is clear as each execution fragment of length 0 yields a forming path. The transitivity follows from the fact that forming paths from s to s' and from s' to s'' can be concatenated resulting in a forming path from s to s''. Secondly, if $s_0 \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_n} s_n$ is a forming path from $s = s_0$ to $s' = s_n$ of length n, then $s_i \lhd s_j$ for $0 \leqslant i \leqslant j \leqslant n$.

Lemma 8.45. Properties of Forming Paths (Part 1)

Let $s_0 \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_n} s_n$ be a forming path from $s = s_0$ to $s' = s_n$ and α be a stutter action such that $s \xrightarrow{\alpha} t$ is a transition in TS. Then, we have:

(a) If $\alpha \notin \{\beta_1, \ldots, \beta_n\}$, then α is independent of $\{\beta_1, \ldots, \beta_n\}$ and there exists a forming path $t_0 \xrightarrow{\beta_1} t_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_n} t_n$ from $t = t_0$ to t_n such that $s_i \xrightarrow{\alpha} t_i$ for all indices $i \in \{1, \ldots, n\}$.

(b) If $\alpha = \beta_j$ and $\alpha \notin \{\beta_1, \ldots, \beta_{j-1}\}$ for some $j \in \{1, \ldots, n\}$, then $t \lhd s'$ and there exists a forming path from t to s' with the action sequence $\beta_1 \ldots \beta_{j-1} \beta_{j+1} \ldots \beta_n$.

Proof: We first prove (a). The independency of α from $\{\beta_1, \ldots, \beta_n\}$ is immediate from the definition of forming paths that require (A2) to hold on all states of a forming path. Lemma 8.15 (page 613) yields the existence of an execution fragment

$$t_0 \xrightarrow{\beta_1} t_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_n} t_n$$

from $t = t_0$ to t_n such that $s_i \xrightarrow{\alpha} t_i$ for all indices $i \in \{1, \ldots, n\}$. It remains to show that this execution fragment is a forming path. Obviously, $\beta_1, \ldots, \beta_n$ are stutter actions. We now have to show that the dependency condition (A2) holds for all states t_i and the singleton action sets $\{\beta_{i+1}\}$. This is obvious since whenever

$$t_i \xrightarrow{\gamma_1} v_1 \xrightarrow{\gamma_2} \ldots \xrightarrow{\gamma_k} v_k \xrightarrow{\gamma} v$$

is an execution fragment where γ depends on β_i, then

$$s_i \xrightarrow{\alpha} t_i \xrightarrow{\gamma_1} v_1 \xrightarrow{\gamma_2} \ldots \xrightarrow{\gamma_k} v_k \xrightarrow{\gamma} v$$

is an execution fragment from s_i. Since (A2) holds for s_i and the singleton set $\{\beta_{i+1}\}$ and γ depends on β_{i+1}, one of the actions $\alpha, \gamma_1, \ldots, \gamma_k$ agrees with β_{i+1}. Since $\alpha \notin \{\beta_1, \ldots, \beta_n\}$, we have $\gamma_j = \beta_{i+1}$ for some $j \in \{1, \ldots, k\}$. Hence, (A2) holds for t_i and the action set $\{\beta_{i+1}\}$.

We now consider (b) and suppose that $\alpha = \beta_j$ and $\alpha \notin \{\beta_1, \ldots, \beta_{j-1}\}$. Applying (a) to the forming path

$$s = s_0 \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_{j-1}} s_{j-1}$$

yields the existence of a forming path

$$t_0 \xrightarrow{\beta_1} t_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_{j-1}} t_{j-1}$$

from $t = t_0$ to some state t_{j-1} such that $s_i \xrightarrow{\alpha} t_i$ is a stutter step for $i = 1, \ldots, j-1$. Since $\alpha = \beta_j$ we have

$$t_{j-1} = \alpha(s_{j-1}) = \beta_j(s_{j-1}) = s_j.$$

Thus, by adding the forming path $s_j \xrightarrow{\beta_{j+1}} s_{j+1} \xrightarrow{\beta_{j+2}} \ldots \xrightarrow{\beta_n} s_n$ from s_j to $s_n = s'$ a forming path

$$t = t_0 \xrightarrow{\beta_1} t_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_{j-1}} t_{j-1} = s_j \xrightarrow{\beta_{j+1}} s_{j+1} \xrightarrow{\beta_{j+2}} \ldots \xrightarrow{\beta_n} s_n = s'$$

from t to s' with the action sequence $\beta_1 \ldots \beta_{j-1}\, \beta_{j+1} \ldots \beta_n$ is obtained. ■

Lemma 8.46. Properties of Forming Paths (Part 2)

Let s, s' be two states in TS such that $s \lhd s'$ and let $\alpha \in Act(s)$.

(a) If there is a forming path from s to s' in which α does not occur, then $\alpha \in Act(s')$ and $\alpha(s) \lhd \alpha(s')$.

(b) If α is a stutter action with $s \xrightarrow{\alpha} t$ and $\neg(t \lhd s')$, then $\alpha \in Act(s')$ and $s' \xrightarrow{\alpha} t'$ where $t \lhd t'$.

Proof: The proof for (a) can be provided using induction on the length n of a forming path from s to s' where α does not occur. The basis of induction $n = 0$ is obvious as we then have $s = s'$. In the induction step $n-1 \Longrightarrow n (n \geqslant 1)$ we assume that

$$s = s_0 \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_{n-1}} s_{n-1} \xrightarrow{\beta_n} s_n = s'$$

is a forming path from s to s' such that $\alpha \notin \{\beta_1, \ldots, \beta_n\}$. By induction hypothesis we have $\alpha \in \mathit{Act}(s_{n-1})$ and

$$\alpha(s) \lhd t$$

where $t = \alpha(s_{n-1})$. As the dependence condition (A2) holds for state s_{n-1} and the singleton action set $\{\beta_n\}$, actions α and β_n are independent. Thus, $\alpha \in \mathit{Act}(s_n)$ and $\beta_n \in \mathit{Act}(t)$ and, with $u = \beta_n(t)$,

$$\alpha(s') = \alpha(s_n) = \alpha(\beta_n(s_{n-1})) = \beta_n(\alpha(s_{n-1})) = \beta_n(t) = u.$$

Condition (A2) also holds for $\alpha(s) = t$ and the singleton action set $\{\beta_n\}$, since α and β_n are independent and (A2) holds for state s_{n-1} and the singleton action set $\{\beta_n\}$. We get

$$t \lhd u.$$

Hence, $\alpha(s) \lhd t \lhd u = \alpha(s')$ which yields $\alpha(s) \lhd \alpha(s')$.

The proof for part (b) can be provided with similar arguments, also using induction of the length of a forming path from s to s'. See Exercise 8.15. ■

Note that part (a) of Lemma 8.46 applies to all actions $\alpha \in \mathit{Act}(s)$ which are nonstutter actions, since they cannot occur on a forming path. In addition, there might also be stutter actions α enabled in s that do not occur on at least one forming path from s to the given state s'.

Notation 8.47. Relation $\mathcal{R}_{fp}$

The relation $\mathcal{R}_{fp}$ is given by $\mathcal{R}_{fp} = \{(s, \hat{s}) \in S \times \hat{S} \mid s \lhd \hat{s}\}$ where, as before, S is the state space of TS and $\hat{S}$ the state space of $\hat{TS}$. ■

Notation 8.48. Forming Path in $\hat{TS}$

A forming path in $\hat{TS}$ means a forming path $\hat{s}_0 \overset{\beta_1}{\Longrightarrow} \hat{s}_1 \overset{\beta_2}{\Longrightarrow} \ldots \overset{\beta_n}{\Longrightarrow} \hat{s}_n$ as in Definition 8.44 consisting of transitions in $\hat{TS}$. (Thus, $\hat{s}_0, \hat{s}_1, \ldots, \hat{s}_n \in \hat{S}$ and $\beta_{i+1} \in \mathit{ample}(\hat{s}_i)$ for $0 \leqslant i < n$.) ■

Lemma 8.49. Properties of Forming Paths in $\hat{TS}$

Let $\hat{s}$ be a state in $\hat{TS}$.

(a) If $\hat{\varrho}$ is a forming path in $\hat{TS}$ starting in state $\hat{s}$ and $(s, \hat{s}) \in \mathcal{R}_{fp}$, then $(s, \hat{u}) \in \mathcal{R}_{fp}$ for all states $\hat{u}$ in $\hat{\varrho}$.

(b) There exists a forming path from $\hat{s}$ in $\hat{TS}$ to some fully expanded state.

Proof: Part (a) is immediate by the transitivity of $\lhd$. The statement in (b) follows from the fact that any execution fragment

$$\hat{s}_0 \xRightarrow{\beta_1} \hat{s}_1 \xRightarrow{\beta_2} \ldots \xRightarrow{\beta_n} \hat{s}_n$$

in $\hat{TS}$ where none of the states $\hat{s}_i$ is fully expanded is a forming path in $\hat{TS}$, because of the dependency condition (A2), the stutter condition (A3), and the branching condition (A5). As $\hat{TS}$ is finite, the nonemptiness condition (A1) and the cycle condition (A4) ensure the existence of a forming path from $\hat{s}$ to a fully expanded state. ■

Notation 8.50. Length of Shortest Forming Paths

For $s, s' \in S$ with $s \lhd s'$ the length of a shortest forming path from s to s' is denoted $|s \lhd s'|$. For $\hat{s} \in \hat{S}$, we define $dist(\hat{s})$ as the length of a shortest forming path in $\hat{TS}$ from $\hat{s}$ to some fully expanded state. ■

Note that $dist(\hat{s})$ refers to the forming paths in $\hat{TS}$. Thus, $dist(\hat{s})$ might be larger than $\min_{\hat{v}} |\hat{s} \lhd \hat{v}|$ where $\hat{v}$ ranges over all fully expanded states in $\hat{TS}$.

Notation 8.51. Norm Functions ν_1, ν_2

The functions ν_1, $\nu_2 : S \times \hat{S} \to \mathbb{N}$ are defined as follows. For all $s \in S$ and $\hat{s} \in \hat{S}$ we put

$$\nu_1(s, \hat{s}) = \begin{cases} |s \lhd \hat{s}| & : \text{ if } (s, \hat{s}) \in \mathcal{R}_{fp} \\ 0 & : \text{ otherwise.} \end{cases}$$

and $\nu_2(s, \hat{s}) = dist(\hat{s})$. ■

Note that part (b) of Lemma 8.49 ensures that every state $\hat{s} \in \hat{S}$ has a forming path to some fully expanded state. Thus, ν_2 is well-defined.

***Lemma 8.52. Normed Bisimulation Equivalence of** TS **and** $\hat{TS}$*

$(\mathcal{R}_{fp}, \nu_1, \nu_2)$ is a normed bisimulation for $(TS, \hat{TS})$.

Proof: We have $(s_0, s_0) \in \mathcal{R}_{fp}$ for all initial states $s_0 \in I$, since we may consider a forming path of length 0.

We show that for any pair $(s, \hat{s}) \in \mathcal{R}_{fp}$ conditions (NI) and (NII) hold for $(s, \hat{s})$ and for $(\hat{s}, s)$ when ν_1 and ν_2 and their arguments are exchanged. The labeling condition (NI) is obvious as all actions on a forming path are stutter actions. Thus, all states on a forming path have the same labeling.

We now consider a pair $(s, \hat{s}) \in \mathcal{R}_{fp}$ and an action $\alpha \in \mathit{Act}(s)$.

Case 1. α does not occur on some forming path from s to $\hat{s}$.

Then, $\alpha \in \mathit{Act}(\hat{s})$ and $\alpha(s) \lhd \alpha(\hat{s})$ by part (a) of Lemma 8.46. Hence, if $\alpha \in \mathit{ample}(\hat{s})$, then case (N1) applies. If $\alpha \notin \mathit{ample}(\hat{s})$, then we choose the first action β of a shortest forming path in $\hat{TS}$ from $\hat{s}$ to some fully expanded state (see part (b) of Lemma 8.49 on page 658). Then,

$$\hat{s} \stackrel{\beta}{\Longrightarrow} \beta(\hat{s})$$

is a forming path in $\hat{TS}$ of length 1. Moreover, we have $(s, \beta(\hat{s})) \in \mathcal{R}_{fp}$ and

$$\nu_2(s, \beta(\hat{s})) = \mathit{dist}(\beta(\hat{s})) = \mathit{dist}(\hat{s}) - 1 = \nu_2(s, \hat{s}) - 1 < \nu_2(s, \hat{s}).$$

Hence, case (N3) applies.

Case 2. α occurs in some shortest forming path from s to $\hat{s}$.

Since α occurs is some forming path, α is a stutter action. Let

$$s_0 \xrightarrow{\beta_1} s_1 \ldots \xrightarrow{\beta_{j-1}} s_{j-1} \xrightarrow{\alpha} s_j \xrightarrow{\beta_{j+1}} \ldots \xrightarrow{\beta_n} s_n$$

be a shortest forming path from $s = s_0$ to $\hat{s} = s_n$ with $\alpha \notin \{\beta_1, \ldots, \beta_{j-1}\}$. By part (b) of Lemma 8.45 on page 655 there exists a forming path from $t = \alpha(s)$ to $\hat{s}$ with the action sequence $\beta_1 \ldots \beta_{j-1}\, \beta_{j+1} \ldots \beta_n$. Thus, $(t, \hat{s}) \in \mathcal{R}_{fp}$ and

$$\nu_1(t, \hat{s}) = |t \lhd \hat{s}| \leqslant n - 1 < n = |s \lhd \hat{s}| = \nu_1(s, \hat{s}).$$

Hence, case (N2) applies for the transition $s \xrightarrow{\alpha} t$.

It remains to show that (NII) holds for $(\hat{s}, s) \in \mathcal{R}^{-1}$ with exchanged roles of ν_1 and ν_2. Let $\alpha \in \mathit{ample}(s)$. If $s = \hat{s}$, then case (N1) applies. If $s \neq \hat{s}$ then we take the first action

β of a shortest forming path from s to $\hat{s}$. We then have $(\hat{s}, \beta(s)) \in \mathcal{R}_{fp}^{-1}$ and

$$\nu_1(\beta(s), \hat{s}) = |\beta(s) \lhd \hat{s}| = |s \lhd \hat{s}| - 1 = \nu_1(s, \hat{s}) - 1 < \nu_1(s, \hat{s}).$$

This yields condition (N3). ■

Example 8.53. Normed Bisimulation Equivalence of TS and $\hat{TS}$

We regard again the transition system TS in Figure 8.19 (page 650) and the ample sets $ample(s_0) = \{\alpha\}$, while all other states are fully expanded. This yields the reduced system shown in Figure 8.20 (page 651). We saw before that conditions (A1)-(A5) are fulfilled. The goal is now to provide a normed bisimulation for $(TS, \hat{TS})$. Besides the trivial forming paths of length 0, we have forming paths $s_0 \stackrel{\alpha}{\Longrightarrow} s_1$ in $\hat{TS}$, and forming paths $s_\beta \xrightarrow{\alpha} t_\beta$ and $s_\gamma \xrightarrow{\alpha} t_\gamma$ in TS. Thus, according to Notation 8.47 we deal with the relation

$$\mathcal{R}_{fp} = \mathrm{id} \cup \{(s_0, s_1), (s_\gamma, t_\gamma), (s_\beta, t_\beta)\}$$

where id denotes the set of all pairs $(\hat{s}, \hat{s})$ with $\hat{s}$ a state in $\hat{TS}$. The norm functions ν_1 and ν_2 are defined as follows. For the pairs $(\hat{s}, \hat{s}) \in \mathrm{id}$ we have $\nu_1(\hat{s}, \hat{s}) = |\hat{s} \lhd \hat{s}| = 0$. Moreover,

$$\nu_1(s_\gamma, t_\gamma) = \nu_1(s_\beta, t_\beta) = \nu_1(s_0, s_1) = 1$$

and

$$\nu_2(s_0, s_1) = \nu_2(s_0, s_0) = dist(s_0) = 1,$$

while $\nu_i(\cdot) = 0$ in all remaining cases. Let us check that $(\mathcal{R}_{fp}, \nu_1, \nu_2)$ is a normed bisimulation. Condition (NI) is obviously fulfilled. Moreover, (NII) is clear for the pairs $(\hat{s}, \hat{s}) \in \mathcal{R}_{fp}$. Let us regard the pair $(s_0, s_1) \in \mathcal{R}_{fp}$. For the transitions $s_\gamma \xrightarrow{\gamma} t_\gamma$ in TS, we take the transition $s_0 \stackrel{\alpha}{\Longrightarrow} s_1$ in $\hat{TS}$ where (N3) applies as we have $\nu_2(s_0, s_1) = 0 < 1 = \nu_2(s_0, s_0)$. An analogy holds for $s_\beta \xrightarrow{\beta} t_\beta$. For the transition $s_0 \xrightarrow{\alpha} s_1$ in TS the conditions of (N1) are fulfilled since $s_0 \stackrel{\alpha}{\Longrightarrow} s_1$ in $\hat{TS}$ and $(s_1, s_1) \in \mathcal{R}_{fp}$. Similarly, the transition $s_0 \stackrel{\alpha}{\Longrightarrow} s_1$ in $\hat{TS}$ is matched by $s_0 \xrightarrow{\alpha} s_1$ in TS according to (N1). For the others pairs in $\mathcal{R}_{fp}$, condition (NII) can be checked with similar arguments. ■

Lemma 8.52, together with Lemma 7.123 on page 555, yields that TS and $\hat{TS}$ are equivalent under stutter bisimulation equivalence with divergence. This completes the proof for Theorem 8.43.

Clearly, the DFS-based technique explained in Sections 8.2.2 and 8.2.3 to generate the reduced system is also applicable to ensure conditions (A1)-(A5). The additional requirement (A5) is local and simply amounts checking whether the candidate action set $Act_i(s)$ for $ample(s)$ of the chosen process $\mathcal{P}_i$ is a singleton set. The static partial order approach is also applicable, but requires extending the obtained ample sets for all states where $|ample(s)| \geqslant 2$.

8.4 Summary

- Partial order reduction attempts to analyze only a fragment $\hat{TS}$ of the full transition system TS by ignoring several interleavings of independent actions.
- Swapping independent actions in an execution fragment yields an execution fragment that starts and ends in the same state.
- The ample set method relies on choosing $\mathit{ample}(s) \subseteq \mathit{Act}(s)$ in state s. If the ample sets satisfy two local conditions (the nonemptiness condition (A1) and the stutter condition (A3)), a global condition for TS (the dependency condition (A2)), and for $\hat{TS}$ (the cycle condition (A4)), then TS and $\hat{TS}$ are stutter trace equivalent, and thus, $\mathrm{LTL}_{\setminus\bigcirc}$-equivalent.
- Conditions (A1) and (A2) ensure that any execution ρ in TS can be turned into an execution ρ' in $\hat{TS}$ by successive permutations of independent actions and adding independent stutter actions. The stutter and the cycle conditions (A3) and (A4) ensure that ρ is stutter-equivalent to ρ'.
- In on-the-fly partial order reduction, the reduced transition system $\hat{TS}$ is generated during (nested) depth-first search. To ensure the cycle condition (A4), states that yield a backward edge to a state on the stack are fully expanded.
- Checking the dependency condition (A2) has the same worst case time complexity as solving a reachability problem. Instead of ensuring (A2), some stronger conditions are imposed that can easily be checked by a static analysis and are stronger than (A2).
- Static partial order reduction generates the reduced transition system prior to the verification. To establish the cycle condition (A4), the static approach relies on determining a set of sticky actions. If the set of sticky actions fulfills the visibility condition (S1) and the cycle-breaking condition (S2), the reduction yields a stutter-equivalent transition system.
- If the ample sets satisfy conditions (A1) through (A4) and the branching condition (A5), then TS and $\hat{TS}$ are divergent-sensitive stutter-bisimilar, and thus, $\mathrm{CTL}^*_{\setminus\bigcirc}$-equivalent.

8.5 Bibliographic Notes

Independence of actions. Partial order reduction has been inspired by earlier work on the commutativity of concurrent activities by Lipton [276] and Mazurkiewicz [287]. They

considered action sequences that are equivalent up to permuting independent actions. Equivalence classes of action sequences are called Mazurkiewicz traces. (Exercise 8.5 deals with a variant of Mazurkiewcz traces as in [253, 357, 125].) Other partial order models for concurrent systems that have been influential on partial order reduction are, among others, pomsets [343], partial orders [258], branching processes of Petri nets [144], and event structures [423]. The notion of independent actions (see Definition 8.3) has been introduced by Katz and Peled [234]. Deductive proof techniques for concurrent systems that are based on a partial order view have been proposed in the early eigthies by Apt, Francez, and de Roever [16], and Elrad and Francez [137]. These techniques have been extended and refined by several others; see e.g., [339, 225, 235, 380].

Partial order reduction. The concepts of partial order reduction for the algorithmic verification of concurrent asynchronous systems have been developed independently by Godefroid, Peled, and Valmari in the beginning of the nineties. This chapter is based on the ample set approach by Peled for $\mathrm{LTL}_{\setminus\bigcirc}$ [324, 211, 325]. Ample sets are similar to Godefroid's persistent sets [168, 171, 169] and Valmari's stubborn sets [398, 399, 400]. The ample set approach for branching-time properties presented in Section 8.3 originates from Gerth, Kuiper, Peled, and Penczek [165, 326]. Alternative partial order reduction approaches for branching-time properties have been provided by Willems and Wolper [422], and Ramakrishna and Smolka [352]. Alternative criteria for preserving linear and branching-time properties as well as sound criteria for the universal fragment of $\mathrm{CTL}^*_{\setminus\bigcirc}$ have been presented by Penczek, Gerth, Kuiper, and Szreter [330] (see also Exercises 8.16–8.18). Partial order reduction techniques for equivalence checking under various implementation relations have been discussed, e.g., in [401, 218].

Computing the reduced model. A detailed discussion of nested depth-first search in combination with partial order reduction is given by Holzmann, Peled, and Yannakakis [212, 325]. For further reading on partial order reduction and its implementation in the model checker SPIN we refer to [169], [92, Chapter 10], and the monograph by Holzmann on SPIN [209]. In the last years, several variants have been discussed, see e.g. [169, 402, 310, 413, 135, 64, 328]. Static partial order reduction (see Section 8.2.4) has been developed by Kurshan, Levin, Minea, Peled and Yenigün [251]. Other applications of partial order reduction include: integration with symbolic techniques (by Abdulla, Jonsson, Kindahl and Peled [3]), infinite-state systems (by Alur, Brayton, Henzinger, Qadeer and Rajamani [8]), and breadth-first search strategies (by Bosnacki and Holzmann [59]).

Related approaches. The concept of *sleep sets* [168, 172, 169] is orthogonal to the ample, persistent, or stubborn sets. Sleep sets are aimed at decreasing the number of transitions rather than the number of states and are based on the (dynamic) knowledge of the graph structure that is obtained during the depth-first search traversal of the (possibly reduced) transition system. The concept of τ-confluence, proposed by Groote and van de Pol [177],

is aimed at symbolic state-space reduction and obtains branching bisimilar transition systems. Rather than perform a partial order reduction of an interleaving representation of a concurrent system, one can attempt to directly perform model checking on partial orders. McMillan [289] has presented an algorithm to obtain an initial part of the (infinite) branching process of a 1-safe Petri net. The so-called complete finite prefix of the branching process contains all information on reachable states and transitions, and can be used as the basis for model-checking algorithms, see, e.g., [145, 416].

8.6 Exercises

EXERCISE 8.1. Let TS be the transition system depicted in Figure8.23 with the action set $Act = \{ \alpha, \beta, \gamma, \delta, \tau \}$. Determine the pairs of independent actions in TS.

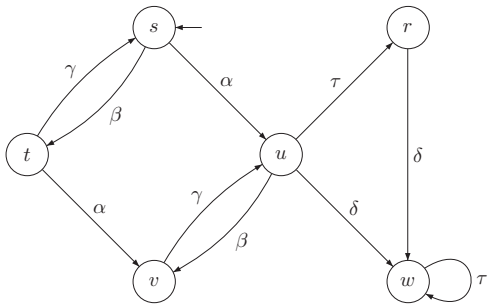


Figure 8.23: Transition system TS for Exercise 8.1.

EXERCISE 8.2. Let TS be a transition system and let $\mathcal{I}$ be the set of action pairs (α, β) such that α and β are independent of TS. Is $\mathcal{I}$ transitive? reflexive? symmetric?

EXERCISE 8.3. Consider the transition system $TS(PG_1 \,|||\, PG_2)$ for the program graphs PG_1 (left) and PG_2 (right) in Figure 8.24. Determine the pairs of independent actions.

EXERCISE 8.4. Consider the transition system TS_{Sem} for mutual exclusion with a semaphore (see Figure 2.8 on page 45). We deal with the action set $Act = \{request_i, rel_i, enter_i : i = 1, 2\}$ and the proposition set $AP = \{crit_1, crit_2\}$.

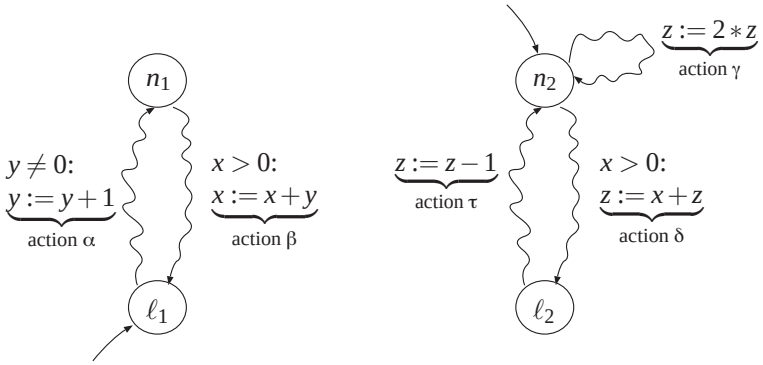


Figure 8.24: Program graphs PG_1 (left) and PG_2 (right) for Exercise 8.3.

(a) Consider the finite execution fragment ϱ

$$\begin{array}{ll}
\langle noncrit_1, noncrit_2, y=1\rangle & \xrightarrow{request_1} \\
\langle wait_1, noncrit_2, y=1\rangle & \xrightarrow{enter_1} \\
\langle crit_1, noncrit_2, y=0\rangle & \xrightarrow{rel_1} \\
\langle noncrit_1, noncrit_2, y=1\rangle & \xrightarrow{request_2} \\
\langle noncrit_1, wait_2, y=1\rangle &
\end{array}$$

Verify the statement of Lemma 8.10 (page 604) by investigating the three action sequences

$$\begin{array}{l}
request_1\ enter_1\ request_2\ rel_1, \\
request_1\ request_2\ enter_1\ rel_1, \\
request_2\ request_1\ enter_1\ rel_1,
\end{array}$$

and checking that they lead from $s = \langle noncrit_1, noncrit_2, y=1\rangle$ to state $t = \langle noncrit_1, wait_2, y=1\rangle$ and are stutter-equivalent to ϱ.

(b) Consider the infinite execution ρ

$$\begin{array}{lll}
\langle noncrit_1, noncrit_2, y=1\rangle & \xrightarrow{request_1} & \\
\langle wait_1, noncrit_2, y=1\rangle & \xrightarrow{enter_1} & \\
\langle crit_1, noncrit_2, y=0\rangle & \xrightarrow{rel_1} & \\
\langle noncrit_1, noncrit_2, y=1\rangle & \xrightarrow{request_1} & \ldots
\end{array}$$

where process $\mathcal{P}_1$ continuously passes through its three phases, while $\mathcal{P}_2$ does nothing. Verify Lemma 8.11 (page 605) by investigating the action sequence $request_2\,(request_1\ enter_1\ rel_1)^\omega$ and checking that it yields an infinite execution that is stuttering equivalent to ρ.

Exercise 8.5. Let $TS = (S, Act, \rightarrow, I, AP, L)$ be an action-deterministic transition system. and let $\mathcal{I}_{st}$ be the set of all pairs $(\alpha, \beta) \in Act \times Act$ of independent actions α and β where α or β (or

both) is a stutter action. Let stutter permutation equivalence $\leftrightharpoons_{perm}$ be the coarsest equivalence on Act^* such that

$$\overline{\gamma}\,\alpha\,\beta\,\overline{\delta} \;\leftrightharpoons_{perm}\; \overline{\gamma}\,\beta\,\alpha\,\overline{\delta}$$

if $\overline{\gamma}$, $\overline{\delta} \in Act^*$ and $(\alpha, \beta) \in \mathcal{I}_{st}$.

(a) Let ϱ and ϱ' be two finite execution fragments in TS that start in the same state and that rely on stutter permutation equivalent action sequences. Show that $\varrho \triangleq \varrho'$.

(b) Let $\varrho = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_k} s_k$ be a finite execution fragment in TS and let $\beta_1 \ldots \beta_k \in Act^*$ such that $\alpha_1 \ldots \alpha_k \leftrightharpoons_{perm} \beta_1 \ldots \beta_k$. Show that there exists a a finite execution fragment in TS which starts in s_0 and relies on the action sequence $\beta_1 \ldots \beta_k$.

The extension of $\leftrightharpoons_{perm}$ to an equivalence for infinite action sequences is defined as follows. If $\widetilde{\alpha} = \alpha_1\,\alpha_2\,\alpha_3 \ldots$ and $\widetilde{\beta} = \beta_1\,\beta_2\,\beta_3 \ldots$ are action sequences in Act^ω then $\widetilde{\alpha} \trianglelefteq_{perm} \widetilde{\beta}$ if for all finite prefixes $\alpha_1 \ldots \alpha_n$ of $\widetilde{\alpha}$ there exists a finite prefix $\beta_1 \ldots \beta_m$ of $\widetilde{\beta}$ with $m \geqslant n$ and a finite word $\overline{\gamma} \in Act^*$ such that

$$\alpha_1 \ldots \alpha_n\,\overline{\gamma} \;\leftrightharpoons_{perm}\; \beta_1 \ldots \beta_m.$$

We then define the binary relation ${\leftrightharpoons_{perm}}^{\omega}$ on Act^ω by

$$\widetilde{\alpha}\;{\leftrightharpoons_{perm}}^{\omega}\;\widetilde{\beta} \quad \text{iff} \quad \widetilde{\alpha} \trianglelefteq_{perm} \widetilde{\beta} \text{ and } \widetilde{\beta} \trianglelefteq_{perm} \widetilde{\alpha}.$$

(a) Show that ${\leftrightharpoons_{perm}}^{\omega}$ is an equivalence.

(b) Let ρ and ρ' be two infinite execution fragments in TS starting in the same state s with the action sequences $\widetilde{\alpha}$ and $\widetilde{\beta}$, respectively. Show that if $\widetilde{\alpha}\;{\leftrightharpoons_{perm}}^{\omega}\;\widetilde{\beta}$, then $\rho \triangleq \rho'$.

(c) Let $\rho = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots$ be a infinite execution fragment in TS with the action sequence $\widetilde{\alpha} = \alpha_1\,\alpha_2 \ldots \in Act^\omega$ and let $\widetilde{\beta} = \beta_1\,\beta_2 \ldots \in Act^\omega$ such that $\widetilde{\alpha}\;{\leftrightharpoons_{perm}}^{\omega}\;\widetilde{\beta}$. Show that there exists an infinite execution fragment in TS which starts in s_0 and relies on the action sequence $\widetilde{\beta}$.

(d) Let $\widetilde{\alpha} = \alpha_1\,\alpha_2\,\alpha_3 \ldots \in Act^\omega$ and let $(\widetilde{\alpha}_i)_{i \geqslant 1}$ be a sequence of infinite action sequences of the form

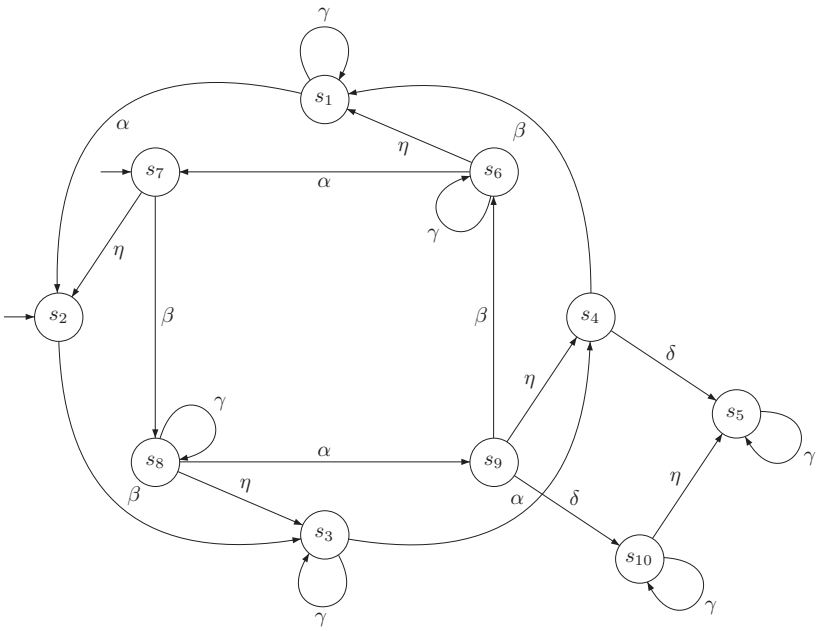
$$\widetilde{\alpha}_i = \underbrace{\alpha_1 \ldots \alpha_i}_{\text{as in } \widetilde{\alpha}}\,\widetilde{\beta}_i$$

where $\widetilde{\beta}_i$ is an element of Act^ω such that

$$\widetilde{\alpha}_1\;{\leftrightharpoons_{perm}}^{\omega}\;\widetilde{\alpha}_2\;{\leftrightharpoons_{perm}}^{\omega}\;\widetilde{\alpha}_3\;{\leftrightharpoons_{perm}}^{\omega}\;\ldots$$

Can we conclude that $\widetilde{\alpha}_i\;{\leftrightharpoons_{perm}}^{\omega}\;\widetilde{\alpha}$ for all $i \geqslant 1$? Provide a proof or a counterexample.

Exercise 8.6.



The state labeling is as follows:

- $L(s_{10}) = \varnothing$
- $L(s_6) = L(s_7) = \{\, a \,\}$
- $L(s_3) = L(s_4) = L(s_5) = L(s_8) = L(s_9) = \{\, b \,\}$
- $L(s_1) = L(s_2) = \{\, a, b \,\}$

Indicate for each of the following *ample sets* whether they satisfy the requirements (A1) through (A3). Also check whether the requirement (A4) holds

- $\mathit{ample}(s_6) = \{\, \gamma, \alpha \,\}$
- $\mathit{ample}(s_7) = \{\, \beta \,\}$
- $\mathit{ample}(s_8) = \{\, \alpha \,\}$
- $\mathit{ample}(s_9) = \{\, \alpha, \beta, \delta \,\}$
- $\mathit{ample}(s_{10}) = \{\, \gamma, \eta \,\}$

If the conditions (A1) through (A4) do not hold, provide a minimal extension of the *ample sets* to fix it. Clarify your adaptations.

EXERCISE 8.7. Consider the transition system TS_{Pet} for the Peterson mutual exclusion algorithm (see page 45).

(a) Which actions are independent?

(b) Apply the partial order reduction approach to TS_{Pet} with small ample sets according to

(i) Algorithm 38 (page 622) for checking the invariant $\Box\neg(crit_1 \wedge crit_2)$. Take $AP = \{ crit_1, crit_2 \}$.

(ii) Algorithm 39 (page 625) for checking the liveness property $\Box\Diamond crit_1$. Take $AP = \{ crit_1 \}$.

EXERCISE 8.8. Consider the transition system $TS = BCR \| BP \| Printer$ for the booking system considered in Example 2.29 on page 50.

(a) Which actions are independent?

(b) Apply the partial order reduction approach to TS with minimal ample sets according to Algorithm 38 (page 622) for checking the liveness property $\varphi = \Box\Diamond$"printer in location 1" where the printer's locations serve as atomic propositions.

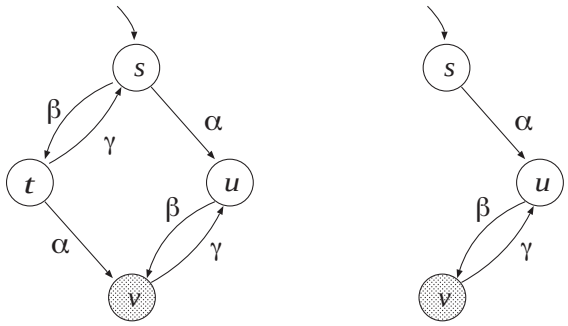


Figure 8.25: Transition system TS (left) and $\hat{TS}$ (right) for Exercise 8.9.

EXERCISE 8.9. Figure 8.25 shows on its left a transition system TS and on its right a reduced system $\hat{TS}$ that results from choosing $ample(s) = \{ \alpha \}$. Check whether TS and $\hat{TS}$ are stutter trace equivalent. If they are not, indicate which of the conditions (A1)-(A4) is (are) violated.

Answer the same question for the transition system in the reduction shown in Figures 8.26 and 8.27, where different colors indicate different state labels.

EXERCISE 8.10. Consider the transition system TS depicted below. Show that conditions (A1)-(A4) do not allow for any state reduction, although there is a smaller subsystem $\hat{TS}$ that is stutter trace equivalent to TS.

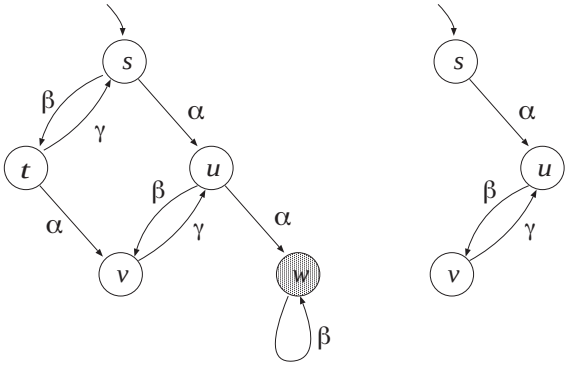


Figure 8.26: Transition system TS (left) and $\hat{TS}$ (right) for Exercise 8.9.

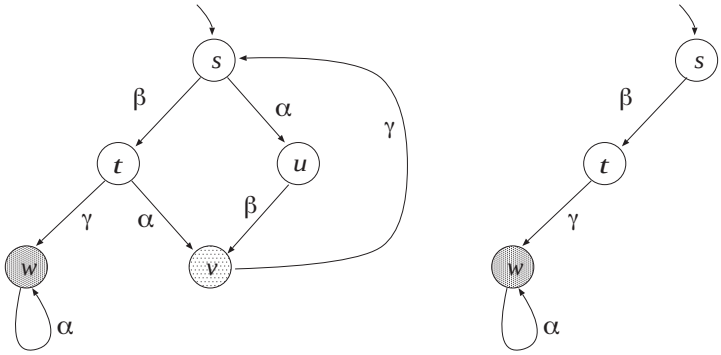
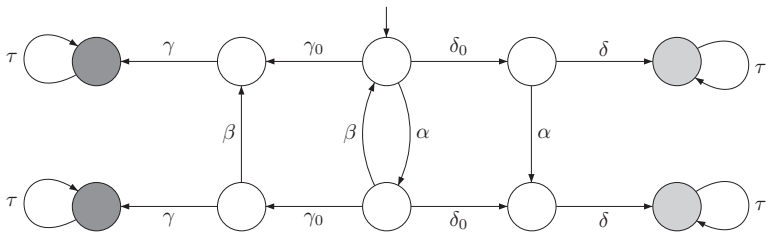


Figure 8.27: Transition system TS (left) and $\hat{TS}$ (right) for Exercise 8.9.



EXERCISE 8.11. Let $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP, L_i)$, $i = 1, \ldots, n$, be action-deterministic transition systems such that $Act_i \cap Act_j \cap Act_k = \varnothing$ if $1 \leqslant i < j < k \leqslant n$. We consider the parallel composition with synchronization over common actions (see page 49), i.e., the transition system

$$TS = TS_1 \| TS_2 \| \ldots \| TS_n.$$

For each state $s = \langle s_1, \ldots, s_n \rangle$ of TS, let $Act_i(s) = Act_i \cap Act(s)$ be the set of actions of TS_i that

are enabled in s.

Show that the dependency condition (A2) holds if for each state s of TS the following conditions (i) and (ii) hold:

(i) If $ample(s) \neq Act(s)$, then $ample(s) = Act_i(s)$ for some $i \in \{1, \ldots, n\}$.

(ii) If $ample(s) = Act_i(s) \neq Act(s)$, then $ample(s) \cap \big(\bigcup_{\substack{1 \leqslant j \leqslant n \\ j \neq i}} Act_j \big) = \varnothing$.

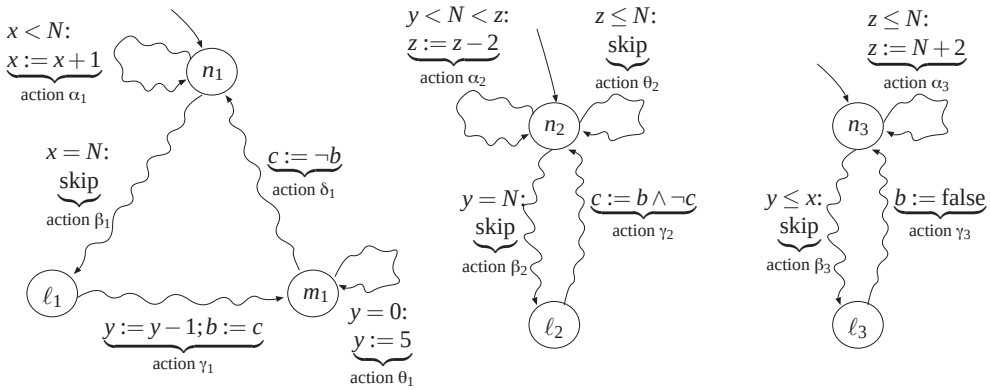


Figure 8.28: Program graphs PG_1, PG_2, PG_3 for Exercise 8.12.

Exercise 8.12. Apply the static partial order reduction approach to generate the action set A_{sticky} by means of Algorithm 42 (page 648) and the modified program graphs $\hat{PG}_1$, $\hat{PG}_2$, and $\hat{PG}_3$ for the three program graphs shown in Figure 8.28.

Exercise 8.13. Explain which modifications of the presented static partial order reduction approach are necessary to treat program graphs with channel-based message passing.

Exercise 8.14. Consider the transition system TS shown on the left of Figure 8.29 where the three black states are labeled by $\{a\}$, while the white states are labeled by $\varnothing$. Let $ample(\cdot)$ be the ample sets that yield the reduced system $\hat{TS}$ shown on the right of Figure 8.29.

(a) Show that (A1)-(A5) are fulfilled.

(b) Provide a normed bisimulation for $(TS, \hat{TS})$.

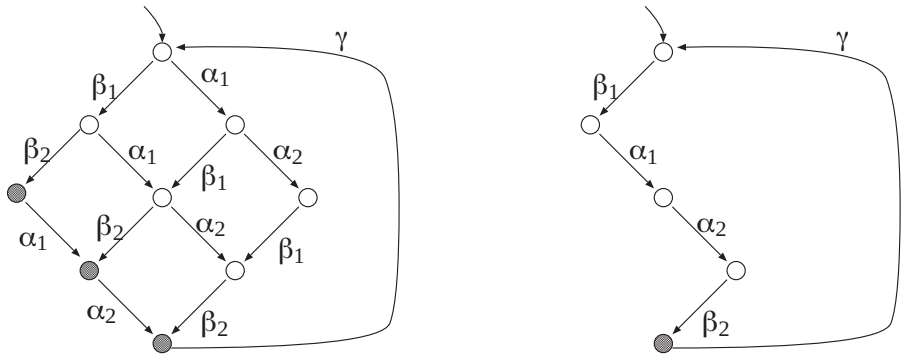


Figure 8.29: Transition system TS (left) and $\hat{TS}$ (right) for Exercise 8.14.

Exercise 8.15. Provide the proof for part (b) of Lemma 8.46 on page 656.

Exercise 8.16. Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a finite, action-deterministic transition system without terminal states and let $\mathcal{I}$ be a binary relation on $Act \times Act$ such that $(\alpha, \beta) \notin \mathcal{I}$ if α and β are dependent actions. Let Vis be the set of visible actions, i.e., Vis is the set of all actions $\alpha \in Act$ that are not stutter actions. Furthermore, let ample sets $ample(s) \subseteq Act(s)$ be given such that the following conditions (A2$_\mathcal{I}$), (A6), (A7) and (A8) hold for all states $s \in S$:

(A2$_\mathcal{I}$) If $s \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} s_2 \xrightarrow{\beta_3} \ldots \xrightarrow{\beta_m} s_m \xrightarrow{\gamma} t$ is a finite execution fragment in TS such that $\gamma \notin ample(s)$ and $(\alpha, \gamma) \notin \mathcal{I}$ for all $\alpha \in ample(s)$. then there exists an index $n \in \{1, \ldots, m\}$ such that $\beta_n \in ample(s)$.

(A6) $(Vis \times Vis) \cap \mathcal{I} = \varnothing$.

(A7) If s is not fully expanded, then $ample(s)$ contains at least one visible action, i.e., $ample(s) \cap Vis \neq \varnothing$ and $ample(s) \setminus Vis \neq \varnothing$.

(A8) If s is not fully expanded, then $ample(s)$ contains at least one stutter action, i.e., $ample(s) \setminus Vis \neq \varnothing$.

Show that TS and $\hat{TS}$ are LTL$_{\setminus\bigcirc}$-equivalent.

Exercise 8.17. Let TS, $\mathcal{I}$, and conditions (A2$_\mathcal{I}$), (A6), (A7) be as in Exercise 8.16 and let ample sets be given such that (A2$_\mathcal{I}$), (A6), (A7) hold. As before, let $\hat{S}$ be the state space of $\hat{TS}$ and let

$$\mathcal{R} \;=\; \big\{(s,\hat{s}) \in S \times \hat{S} \;\mid\; \text{there exists an execution fragment } s \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} \hat{s} \text{ such that } \alpha_1, \ldots, \alpha_n \text{ are stutter actions} \big\}.$$

Show that $\mathcal{R}$ is a stutter simulation for $(TS, \hat{TS})$, defined as in Exercise 7.29 on page 592.

EXERCISE 8.18. Let TS, $\mathcal{I}$, and conditions ($A2_{\mathcal{I}}$), (A6), (A7) be as in Exercise 8.16. Show that if conditions ($A2_{\mathcal{I}}$), (A6), (A7), and the branching condition (A5) hold then TS and $\hat{TS}$ are $\forall CTL^*_{\setminus\bigcirc}$-equivalent.

(*Hint: Show that the relation $\mathcal{R}$ as in Exercise 8.17 is a divergence-sensitive stutter simulation as in Exercise 7.29 (page 592) and apply the statement (h) of Exercise 7.29 and Theorem 7.76 (page 517)*).

Chapter 9

Timed Automata

The logics we have encountered so far are interpreted over transition systems that describe how a reactive system may evolve from one state to another. Timing aspects are, however, not covered. That is, indications are given neither about the residence time of a state nor about the possibility of taking a transition within a particular time interval. However, reactive systems such as device drivers, coffee machines, communication protocols, and automatic teller machines, to mention a few, must react in time—they are *time-critical*. The behavior of time-critical systems is typically subject to rather stringent timing constraints. For a train crossing it is essential that on detecting the approach of a train, the gate is closed within a certain time bound in order to halt car and pedestrian traffic before the train reaches the crossing. For a radiation machine the time period during which a cancer patient is subjected to a high dose of radiation is extremely important; a small extension of this period is dangerous and can cause the patient's death.

To put it in a nutshell:

> *Correctness in time-critical systems not only depends on the logical result of the computation but also on the time at which the results are produced.*

As timeliness is of vital importance to reactive systems, it is essential that the timing constraints of the system are guaranteed to be met. Checking whether timing constraints are met is the subject of this chapter. In order to express such timing constraints, the strategy will be to extend logical formalisms that allow expression of the ordering of events, with a notion of quantitative time. Such extensions allow expression of timing constraints such as:

"The traffic light will turn green within the next 30 seconds."

A first choice to be made is the time domain: is it discrete or continuous? A discrete time domain is conceptually simple. Transition systems are used to model timed systems where each action is assumed to last for a single time unit. More general delays can be modeled by using a dedicated unobservable action, τ (for tick), say. The fact that action α lasts $k > 1$ time units may be modeled by $k-1$ tick actions followed (or preceded) by α. This approach typically leads to very large transition systems. Note that in such models the minimal time difference between any pair of actions is a multiple of an a priori, fixed, time unit. For synchronous systems, for instance, in which the involved processes proceed in a lockstep fashion, discrete time domains are appropriate: one time unit corresponds to one clock pulse. In this setting, traditional temporal logics can be used to express timing constraints. The next-step operator can be used to "measure" the discrete elapse of time, i.e., $\bigcirc \Phi$ means that Φ holds after exactly one time unit. By defining $\bigcirc^{k+1} \Phi = \bigcirc^{k} (\bigcirc \Phi)$ and $\bigcirc^{0} \Phi = \Phi$, general timing constraints can be specified. Using the shorthand $\Diamond^{\leqslant k} \Phi = \bigcirc^{0} \Phi \vee \bigcirc \Phi \vee \ldots \vee \bigcirc^{k} \Phi$, the above informally stated timing constraint on the traffic light may be expressed as

$$\Box (\mathit{red} \Rightarrow \Diamond^{\leqslant 30} \mathit{green}).$$

For synchronous systems, transition systems and logics such as LTL or CTL can be used to express timing constraints, and traditional model checking algorithms suffice.

In this monograph we do not want to restrict ourselves to synchronous systems, and will consider—as in Newtonian physics—time of a continuous nature. That is to say, the non-negative real numbers (the set $\mathbb{R}_{\geqslant 0}$) will be used as time domain. A main advantage is that there is no need to fix a minimal time unit in advance as a continuous time model is invariant against changes of the time scale. This is more adequate for asynchronous systems, such as distributed systems, in which components may proceed at distinct speeds, and is more an intuitive than a discrete time model. Transition system representations of asynchronous systems without additional timing information are indeed too abstract to adequately model timing constraints, as illustrated by the following example.

Example 9.1. A Railroad Crossing

Consider a railroad crossing, as discussed in Example 2.30 (page 51), see the schematic representation in Figure 9.1. For this railroad crossing a control system needs to be developed that closes the gate on receipt of a signal indicating that a train is approaching and only opens the gate once the train has signaled that it entirely crossed the road. The safety property that should be established by the control system is that the gates are always closed when the train is crossing the road. The complete system consists of the three components: *Train*, *Gate*, and *Controller*:

$$\mathit{Train} \parallel \mathit{Gate} \parallel \mathit{Controller}.$$

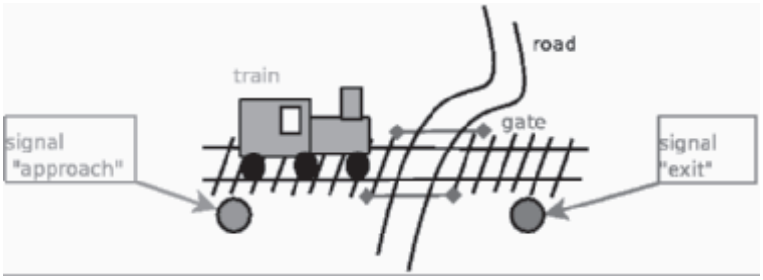


Figure 9.1: Railroad crossing (time abstract).

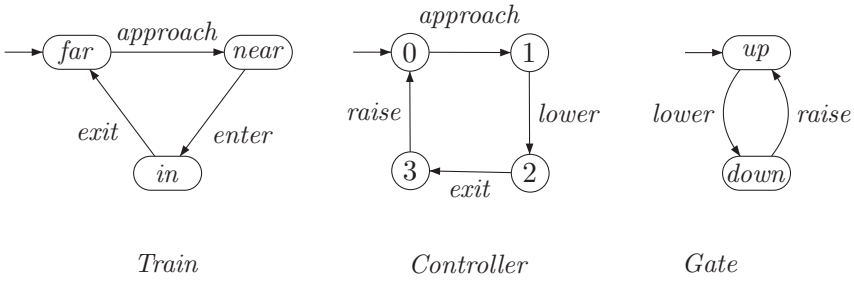


Figure 9.2: Transition systems for processes *Train* (left), *Controller* (middle), and *Gate* (right).

Recall that actions common to a pair of processes need to be performed jointly, while other actions are performed autonomously. The transition systems of these processes are depicted in Figure 9.2. It follows that the composite system $\mathit{Train} \parallel \mathit{Gate} \parallel \mathit{Controller}$ does not guarantee that the gate is closed when the train is passing the crossing. This can easily be seen by inspecting an initial fragment of the composite transition system; see Figure 9.3—it cannot be deduced from the transition system whether, after sending the "approach" signal, the train reaches the road before or after the gate has been closed.

Under the assumption, though, that the train does not exceed a certain maximum speed, a lower bound for the duration between the signal "approach" and the time instant at which the train has reached the crossing can be indicated, see Figure 9.4. Let us assume that the train needs more than 2 minutes to reach the crossing after emission of the "approach" signal. Accordingly, timing assumptions are made for the controller and the gate. On receiving the "approach" signal, after exactly 1 minute the controller will signal the gate to be lowered. The actual closing of the gate is assumed not to exceed a minute. The branching in global state $\langle \mathit{near}, 1, \mathit{up} \rangle$ can now be labeled with timing information:

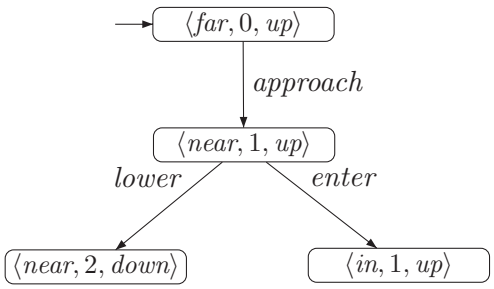


Figure 9.3: Initial fragment of the transition system *Train* ‖ *Controller* ‖ *Gate*.

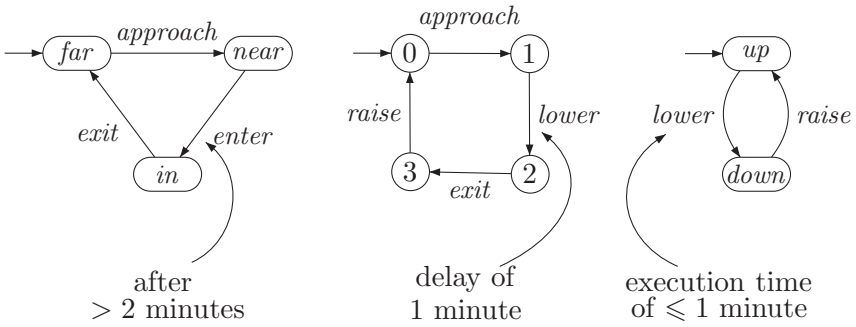
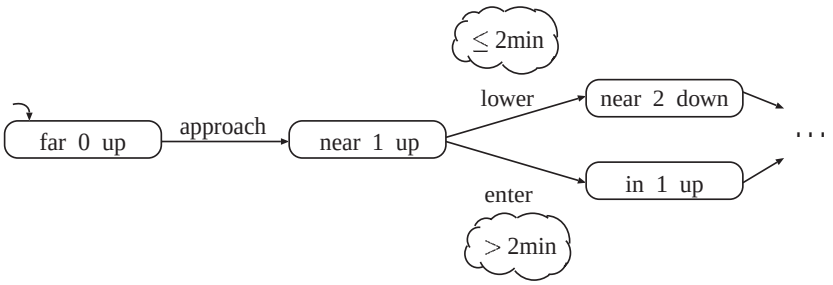


Figure 9.4: The *Train* (left), *Controller* (middle), and *Gate* (right) with timing assumptions.



The train can only execute the local state change $near \xrightarrow{enter} in$ after more than 2 minutes. On the other hand, the gate is closed at most 2 minutes after receiving the "approach" signal. Therefore, the global state change

$$\langle near, 1, up \rangle \xrightarrow{enter} \langle in, 1, up \rangle$$

never occurs. Thus, the gate is always closed before the train reaches the crossing. The fact that the gate remains closed as long as the train is on the crossing is ensured by the fact that action *raise* can only happen after the train has indicated *exit*. ■

As a modeling formalism for time-critical systems, the notion of *timed automata* has been developed, an extension of transition systems (in fact, program graphs) with clock variables that measure the elapse of time. This model includes means to impose constraints on the residence times of states, and on the timing of actions.

9.1 Timed Automata

Timed automata model the behavior of time-critical systems. A timed automaton is in fact a program graph that is equipped with a finite set of real-valued clock variables, called *clocks* for short. In the sequel we assume that the set of clocks is denumerable, and we will use x, y, and z as clocks. Clocks are different from the usual variables, as their access is limited: clocks may only be inspected, and reset to zero. Clocks can be reset to zero after which they start increasing their value implicitly as time progresses. All clocks proceed at rate one, i.e., after the elapse of d time units, all clocks advance by d. The value of a clock thus denotes the amount of time that has been elapsed since its last reset. Clocks can intuitively be considered as stopwatches that can be started and checked independently of one another. Conditions on the values of the clocks are used as enabling conditions (i.e., guards) of actions: only if the condition is fulfilled is the action enabled and capable of being taken; otherwise, the action is disabled. Conditions which depend on clock values are called *clock constraints*. For the sake of simplicity, it is assumed that enabling conditions only depend on clocks and not on other data variables. Clock

constrains are also used to limit the amount of time that may be spent in a location. The following definition prescribes how constraints over clocks are to be defined.

Definition 9.2. Clock Constraint

A *clock constraint* over set C of clocks is formed according to the grammar

$$g ::= \quad x < c \;\Big|\; x \leqslant c \;\Big|\; x > c \;\Big|\; x \geqslant c \;\Big|\; g \wedge g$$

where $c \in \mathbb{N}$ and $x \in C$. Let $CC(C)$ denote the set of clock constraints over C.

Clock constraints that do not contain any conjunctions are *atomic*. Let $ACC(C)$ denote the set of all atomic clock constraints over C.

■

Clock constraints are often written in abbreviated form, e.g., $(x \geqslant c_1) \wedge (x < c_2)$ may be abbreviated by $x \in [c_1, c_2)$ or $c_1 \leqslant x < c_2$. Clock difference constraints such as $x - y < c$ can be added at the expense of a slightly more involved theory. For simplicity, they are omitted here and we restrict the discussion to atomic clock constraints that compare a clock with a constant $c \in \mathbb{N}$. The decidability of the model-checking problem is not affected if c is allowed to be rational. In this case the rationals in each formula can be converted into natural numbers by suitable scaling. In general, we can multiply each constant by the least common multiple of denominators of all constants appearing in all clock constraints.

Intuitively, a timed automaton is a (slightly modified) program graph, whose variables are clocks. The clocks are used to formulate the real-time assumptions on system behavior. An edge in a timed automaton is labeled with a guard (when is it allowed to take an edge?), an action (what is performed when taking the edge?), and a set of clocks (which clocks are to be reset?). A location is equipped with an invariant that constrains the amount of time that may be spent in that location. The formal definition is:

Definition 9.3. Timed Automaton

A *timed automaton* is a tuple $TA = (Loc, Act, C, \hookrightarrow, Loc_0, Inv, AP, L)$ where

- Loc is a finite set of locations, ,
- $Loc_0 \subseteq Loc$ is a set of initial locations,
- Act is a finite set of actions,
- C is a finite set of clocks,

- $\hookrightarrow \subseteq Loc \times CC(C) \times Act \times 2^C \times Loc$ is a transition relation,
- $Inv : Loc \to CC(C)$ is an invariant-assignment function,
- AP is a finite set of atomic propositions, and
- $L : Loc \to 2^{AP}$ is a labeling function for the locations.

$ACC(TA)$ denotes the set of atomic clock constraints that occur in either a guard or a location invariant of TA. ■

A timed automaton is a program graph with a finite set C of clocks. Edges are labeled with tuples (g, α, D) where g is a clock constraint on the clocks of the timed automaton, α is an action, and $D \subseteq C$ is a set of clocks. The intuitive interpretation of $\ell \overset{g:\alpha,D}{\hookrightarrow} \ell'$ is that the timed automaton can move from location ℓ to location ℓ' when clock constraint g holds. Besides, when moving from location ℓ to ℓ', any clock in D will be reset to zero and action α is performed. Function Inv assigns to each location a location invariant that specifies how long the timed automaton may stay there. For location l, $Inv(\ell)$ constrains the amount of time that may be spent in ℓ. That is to say, the location ℓ should be left before the invariant $Inv(\ell)$ becomes invalid. If this is not possible—as there is no outgoing transition enabled—no further progress is possible. In the formal semantics of timed automata (see Definition 9.11) this situation causes time progress to halt. As time progress is no longer possible, this situation is also known as a *timelock*. This phenomenon will be discussed in more detail later. The function L has the same role as for transition systems and associates to any location the set of atomic propositions that are valid in that location.

Before considering the precise interpretation of timed automata, we give some simple examples.

For depicting timed automata we adopt the drawing conventions for program graphs. Invariants are indicated inside locations and are omitted when they equal true. Edges are labeled with the guard, the action, and the set of clocks to be reset. Empty sets of clocks are often omitted. The same applies to clock constraints that are constantly true. The resetting of set D of clocks is sometimes indicated by $reset(D)$. If the actions are irrelevant, they are omitted.

Example 9.4. Guards vs. Location Invariants

Figure 9.5(a) depicts a simple timed automaton with one clock x and one location ℓ equipped with a self-loop. The self-loop can be taken if clock x has at least the value 2, and when being taken, clock x is reset. Initially, by default clock x has the value 0.

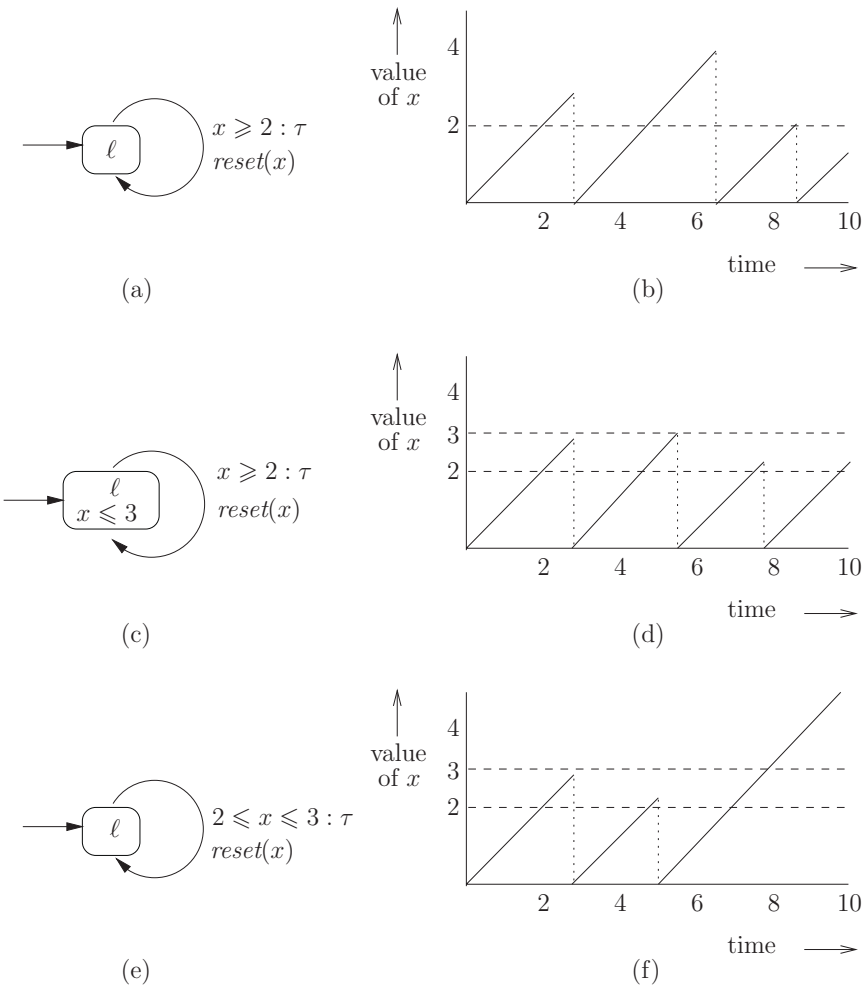


Figure 9.5: Some timed automata with a single clock and one of their evolutions.

Figure 9.5(b) gives an example of an execution of this timed automaton, by depicting the value of clock x vs. the elapsed time since the start of the automaton. Each time the clock is reset to 0, the automaton traverses the self-loop at location ℓ. As $Inv(\ell) = \text{true}$, time can progress without any restriction while residing in ℓ. In particular, a legal behavior of this automaton is to stay in location ℓ ad infinitum. Formally,

$$Loc = Loc_0 = \{\,\ell\,\},\ C = \{\,x\,\},\ \ell \xleftarrow{\text{true}:x\geqslant 2,\{\,x\,\}} \ell, \text{ and } Inv(\ell) = \text{true}.$$

Labelings and actions are omitted.

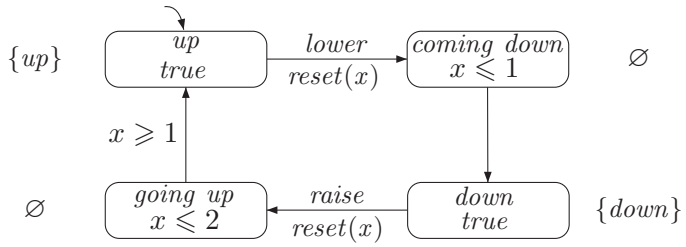


Figure 9.6: Timed automaton for the *Gate*.

Changing the timed automaton of Figure 9.5(a) slightly by incorporating a location invariant $x \leqslant 3$ in location ℓ leads to the effect that x cannot progress unboundedly anymore. Rather, if $x \geqslant 2$ (guard) and $x \leqslant 3$ (invariant) the outgoing transition must be taken. Note that it is not specified at which time instant in the interval $[2, 3]$ the transition is taken, i.e., this is determined nondeterministically. The timed automaton and an example of its behavior are illustrated in Figure 9.5(c) and (d), respectively.

Observe that the same effect is not obtained when strengthening the guard in Figure 9.5(a) into $2 \leqslant x \leqslant 3$ while keeping $Inv(\ell) = \text{true}$. In that case, the outgoing transition can only be taken when $2 \leqslant x \leqslant 3$—as in the previous scenario—but is not forced to be taken, i.e., it can simply be ignored by letting time pass while staying in ℓ. This is illustrated in Figure 9.5(e) and (f).

Put in a nutshell, invariants are the only means to *force* transitions to be taken. ■

Example 9.5. Timed Automaton for the Gate

Consider the gate for the railroad crossing (see Example 9.1, page 674). Assuming that lowering the gate takes at most a single time unit, and raising the gate takes at least one and at most two time units, the timed automaton for process *Gate* is given as in Figure 9.6. We have $Act = \{ \mathit{lower}, \mathit{raise} \}$ and

$$Loc \;=\; \{ \mathit{up}, \mathit{comingdown}, \mathit{down}, \mathit{goingup} \}$$

with $Loc_0 = \{ \mathit{up} \}$. The transitions of the timed automaton are

$$\mathit{up} \xrightarrow{\text{true: } \mathit{lower},\ \{x\}} \mathit{comingdown} \qquad \mathit{comingdown} \xrightarrow{\text{true: } \tau,\ \varnothing} \mathit{down}$$

$$\mathit{down} \xrightarrow{\text{true: } \mathit{raise},\ \{x\}} \mathit{goingup} \qquad \mathit{goingup} \xrightarrow{x \geqslant 1\text{: } \tau,\ \varnothing} \mathit{up}$$

The location *coming down* with invariant $x \leqslant 1$ has been added to model that the maximal delay between the occurrence of action *lower* and the change to location *down* is at most

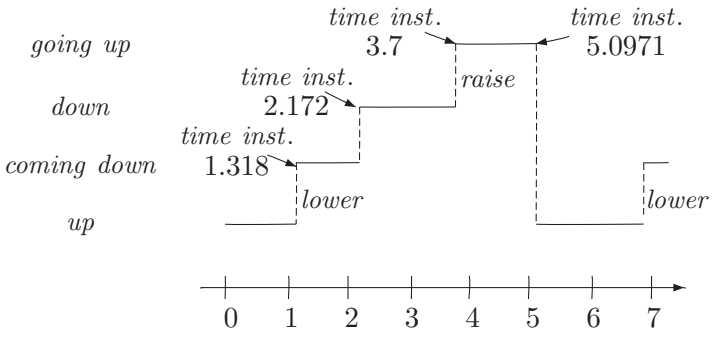


Figure 9.7: Location diagram for the timed automaton *Gate*.

a single time unit. Clock x is set to zero on the occurrence of action *lower* and thus "measures" the elapse of time since that occurrence. By restricting the residence time of *coming down* to $x \leqslant 1$, the switch to *down* must be made within one time unit. Note that this would not have been established by having a direct edge between locations *up* and *down* with guard $x \leqslant 1$, as the value of x would not refer to the time of occurrence of *lower*. In a similar way, the purpose of location *goingup* with invariant $x \leqslant 2$ is to model that raising the gate takes at most two time units. In the initial location *up*, no constraints are imposed on the residence time, i.e., $Inv(up) = \text{true}$. The same applies to location *down*. Let $AP = \{up, down\}$ with the labeling function $L(up) = \{ up \}$, $L(down) = \{ down \}$, and $L(comingdown) = L(goingup) = \varnothing$. ■

Remark 9.6. Location Diagram

Every finite behavior of a timed automaton can be represented by a *location diagram*. This depicts for every time instant up to some a priori fixed upper bound, the location of the timed automaton during that behavior. For the timed automaton of the *Gate* a possible real-time behavior is indicated by the location diagram in Figure 9.7. ■

Parallel Composition of Timed Automata For modeling complex systems it is convenient to allow parallel composition of timed automata. This allows for the modeling of time-critical systems in a compositional manner. We consider a parallel composition operator, denoted $||_H$, that is parameterized with a set of *handshaking* actions H. This operator is similar in spirit to the corresponding operator on transition systems; see Definition 2.26, page 48: actions in H need to be performed jointly by both involved timed

automata, whereas actions outside H are performed autonomously in an interleaved fashion.

Definition 9.7. Handshaking for Timed Automata

Let $TA_i = (Loc_i, Act_i, C_i, \hookrightarrow_i, Loc_{0,i}, Inv_i, AP_i, L_i)$, $i = 1, 2$ be timed automata with $H \subseteq Act_1 \cap Act_2$, $C_1 \cap C_2 = \varnothing$ and $AP_1 \cap AP_2 = \varnothing$. The timed automaton $TA_1 \parallel_H TA_2$ is defined as

$$(Loc_1 \times Loc_2, Act_1 \cup Act_2, C_1 \cup C_2, \hookrightarrow, Loc_{0,1} \times Loc_{0,2}, Inv, AP_1 \cup AP_2, L)$$

where $L(\langle \ell_1, \ell_2 \rangle) = L_1(\ell_1) \cup L_2(\ell_2)$ and $Inv(\langle \ell_1, \ell_2 \rangle) = Inv_1(\ell_1) \wedge Inv_2(\ell_2)$. The transition relation $\hookrightarrow$ is defined by the following rules:

- for $\alpha \in H$:

$$\frac{\ell_1 \xrightarrow{g_1:\alpha, D_1}_1 \ell_1' \ \wedge \ \ell_2 \xrightarrow{g_2:\alpha, D_2}_2 \ell_2'}{\langle \ell_1, \ell_2 \rangle \xrightarrow{g_1 \wedge g_2:\alpha, D_1 \cup D_2} \langle \ell_1', \ell_2' \rangle}$$

- for $\alpha \notin H$:

$$\frac{\ell_1 \xrightarrow{g:\alpha, D}_1 \ell_1'}{\langle \ell_1, \ell_2 \rangle \xrightarrow{g:\alpha, D} \langle \ell_1', \ell_2 \rangle} \quad \text{and} \quad \frac{\ell_2 \xrightarrow{g:\alpha, D}_2 \ell_2'}{\langle \ell_1, \ell_2 \rangle \xrightarrow{g:\alpha, D} \langle \ell_1, \ell_2' \rangle}$$

■

The location invariant of a composite location is simply the conjunction of the location invariants of its constituents. For $\alpha \in H$, the transition in the resulting timed automaton is guarded by the conjunction of the guards of the individual timed automata. This entails that an action in H can only be taken when it is enabled in both timed automata. Besides, the clocks that are reset in the individual automata are all reset. As for transition systems, the operator $\parallel_H$ is associative for a fixed set H. Let $TA_1 \parallel_H TA_2 \parallel_H \ldots \parallel TA_n$ denote the parallel composition of timed automata TA_1 through TA_n where $H \subseteq Act_1 \cap \ldots \cap Act_n$, assuming that all timed automata are compatible, i.e., each pair of TA_i and TA_j, $i \neq j$ have disjoint sets of atomic propositions and disjoint clocks.

Example 9.8. Railroad Crossing

Consider again the railroad crossing example. We extend the timed automaton for the gate (see Example 9.5) with timed automata for the controller and the train. The complete system is then given by

$$(Train \parallel_{H_1} Controller) \parallel_{H_2} Gate$$

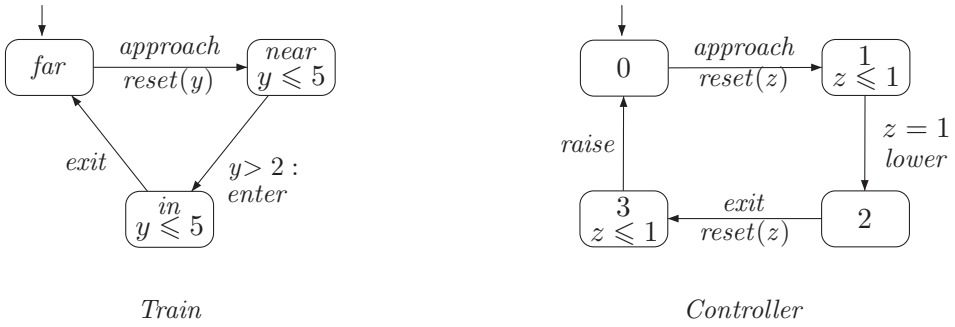


Figure 9.8: Timed automata for the train and the controller .

where $H_1 = \{ \mathit{approach}, \mathit{exit} \}$ and $H_2 = \{ \mathit{lower}, \mathit{raise} \}$.

Let us assume that the train signals its approaching of the gate at least two time units before it enters the railroad crossing. Besides, it is assumed that the train has sufficient speed such that it leaves the crossing five time units after approaching it, at the latest. The timed automaton for the *Train* is depicted in Figure 9.8 (left). On approaching the gate, clock y is set to zero, and only if $y > 2$ is the train allowed to enter the crossing. The *Controller* is depicted in Figure 9.8 (right) and is forced to send the signal "lower" (to the *Gate*) exactly after one time unit after the *Train* has signaled its approaching.

Figure 9.9 shows the composite timed automaton. The prefix of a possible behavior of the complete system is sketched in the location diagram in Figure 9.10.

Note that this timed automaton contains the location $\langle \mathit{in}, 1, \mathit{up} \rangle$. In this location, the train is at the crossing while the gate it still open. However, this location turns out to be unreachable. It can only be reached when $y > 2$, but as y and z are reset at the same time (on entrance of the preceding location), $y > 2$ implies $z > 2$, which is impossible due to the location invariant $z \leqslant 1$. ■

9.1.1 Semantics

The previous examples suggest that the state of a timed automaton is determined by its current location and the current values of all its clocks. In fact, any timed automaton can—like program graphs—be interpreted as a transition system. Due to the continuous time domain, these underlying transition systems have infinitely many states (even uncountably

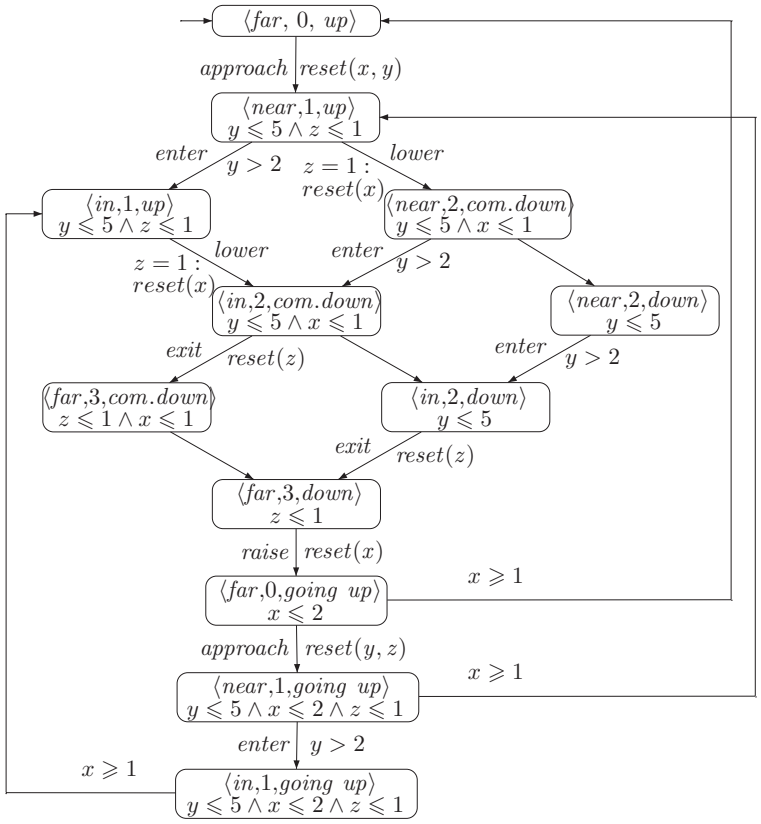


Figure 9.9: The timed automaton $(\mathit{Train} \|_{H_1} \mathit{Controller}) \|_{H_2} \mathit{Gate}$.

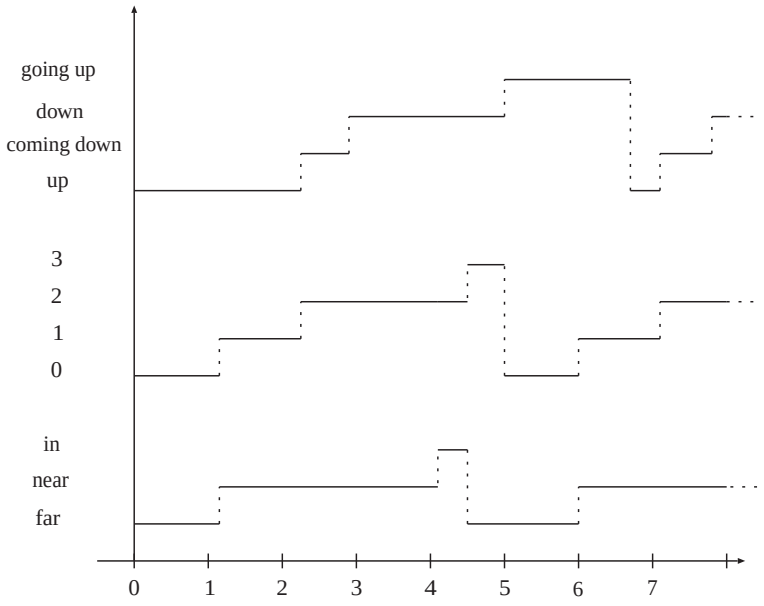


Figure 9.10: Location diagram for a behavior of $(\mathit{Train} \|_{H_1} \mathit{Controller}) \|_{H_2} \mathit{Gate}$.

many), and are infinitely branching. Timed automata can thus be considered as *finite* descriptions of infinite transition systems. The underlying transition system of a timed automaton results from unfolding. Its states consist of a control component, i.e., a location ℓ of the timed automaton, together with a valuation η of the clocks. States are thus pairs of the form $\langle \ell, \eta \rangle$. Let us first consider clock valuations.

Definition 9.9. Clock Valuation

A *clock valuation* η for a set C of clocks is a function $\eta : C \to \mathbb{R}_{\geqslant 0}$, assigning to each clock $x \in C$ its current value $\eta(x)$. ■

Let $\mathit{Eval}(C)$ denote the set of all clock valuations over C. In the following, we often use notations like $[x = v, y = v']$ to denote the clock evaluation $\eta \in \mathit{Eval}(\{ x, y \})$ with $\eta(x) = v$ and $\eta(y) = v'$.

We can now formally define what it means for a clock constraint to hold for a clock valuation or not. This is done in a similar way as characterizing the semantics of a temporal logic, namely by defining a satisfaction relation. In this case the satisfaction relation $\models$ is a relation between clock valuations (over a set of clocks C) and clock constraints (over C).

Definition 9.10. Satisfaction Relation for Clock Constraints

For set C of clocks, $x \in C$, $\eta \in \mathit{Eval}(C)$, $c \in \mathbb{N}$, and $g, g' \in \mathit{CC}(C)$, let $\models \subseteq \mathit{Eval}(C) \times \mathit{CC}(C)$ be defined by

$$\begin{array}{ll} \eta \models \text{true} & \\ \eta \models x < c & \text{iff } \eta(x) < c \\ \eta \models x \leqslant c & \text{iff } \eta(x) \leqslant c \\ \eta \models \neg\, g & \text{iff } \eta \not\models g \\ \eta \models g \wedge g' & \text{iff } \eta \models g \ \wedge \ \eta \models g' \end{array}$$

■

Let η be a clock valuation on C. For positive real d, $\eta{+}d$ denotes the clock valuation where all clocks of η are increased by d. Formally, $(\eta{+}d)(x) = \eta(x) + d$ for all clocks $x \in C$. $\mathsf{reset}\ x\ \mathsf{in}\ \eta$ denotes the clock valuation which is equal to η except that clock x reset. Formally:

$$(\mathsf{reset}\ x\ \mathsf{in}\ \eta)(y) = \begin{cases} \eta(y) & \text{if } y \neq x \\ 0 & \text{if } y = x. \end{cases}$$

For the clock valuation $\eta = [x = \pi, y = 4]$, valuation $\eta{+}9 = [x = \pi{+}9, y = 13]$, and $\mathsf{reset}\ x\ \mathsf{in}\ (\eta{+}9) = [x = 0, y = 13]$. Nested occurrences of reset are typically abbreviated. For instance, $\mathsf{reset}\ x\ \mathsf{in}\ \ (\mathsf{reset}\ y\ \mathsf{in}\ \eta)$ is denoted $\mathsf{reset}\ x, y\ \mathsf{in}\ \eta$.

There are two possible ways in which a timed automaton can proceed: by taking a transition in the timed automaton, or by letting time progress while staying in a location. In the underlying transition system, the former is represented by a *discrete* transition and the latter by a *delay* transition. In the former case, the corresponding transition of the underlying transition system is labeled with the action of the transition in the timed automaton, in the latter case, it is labeled with a positive real number indicating the amount of time that has elapsed.

Definition 9.11. Transition System Semantics of a Timed Automaton

Let $\mathit{TA} = (\mathit{Loc}, \mathit{Act}, C, \hookrightarrow, \mathit{Loc}_0, \mathit{Inv}, \mathit{AP}, L)$ be a timed automaton. The transition system $\mathit{TS}(\mathit{TA}) = (S, \mathit{Act}', \rightarrow, I, \mathit{AP}', L)$ with:

- $S = \mathit{Loc} \times \mathit{Eval}(C)$
- $\mathit{Act}' = \mathit{Act} \cup \mathbb{R}_{\geqslant 0}$
- $I = \{\, \langle \ell_0, \eta \rangle \mid \ell_0 \in \mathit{Loc}_0 \ \wedge \ \eta(x) = 0 \text{ for all } x \in C \,\}$
- $\mathit{AP}' = \mathit{AP} \cup \mathit{ACC}(C)$

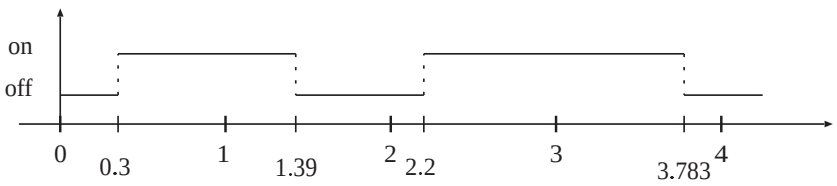
- $L'(\langle \ell, \eta \rangle) = L(\ell) \cup \{\, g \in ACC(C) \mid \eta \models g \,\}$
- the transition relation $\longrightarrow$ is defined by the following two rules:
  - *discrete transition*: $\langle \ell, \eta \rangle \xrightarrow{\alpha} \langle \ell', \eta' \rangle$ if the following conditions hold:
    (a) there is a transition $\ell \overset{g:\alpha,D}{\hookrightarrow} \ell'$ in TA
    (b) $\eta \models g$
    (c) $\eta' = \mathsf{reset}\ D\ \mathsf{in}\ \eta$
    (d) $\eta' \models Inv(\ell')$
  - *delay transition*: $\langle \ell, \eta \rangle \xrightarrow{d} \langle \ell, \eta{+}d \rangle$ for $d \in \mathbb{R}_{\geqslant 0}$
    (e) if $\eta{+}d \models Inv(\ell)$

■

For a transition that corresponds to (a) traversing a transition $\ell \overset{g:\alpha,D}{\hookrightarrow} \ell'$ in the timed automaton TA it must hold that (b) η satisfies the clock constraint g (ensuring the transition is enabled), and (c) the new clock valuation η' is obtained by resetting all clocks D in η should (d) satisfy the location invariant of ℓ' (otherwise it is not allowed to be in ℓ'). Idling in a location (second clause) for some non-negative amount of time is allowed (e) if the location invariant remains true while time progresses. For state $\langle \ell, \eta \rangle$ such that $\eta \models Inv(\ell)$, there are typically uncountably many outgoing delay transitions of the form $\langle \ell, \eta \rangle \xrightarrow{d}$ as d can be selected from a continuous domain.

Example 9.12. Light Switch

The timed automaton *Switch* in Figure 9.11 models a switch that controls a light. When off, the switch may be turned on at any time instant. The user may switch off the light at least one time unit after the most recent time the light was switched on. After two time units the light automatically switches off. Clock x is used to keep track of the delay since the last time the light has been switched on. (The timed automaton does not distinguish between the *switch_off* action activated by the user and by the light. This could be made explicit by adding an edge from location *on* to *off*, with guard $x = 2$ and action τ.) The following location diagram indicates a possible behavior:



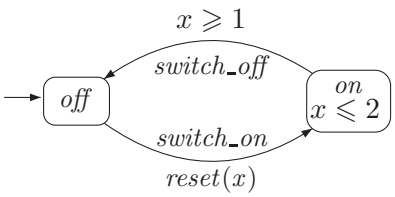


Figure 9.11: A simple light switch.

The transition system $TS(Switch)$ has the state space

$$S \;=\; \{\langle \mathit{off}, t\rangle \mid t \in \mathbb{R}_{\geqslant 0}\} \;\cup\; \{\langle \mathit{on}, t\rangle \mid t \in \mathbb{R}_{\geqslant 0}\}$$

where t is a shorthand for the clock evaluation η with $\eta(x) = t$. Uncountably many transitions emanate from the initial state $\langle \mathit{off}, 0\rangle$. $TS(Switch)$ has the following transitions for reals d and t:

$$\begin{array}{lcll}
\langle \mathit{off}, t\rangle & \xrightarrow{d} & \langle \mathit{off}, t+d\rangle & \text{for all } t \geqslant 0 \text{ and } d \geqslant 0 \\
\langle \mathit{off}, t\rangle & \xrightarrow{\mathit{switch_on}} & \langle \mathit{on}, 0\rangle & \text{for all } t \geqslant 0 \\
\langle \mathit{on}, t\rangle & \xrightarrow{d} & \langle \mathit{on}, t+d\rangle & \text{for all } t \geqslant 0 \text{ and } d \geqslant 0 \text{ with } t+d \leqslant 2 \\
\langle \mathit{on}, t\rangle & \xrightarrow{\mathit{switch_off}} & \langle \mathit{off}, t\rangle & \text{for all } 1 \leqslant t \leqslant 2.
\end{array}$$

The set of reachable states in $TS(Switch)$ from state $\langle \mathit{off}, 0\rangle$ is

$$\{\langle \mathit{off}, t\rangle \mid t \in \mathbb{R}_{\geqslant 0}\} \;\cup\; \{\langle \mathit{on}, t\rangle \mid 0 \leqslant t \leqslant 2\}$$

as the location invariant $x \leqslant 2$ is violated in any state $\langle \mathit{on}, t\rangle$ with $t > 2$. A prefix of an example path of $TS(Switch)$ is

$$\begin{array}{l}
\langle \mathit{off}, 0\rangle \xrightarrow{0.57} \langle \mathit{off}, 0.57\rangle \xrightarrow{\mathit{switch_on}} \langle \mathit{on}, 0\rangle \xrightarrow{\sqrt{2}} \langle \mathit{on}, \sqrt{2}\rangle \xrightarrow{0.2} \\
\langle \mathit{on}, \sqrt{2}{+}0.2\rangle \xrightarrow{\mathit{switch_off}} \langle \mathit{off}, \sqrt{2}{+}0.2\rangle \xrightarrow{\mathit{switch_on}} \langle \mathit{on}, 0\rangle \xrightarrow{1.7} \langle \mathit{on}, 1.7\rangle \ldots
\end{array}$$

■

Remark 9.13. Parallel Composition

For timed automata we have

$$TS(TA_1) \parallel_{H \cup \mathbb{R}_{>0}} TS(TA_2) \;=\; TS(TA_1 \parallel_H TA_2)$$

up to isomorphism. This is due to the fact that TA_1 and TA_2 do not have any shared variables. Synchronization over time passage actions reflects the natural fact that time proceeds equally fast in both components. ■

The paths in $TS(TA)$ are discrete representations of the continuous-time "behavior" of TA. They indicate at least the states immediately before and after the execution of an action $\alpha \in Act$. However, due to the fact that, e.g., interval delays may be realized in uncountably many ways, different paths may describe the same behavior (i.e., location diagram). Consider, e.g., the behavior of the light switch of Example 9.12 where the light alternates between *off* and *on* while being *off* for exactly one time unit and *on* for two time units, i.e., a return to *off* takes place after exactly three time units. The following three example paths correspond to this continuous-time behavior:

$$\begin{array}{lcccccccccc}
\pi_1 & = & \langle off,0\rangle & & & \langle off,1\rangle & \langle on,0\rangle & & & \langle on,2\rangle & \langle off,2\rangle & \ldots \\
\pi_2 & = & \langle off,0\rangle & \langle off,0.5\rangle & & \langle off,1\rangle & \langle on,0\rangle & & \langle on,1\rangle & \langle on,2\rangle & \langle off,2\rangle & \ldots \\
\pi_3 & = & \langle off,0\rangle & \langle off,0.1\rangle & & \langle off,1\rangle & \langle on,0\rangle & \langle on,0.53\rangle & \langle on,1.3\rangle & \langle on,2\rangle & \langle off,2\rangle & \ldots
\end{array}$$

The only difference between these paths is the delay transitions. In path π_1, the one time unit residence in the initial state $\langle off,0\rangle$ is realized by means of the delay transition $\langle off,0\rangle \xrightarrow{1} \langle off,1\rangle$. In contrast, the paths π_2 and π_3 realize this single time unit by two delay transitions:

$$\begin{array}{lcl}
\langle off,0\rangle & \xrightarrow{0.5} & \langle off,0.5\rangle \xrightarrow{0.5} \langle off,1\rangle \quad \text{and} \\
\langle off,0\rangle & \xrightarrow{0.1} & \langle off,0.1\rangle \xrightarrow{0.9} \langle off,1\rangle.
\end{array}$$

But the effect of the transition $\langle \ell,\eta\rangle \xrightarrow{d_1+d_2} \langle \ell,\eta{+}d_1{+}d_2\rangle$ corresponds to the effect of the sequence of transitions:

$$\langle \ell,\eta\rangle \xrightarrow{d_1} \langle \ell,\eta{+}d_1\rangle \xrightarrow{d_2} \langle \ell,\eta{+}d_1{+}d_2\rangle.$$

In both cases, $d_1{+}d_2$ time units pass without executing an action $\alpha \in Act$. Thereby, uncountably many states of the form $\langle \ell,\eta{+}t\rangle$ with $0 \leqslant t \leqslant d_1{+}d_2$ are passed through.

Remark 9.14. Multiple Actions in Zero Time

The elapse of time in timed automata only takes place in locations. Actions $\alpha \in Act$ take place instantaneously, i.e., they have a duration of zero time units. As a result, at a single time instant, several actions take place. ■

9.1.2 Time Divergence, Timelock, and Zenoness

The semantics of a timed automaton is given by a transition system with uncountably many states (and transitions). Paths through this transition system correspond to possible

behaviors of the timed automaton. However, not every such path represents a realistic behavior. This subsection treats three essential phenomena for timed automata: time divergence, timelock, and zenoness.

Time Divergence Consider a location ℓ such that for any $t < d$, for fixed constant $d \in \mathbb{R}_{>0}$, clock valuation $\eta{+}t \models \mathit{Inv}(\ell)$. A possible execution fragment starting from this location is

$$\langle \ell, \eta \rangle \xrightarrow{d_1} \langle \ell, \eta{+}d_1 \rangle \xrightarrow{d_2} \langle \ell, \eta{+}d_1{+}d_2 \rangle \xrightarrow{d_3} \langle \ell, \eta{+}d_1{+}d_2{+}d_3 \rangle \xrightarrow{d_4} \ldots$$

where $d_i > 0$ and the infinite sequence $d_1 + d_2 + \ldots$ converges toward d. Such infinite path fragments are called *time-convergent*. A time-convergent path is counterintuitive as time advances only up to a certain value whereas by nature time always progresses. For example, the transition system of the timed automaton for the light switch (see Figure 9.11 on page 689) exhibits the time-convergent execution fragment

$$\langle \mathit{off}, 0 \rangle \xrightarrow{2^{-1}} \langle \mathit{off}, 1{-}2^{-1} \rangle \xrightarrow{2^{-2}} \langle \mathit{off}, 1{-}2^{-2} \rangle \xrightarrow{2^{-3}} \langle \mathit{off}, 1{-}2^{-3} \rangle \ \ldots\ldots$$

which visits infinitely many states in the interval $[\frac{1}{2}, 1]$. Time never proceeds beyond one. The corresponding path is time-convergent. As time-convergent paths are not realistic, they are not considered. That is to say, the analysis of timed automata is focused on *time-divergent* paths, i.e., paths in which time always progresses.

In order to formally define time-divergent paths, let us first define the elapsed time of a path. Intuitively, the elapsed time of a path is the total time that elapses along a path. The duration of an action $\alpha \in \mathit{Act}$ is zero; the duration of a delay action d is d.

Definition 9.15. Elapsed Time on a Path

Let TA be a timed automaton with the set Act of actions. The function $\mathit{ExecTime} : \mathit{Act} \cup \mathbb{R}_{>0} \to \mathbb{R}_{\geqslant 0}$ is defined as

$$\mathit{ExecTime}(\tau) \ = \ \begin{cases} 0 & \text{if } \tau \in \mathit{Act} \\ d & \text{if } \tau = d \in \mathbb{R}_{>0}. \end{cases}$$

For infinite execution fragment $\rho = s_0 \xrightarrow{\tau_0} s_1 \xrightarrow{\tau_1} s_2 \ldots$ in $TS(TA)$ with $\tau_i \in \mathit{Act} \cup \mathbb{R}_{>0}$ let

$$\mathit{ExecTime}(\rho) \ = \ \sum_{i=0}^{\infty} \mathit{ExecTime}(\tau_i).$$

The execution time of finite execution fragments is defined analogously. For the path fragment π in $TS(TA)$ induced by ρ, $\mathit{ExecTime}(\pi) \ = \ \mathit{ExecTime}(\rho)$. ■

Note that path fragment π may be induced by several execution fragments. However, every pair of execution fragments with the same path fragment are only distinguished by discrete transitions and not by delay transitions. Hence, $\mathit{ExecTime}(\pi)$ is well-defined.

Definition 9.16. Time Divergence and Time Convergence

The infinite path fragment π is *time-divergent* if $\mathit{ExecTime}(\pi) = \infty$; otherwise, π is *time-convergent*. ■

Example 9.17. Light Switch

For the light switch described in Example 9.12 (page 688), the path π in $TS(\mathit{Switch})$ in which on and off periods of 1 minute alternate:

$$\pi = \langle \mathit{off}, 0\rangle \langle \mathit{off}, 1\rangle \langle \mathit{on}, 0\rangle \langle \mathit{on}, 1\rangle \langle \mathit{off}, 1\rangle \langle \mathit{off}, 2\rangle \langle \mathit{on}, 0\rangle \langle \mathit{on}, 1\rangle \langle \mathit{off}, 1\rangle \ldots$$

is time-divergent as $\mathit{ExecTime}(\pi) = 1 + 1 + 1 + \ldots = \infty$. The path

$$\pi' \;=\; \langle \mathit{off}, 0\rangle \langle \mathit{off}, 1/2\rangle \langle \mathit{off}, 3/4\rangle \langle \mathit{off}, 7/8\rangle \langle \mathit{off}, 15/16\rangle \ldots$$

in $TS(\mathit{Switch})$ is time-convergent, as $\mathit{ExecTime}(\pi') \;=\; \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^{i+1} = 1 < \infty$. ■

Definition 9.18. Time-Divergent Set of Paths

For state s in $TS(TA)$ let: $\mathit{Paths}_{div}(s) = \{\, \pi \in \mathit{Paths}(s) \mid \pi \text{ is time-divergent } \}$. ■

That is, $\mathit{Paths}_{div}(s)$ denotes the set of time-divergent paths in $TS(TA)$ that start in s. Although time-convergent paths are not realistic, their existence cannot be avoided. For the analysis of timed automata, time-convergent paths are simply ignored, e.g., a timed automaton satisfies an invariant when along all its time-divergent paths the invariant is satisfied.

Timelock. State s in $TS(TA)$ contains a *timelock* if there is no time-divergent path starting in s. Such states are unrealistic as time cannot progress for ever from these states. Timelocks are considered as undesired and need to be avoided when modeling a time-critical system by means of a timed automaton.

Definition 9.19. Timelock

Let TA be a timed automaton. State s in $TS(TA)$ contains a *timelock* if $\mathit{Paths}_{div}(s) = \varnothing$. TA is *timelock-free* if no state in $\mathit{Reach}(TS(TA))$ contains a timelock. ■

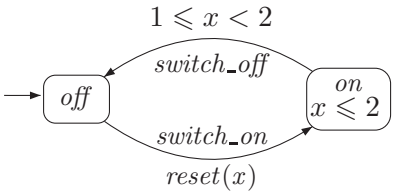


Figure 9.12: Timed automaton $Switch_1$.

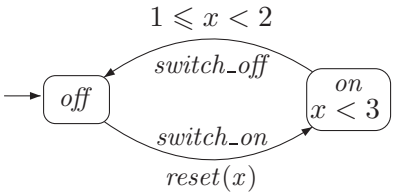


Figure 9.13: Timed automaton $Switch_2$.

Example 9.20. Modified Light Switch

We modify the light switch such that the light is on for a period with duration $t \in [1, 2)$, i.e., the light is always switched off within 2 minutes; see the timed automaton $Switch_1$ in Figure 9.12. The state $\langle on, 2\rangle$ is reachable in transition system $TS(Switch_1)$, e.g., via the execution fragment:

$$\langle off, 0\rangle \xrightarrow{switch\_on} \langle on, 0\rangle \xrightarrow{2} \langle on, 2\rangle.$$

As $\langle on, 2\rangle$ is a terminal state, $Paths_{div}(\langle on, 2\rangle) = \varnothing$, and the state contains a timelock. Timed automaton $Switch_1$ is thus not timelock-free.

Any terminal state of a transition system that results from a timed automaton contains a timelock. Terminal states should not be confused with terminal locations, i.e., locations that have no outgoing edges. A terminal location ℓ with $Inv(\ell) = \text{true}$, e.g., does not result in a terminal state in the underlying transition system, as time may progress in ℓ for ever. Terminal locations thus do not necessarily yield states with timelocks.

Not only terminal states may contain timelocks. Consider, e.g., another variant of the light switch where $Inv(on) = x < 3$, see timed automaton $Switch_2$ in Figure 9.13. The reachable state $\langle on, 2\rangle$ is not terminal, , e.g., the time-convergent path in $TS(Switch_2)$:

$$\langle on, 2\rangle \langle on, 2.9\rangle \langle on, 2.99\rangle \langle on, 2.999\rangle \langle on, 2.9999\rangle \ldots$$

emanates from it. However, $Paths_{div}(\langle on, 2\rangle) = \varnothing$ as the state $\langle on, 2\rangle$ has no outgoing discrete transitions (as the guard $1 \leqslant x < 2$ is violated), and time cannot progress beyond three (due to $Inv(on) = x < 3$). State $\langle on, 2\rangle$ in $TS(Switch_2)$ contains a timelock. Timed automaton $Switch_2$ is thus not timelock-free. ■

Zenoness As opposed to the presence of time-convergent paths, timelocks are considered as modeling flaws that should be avoided. The latter also applies to zenoness. Recall that the execution of actions $\alpha \in Act$ is instantaneous, i.e., actions take no time. Without further restrictions, a timed automaton may perform infinitely many actions in a finite time interval. This phenomenon is also called *zeno* and represents nonrealizable behavior, since it would require infinitely fast processors.

Definition 9.21. Zeno Paths

Let TA be a timed automaton. The infinite path fragment π in $TS(TA)$ is *zeno* (or: a *zeno path*) if it is time-convergent and infinitely many actions $\alpha \in Act$ are executed along π. ■

Definition 9.22. Nonzenoness

A timed automaton TA is *non-zeno* if there does not exist an initial zeno path in $TS(TA)$. ■

Timed automaton TA is thus non-zeno if and only if for every path π in $TS(TA)$, either π is time-divergent or π is time-convergent with almost only (i.e., all except for finitely many) delay transitions.

Note that non-zenoness, as well as timelock freedom, only refers to the *reachable* fragment of the transition system $TS(TA)$. A non-zeno timed automaton may possess zeno paths starting in an unreachable state. Similarly, a timelock-free timed automaton may contain unreachable timelock states.

Example 9.23. Zeno Paths of a Light Switch

Consider yet another variant of the light switch, in which the user has the possibility to push the on button while the light is on. While doing so, clock x is reset, and the light stays on for at most two units, unless the user pushes the on button again; see timed automaton $Switch_3$ in Figure 9.14. The paths induced by the following execution fragments of $TS(Switch_3)$

$$\langle off, 0\rangle \xrightarrow{switch\_on} \langle on, 0\rangle \xrightarrow{switch\_on} \langle on, 0\rangle \xrightarrow{switch\_on} \langle on, 0\rangle \xrightarrow{switch\_on} \ldots$$

$$\langle off, 0\rangle \xrightarrow{switch\_on} \langle on, 0\rangle \xrightarrow{0.5} \langle on, 0.5\rangle \xrightarrow{switch\_on} \langle on, 0\rangle \xrightarrow{0.25} \langle on, 0.25\rangle \xrightarrow{switch\_on} \ldots,$$

are zeno paths during which the user presses the on button infinitely fast, or faster and faster, respectively.

This unrealizable behavior can be avoided by imposing a minimal non-zero delay, c, say, between successive button pushings by the user. This is established by imposing $Inv(on) =$

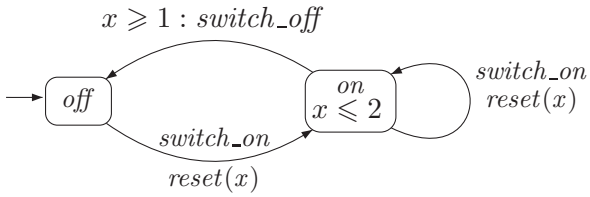


Figure 9.14: Timed automaton $Switch_3$.

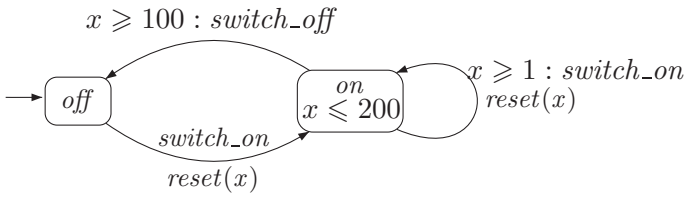


Figure 9.15: Timed automaton $Switch_4$.

$x \geqslant c$ for $c > 0$. Note that c should be a natural number. In order to model a minimal response $0 < c < 1$ where c is rational, say $c = \frac{1}{100}$, all time constraints in the timed automaton $Switch_3$ need to be rescaled (see Figure 9.15). Essentially, the timed automaton $Switch_4$ computes with a modified time unit: one time unit for x in $Switch_4$ corresponds to $\frac{1}{100}$ minutes.

■

To check whether or not a timed automaton is non-zeno is algorithmically difficult. Instead, sufficient conditions are considered that are simple to check, e.g., by a static analysis of the timed automaton. The following criterion is based on the intuition that a timed automaton is non-zeno if on any of its control cycles, time advances with at least some constant amount (larger than zero). This yields:

Lemma 9.24. Sufficient Criterion for Nonzenoness

Let TA be a timed automaton with set C of clocks such that for every control cycle in TA

$$\ell_0 \xrightarrow{g_1:\alpha_1,C_1} \ell_1 \xrightarrow{g_2:\alpha_2,C_2} \ldots \xrightarrow{g_n:\alpha_n,C_n} \ell_n \text{ with } \ell_0 = \ell_n,$$

there exists a clock $x \in C$ such that

1. *$x \in C_i$ for some $0 < i \leqslant n$, and*

2. for all clock evaluations η there exists $c \in \mathbb{N}_{>0}$ such that

$$\eta(x) < c \quad \textit{implies} \quad (\eta \not\models g_j \ \textit{or}\ \mathit{Inv}(\ell_j)), \ \textit{for some}\ 0 < j \leqslant n.$$

Then: TA is non-zeno.

Proof: Let TA be a timed automaton over C with $x \in C$ satisfying the two constraints stated in the claim and i, j the corresponding indices. Let π be a path in $TS(TA)$ that performs infinitely many actions $\alpha \in Act$. As TA contains finitely many states, π traverses some control cycle $\ell_0 \hookrightarrow \ell_1 \hookrightarrow \ldots \hookrightarrow \ell_n = \ell_0$. Assume that $i \leqslant j$. (As locations on cycles can be renumbered, this is not a restriction.) Consider the path fragment in $TS(TA)$ that starts and ends in location ℓ_0:

$$\langle \ell_0, \eta_0 \rangle \ldots \langle \ell_{i-1}, \eta_{i-1} \rangle \langle \ell_i, \eta_i \rangle \ldots \langle \ell_{j-1}, \eta_{j-1} \rangle \langle \ell_j, \eta_j \rangle \ldots \langle \ell_0, \eta'_0 \rangle.$$

By the constraints satisfied by TA, clock x is reset on the transition $\ell_{i-1} \hookrightarrow \ell_i$, and the transition $\ell_{j-1} \hookrightarrow \ell_j$ is only possible when $\eta_{i-1}(x) \geqslant c$ (as for $\eta_{i-1}(x) < c$, either the guard or the location invariant of ℓ_j are violated). This implies that by traversing the cycle $\ell_0 \hookrightarrow \ldots \hookrightarrow \ell_0$ time advances with at least $c > 0$ time units. Hence, π is time-divergent and TA is non-zeno. ■

The condition in Lemma 9.24 is compositional, i.e., if TA and TA' both satisfy the constraints, then the parallel composed timed automaton $TA \parallel TA'$ also satisfies the constraints. This follows directly from the fact that a control cycle in $TA \parallel TA'$ consists of a control cycle in TA, or TA', or both. If each control cycle in TA and in TA' satisfies the constraint that time should advance with at least some positive amount, then this thus also applies to each control cycle in $TA \parallel TA'$. This property significantly simplifies to checking whether a composite timed automaton is non-zeno. In case a timed automaton is in fact untimed (as no clocks are used or all guards and invariants are vacuously true), it can be considered as non-zeno, and thus not affect the control cycles of other component timed automata in a composite system.

Example 9.25. Sufficient Condition for Nonzenoness

The timed automaton in Figure 9.15 (page 695) satisfies the constraints in Lemma 9.24. In the control cycle *off* $\hookrightarrow$ *on* $\hookrightarrow$ *off*, clock x is reset on *off* $\hookrightarrow$ *on*, and the guard $x \geqslant 100$ ensures that when going from location *off* to *on*, time has advanced with at least 100 time units. On the control cycle *on* $\hookrightarrow$ *on*, clock x is reset, and the guard $x \geqslant 1$ ensures that at least one time unit elapses on traversing this control cycle.

The timed automata *Train*, *Gate*, and *Controller* of Example 9.8 (page 683) all satisfy for any control cycle the constraints in Lemma 9.24. This can be seen as follows. Timed

automaton *Gate* has one control cycle: $up \hookrightarrow \ldots \hookrightarrow up$. In that cycle, clock x is reset when moving from location *down* to *going_up*. Furthermore, for $\eta(x) < 1$, location *up* is not reachable due to the guard $x \geqslant 1$ on $going\_up \hookrightarrow up$. This ensures that on traversing the control cycle $up \hookrightarrow \ldots \hookrightarrow up$, time advances with at least one time unit. The timed automaton *Train* contains the control cycle $far \hookrightarrow \ldots \hookrightarrow far$. However, clock y is reset on that cycle before reaching location *near*, and the guard $y > 2$ on $near \hookrightarrow in$ guarantees that on traversing the control cycle at least one (in fact, more than two) time unit has elapsed. For *Controller*, the resetting of clock z and guard $z{=}1$ ensure that also this timed automaton fulfills the constraints of Lemma 9.24. The timed automata *Train*, *Gate* and *Controller* are thus non-zeno. As the control cycles in the composed timed automaton $(Train \parallel_{H_1} Gate) \parallel_{H_2} Controller$ are comprised of the constituting timed automata, this composed timed automaton is non-zeno.

■

The previous considerations indicate that a timed automaton is adequately modeling a time-critical system whenever it is non-zeno and does not contain any timelock. Timelock-free, non-zeno timed automata induce transition systems without terminal states such that along any path only finitely many actions are executed in finite time. In contrast to zeno paths and timelocks, time-convergent paths will be treated akin to unfair paths (in fair CTL) and are explicitly excluded for analysis purposes.

A delay of $d > 0$ time units can be realized in different ways, in general by $n > 0$ delay transitions of size d_1 through d_n with $d_i > 0$ such that $d = d_1 + \ldots + d_n$. As we are only interested in the amount of time advancing, a sequence of delay transitions labeled with d_1 through d_n and a sequence labeled with d'_1 through d'_k, say, such that $\sum_{i=1}^{n} d_i = \sum_{i=1}^{k} d'_i = d$, are considered equivalent and denoted by $\stackrel{d}{\Longrightarrow}$. This relation is used later for defining the semantics of timed CTL.

Notation 9.26. Sets of Path Fragments

Let *TA* be a timed automaton. For path fragments in $TS(TA)$ along which infinitely many actions are performed, let

$$s_0 \stackrel{d_0}{\Longrightarrow} s_1 \stackrel{d_1}{\Longrightarrow} s_2 \stackrel{d_2}{\Longrightarrow} \ldots \qquad \text{with } d_0, d_1, d_2 \ldots \geqslant 0$$

denote the equivalence class containing all infinite path fragments induced by execution fragments in $TS(TA)$ of the form

$$s_0 \underbrace{\stackrel{d_0^1}{\rightarrow} \ldots \stackrel{d_0^{k_0}}{\rightarrow}}_{\substack{\text{time passage of} \\ d_0 \text{ time-units}}} s_0 + d_0 \stackrel{\alpha_0}{\longrightarrow} s_1 \underbrace{\stackrel{d_1^1}{\rightarrow} \ldots \stackrel{d_1^{k_1}}{\rightarrow}}_{\substack{\text{time passage of} \\ d_1 \text{ time-units}}} s_1 + d_1 \stackrel{\alpha_1}{\longrightarrow} s_2 \underbrace{\stackrel{d_2^1}{\rightarrow} \ldots \stackrel{d_2^{k_2}}{\rightarrow}}_{\substack{\text{time passage of} \\ d_2 \text{ time-units}}} s_2 + d_2 \stackrel{\alpha_2}{\longrightarrow} \ldots$$

where $k_i \in \mathbb{N}$, $d_i \in \mathbb{R}_{\geqslant 0}$ and $\alpha_i \in Act$ such that $\sum_{j=1}^{k_i} d_i^j = d_i$. Note that in the $\Longrightarrow$ notations, actions are abstracted away.

For infinite path fragment $\pi \in s_0 \xRightarrow{d_0} s_1 \xRightarrow{d_1} \ldots$ that performs infinitely many actions, we have $ExecTime(\pi) = \sum_{i \geqslant 0} d_i$. Path fragments in $s_0 \xRightarrow{d_0} s_1 \xRightarrow{d_1} \ldots$ are time-divergent if and only if $\sum_i d_i$ diverges.

Time-divergent path fragments that perform finitely many actions $\alpha \in Act$ (but contain infinitely many delay transitions) are represented in a similar way, except that after the execution of the last action $\alpha \in Act$ the advance of time is represented by infinitely many $\xRightarrow{1}$ transitions. That is, the set

$$s_0 \xRightarrow{d_0} s_1 \xRightarrow{d_1} \ldots \xRightarrow{d_{n-1}} s_n \xRightarrow{1} s_n{+}1 \xRightarrow{1} s_n{+}2 \xRightarrow{1} \ldots$$

contains all infinite path fragments induced by the execution fragment of the form

$$s_0 \underbrace{\rightarrow \ldots \rightarrow}_{\substack{\text{time passage of} \\ d_0 \text{ time-units}}} \xrightarrow{\alpha_0} \ldots \xrightarrow{\alpha_{n-2}} s_{n-1} \underbrace{\rightarrow \ldots \rightarrow}_{\substack{\text{time passage of} \\ d_{n-1} \text{ time-units}}} \xrightarrow{\alpha_{n-1}} s_n \underbrace{\xrightarrow{1} s_n{+}1 \xrightarrow{1} s_n{+}2 \xrightarrow{1} \ldots}_{\text{infinite time passage}}$$

Hence, $s_0 \xRightarrow{d_0} s_1 \xRightarrow{d_1} \ldots$ is a uniform notation for all infinite time-divergent path fragments. ■

9.2 Timed Computation Tree Logic

Timed CTL (TCTL, for short) is a real-time variant of CTL aimed to express properties of timed automata. In TCTL, the until modality is equipped with a time interval such that $\Phi \,\mathsf{U}^{\,J}\, \Psi$ asserts that a Ψ-state is reached within $t \in J$ time units while only visiting Φ-states before reaching the Ψ-state. The fact that a deadlock may be reached within thirty time units via legal states only, say, can be expressed as $\mathit{legal}\,\mathsf{U}^{\,[0,30]}\,\mathit{deadlock}$, where the atomic propositions *legal* and *deadlock* have their obvious meaning. Timed CTL is sufficiently expressive to allow for the formulation of an important set of real-time system properties.

Definition 9.27. Syntax of Timed CTL

Formulae in TCTL are either state or path formulae. TCTL *state formulae* over the set AP of atomic propositions and set C of clocks are formed according to the following grammar:

$$\Phi ::= \text{true} \;\Big|\; a \;\Big|\; g \;\Big|\; \Phi \wedge \Phi \;\Big|\; \neg\Phi \;\Big|\; \exists\varphi \;\Big|\; \forall\varphi$$

where $a \in AP$, $g \in ACC(C)$ and φ is a *path formula* defined by:

$$\varphi ::= \Phi \,\mathsf{U}^{J}\, \Phi$$

where $J \subseteq \mathbb{R}_{\geqslant 0}$ is an interval whose bounds are natural numbers. ■

Timed CTL extends CTL with atomic clock constraints over the clocks in C, typically the set of clocks in the timed automaton under consideration. The propositional logic operators $\vee$, $\rightarrow$, true, etc. are obtained in the usual way. The until operator is equipped with an interval J of real numbers. Timed variants of the modal operators $\Diamond$ and $\Box$ are obtained as follows: $\Diamond^{J} \Phi \;=\; \mathrm{true}\,\mathsf{U}^{J}\,\Phi$ and

$$\exists\Box^{J}\,\Phi \;=\; \neg\forall\Diamond^{J}\,\neg\Phi \quad \text{and} \quad \forall\Box^{J}\,\Phi \;=\; \neg\exists\Diamond^{J}\,\neg\Phi.$$

The formula $\exists\Box^{J}\,\Phi$ asserts that there exists a path for which during the interval J, Φ holds; $\forall\Box^{J}\,\Phi$ requires this to hold for all paths. As we will see later on when defining the formal semantics of TCTL, the path quantifiers quantify over time-divergent paths only. Accordingly, a state in $TS(TA)$ satisfies $\forall\Diamond^{J}\Phi$ whenever all time-divergent paths starting in s satisfy $\Diamond^{J}\Phi$. The next-step operator is absent in TCTL. As the time domain is continuous there is no unique next time instant which makes it impossible to provide a suitable meaning to the next-step operator. Note that $J \subseteq \mathbb{R}_{\geqslant 0}$ has natural bounds, i.e., the interval J is either of the form $[n,m]$, $(n,m]$, $[n,m)$ or (n,m) for $n,m \in \mathbb{N}$ and $n \leqslant m$. For right-open intervals, $m = \infty$ is allowed.

In the sequel, intervals are often denoted by shorthand, e.g., $\Diamond^{\leqslant 2}$ denotes $\Diamond^{[0,2]}$ and $\Box^{>8}$ denotes $\Box^{(8,\infty)}$. For the special case $J = [0,\infty)$, the timing requirements are in fact trivially fulfilled. That is:

$$\Phi \,\mathsf{U}^{[0,\infty)}\, \Psi = \Phi \,\mathsf{U}\, \Psi \quad \text{and} \quad \Diamond\,\Phi = \Diamond^{[0,\infty)}\,\Phi \quad \text{and} \quad \Box\,\Phi = \Box^{[0,\infty)}\,\Phi.$$

The following examples illustrate the kind of timing properties that can be expressed in TCTL.

Example 9.28. Light Switch

Consider the light switch of Example 9.12 (page 688). The property

"the light cannot be continuously switched on for more than 2 minutes"

is expressed by the TCTL formula:

$$\forall\Box(\mathit{on} \longrightarrow \forall\Diamond^{>2}\,\neg\mathit{on}).$$

The property

"the light will stay on for at least 1 time unit and then switch off"

is expressed by the TCTL formula:

$$\forall\Box\Big((on \wedge (x = 0)) \longrightarrow (\forall\Box^{\leqslant 1} on \ \wedge \ \forall\Diamond^{>1} off)\Big).$$

The clock x that occurs in the formula is used to specify the time instant at which the light is switched on. ■

Example 9.29. Railroad Crossing

Consider the railroad crossing example. The safety property

"the gate is always closed when the train is at the crossing"

does not contain any timing aspects, and can be described as in CTL by the formula $\forall\Box(in \longrightarrow down)$, where *in* and *down* are locations in the timed automata *Train* and *Gate*, respectively. The (timed liveness) property

"once the train is far, within 1 minute the gate is up for at least 1 minute"

is expressed by the TCTL formula:

$$\forall\Box(far \ \longrightarrow \ \forall\Diamond^{\leqslant 1} \forall\Box^{\leqslant 1} \ up).$$

The fact that the train needs at least 2 minutes to reach the crossing after transmitting the "approach" signal is expressed by

$$\forall\Box\Big((near \wedge (y = 0)) \ \longrightarrow \ \forall\Box^{\leqslant 2} \neg in\Big)$$

where the atomic clock constraint $y{=}0$ indicates the time instant at which the train signals its approach. Finally, the property that the train needs at most five minutes to pass the crossing since its approach is expressed by:

$$\forall\Box\Big((near \wedge (y = 0)) \ \longrightarrow \ \forall\Diamond^{\leqslant 5} far\Big).$$

■

The semantics of TCTL formulae is defined for states of the form $\langle \ell, \eta \rangle$. The state formulae $\forall\varphi$ and $\exists\varphi$ are interpreted over all *time-divergent* paths. That is to say, time-convergent paths are not of any importance for the satisfaction of TCTL state formulae. This is similar to the treatment of unfair paths in the semantics of fair CTL; see Definition 6.33 on page 360.

Definition 9.30. Satisfaction Relation for TCTL

Let $TA = (Loc, Act, C, \hookrightarrow, Loc_0, Inv, AP, L)$ be a timed automaton, $a \in AP$, $g \in ACC(C)$, and $J \subseteq \mathbb{R}_{\geqslant 0}$. For state $s = \langle \ell, \eta \rangle$ in $TS(TA)$ and TCTL state formulae Φ and Ψ, and TCTL path formula φ, the satisfaction relation $\models$ is defined for state formulae by

$$
\begin{array}{lll}
s \models \text{true} & & \\
s \models a & \text{iff} & a \in L(\ell) \\
s \models g & \text{iff} & \eta \models g \\
s \models \neg \Phi & \text{iff} & \text{not } s \models \Phi \\
s \models \Phi \wedge \Psi & \text{iff} & (s \models \Phi) \text{ and } (s \models \Psi) \\
s \models \exists \varphi & \text{iff} & \pi \models \varphi \text{ for some } \pi \in Paths_{div}(s) \\
s \models \forall \varphi & \text{iff} & \pi \models \varphi \text{ for all } \pi \in Paths_{div}(s).
\end{array}
$$

For time-divergent path $\pi \in s_0 \xrightarrow{d_0} s_1 \xrightarrow{d_1} \ldots$, the satisfaction relation $\models$ for path formulae is defined by

$$
\begin{aligned}
\pi \models \Phi \,\mathsf{U}^{\,J}\, \Psi \quad \text{iff} \quad & \exists\, i \geqslant 0.\, s_i{+}d \models \Psi \text{ for some } d \in [0, d_i] \text{ with} \\
& \qquad \sum_{k=0}^{i-1} d_k + d \in J \qquad \text{and} \\
& \forall\, j \leqslant i.\, s_j{+}d' \models \Phi \vee \Psi \text{ for any } d' \in [0, d_j] \text{ with} \\
& \qquad \sum_{k=0}^{j-1} d_k + d' \;\leqslant\; \sum_{k=0}^{i-1} d_k + d
\end{aligned}
$$

where for $s_i = \langle \ell_i, \eta_i \rangle$ and $d \geqslant 0$ we have $s_i{+}d = \langle \ell_i, \eta_i{+}d \rangle$. ■

The interpretations for atomic propositions, negation, and conjunction are as usual. Clock constraint g holds in $\langle \ell, \eta \rangle$ whenever the values of the clocks in η satisfy g. State formula $\exists\varphi$ is true in state s if and only if there exists some time-divergent path starting in s that satisfies φ. $\forall\varphi$ holds in s whenever all time-divergent paths starting in s satisfy φ. As stated before, path quantification is over time-divergent paths. Truth of φ on time-convergent paths of s is irrelevant. Let us now consider the semantics of the until operator. Time-divergent path $\pi \in s_0 \xrightarrow{d_0} s_1 \xrightarrow{d_1} \ldots$ satisfies $\Phi \,\mathsf{U}^{\,J}\, \Psi$ whenever at some time point in J, a state is reached satisfying Ψ and at all previous time instants $\Phi \vee \Psi$ holds. The reader might wonder why it is not required—as in the temporal logics CTL and LTL before—that just Φ holds at all preceding time instants. This is justified by the following example.

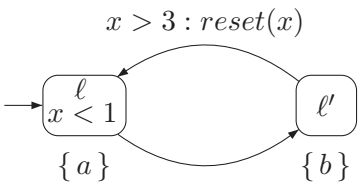


Figure 9.16: An example of a timed automaton .

Example 9.31. Semantics of Until

Consider the timed automaton *TA* in Figure 9.16 and let $\Phi = \forall(a \,\mathsf{U}^{>1} b)$. Intuitively, we expect $TA \models \Phi$, since $Inv(\ell) = x < 1$, and location ℓ' cannot be left before three time units have elapsed. At time instant $t = 1.5$, e.g., *TA* always resides in location ℓ'. Consider now the path

$$\pi \;=\; \langle \ell, 0\rangle \, \langle \ell, 0.5\rangle \, \langle \ell', 0.5\rangle \, \langle \ell', 3\rangle \, \langle \ell, 0\rangle \ldots \;.$$

It follows that:

$$\pi \in \underbrace{\langle \ell, 0\rangle}_{s_0} \xRightarrow{0.5} \underbrace{\langle \ell', 0.5\rangle}_{s_1} \xRightarrow{2.5} \underbrace{\langle \ell, 0\rangle}_{s_2} \ldots .$$

According to the TCTL semantics, $\pi \models a \,\mathsf{U}^{>1} b$ since

$$s_1{+}d \models b \text{ for some } d \in [0, 2.5] \text{ such that } 0.5{+}d > 1$$

and

$$s_0{+}d' \models a \text{ for all } d' \in [0, 0.5] \quad \text{and} \quad s_1{+}d' \models a \vee b \text{ for all } d' \in [0, d].$$

A semantics along the lines of CTL would require $s_1{+}d' \models a$ for all $d' \in [0, d]$ with $d > 0.5$. The event $\neg a \wedge b$, however, occurs at a time instant that is *before* the required time bound $]1, \infty)$ is reached. As a result, the statement $s_1{+}d' \models a$ for all $d' \in [0, d]$ with $d > 0.5$ does not hold.

Note that in CTL (and LTL) it holds that $\Phi \,\mathsf{U}\, \Psi$ is equivalent to $(\Phi \vee \Psi) \,\mathsf{U}\, \Psi$. ■

From the TCTLsemantics it follows for time-divergent path $\pi \in s_0 \xRightarrow{d_0} s_1 \xRightarrow{d_1} \ldots$ that:

$$\pi \models \Diamond^J \Phi \;\text{ iff }\; \exists\, i \geqslant 0.\, s_i{+}d \models \Phi \text{ for some } d \in [0, d_i] \text{ with } \sum_{k=0}^{i-1} d_k + d \in J.$$

As expected, a time-divergent path fragment satisfies $\Diamond^J \Phi$ whenever a Φ-state is reached at some time instant $t \in J$. For the $\Box^J$ operator we obtain that

$$\pi \models \;\Box^J \Phi \;\text{ iff }\; \forall i \geqslant 0.\, s_i{+}d \models \Phi \text{ for any } d \in [0, d_i] \text{ with } \sum_{k=0}^{i-1} d_k + d \in J.$$

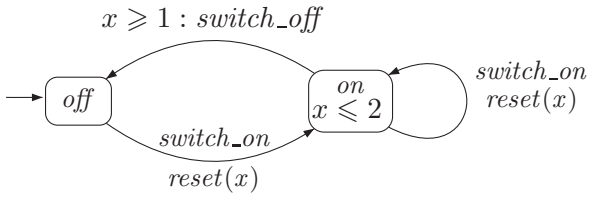


Figure 9.17: Timed automaton $Switch_3$ (again).

Thus, $\pi \models \Box^J \Phi$ iff all states visited by π in the time interval J satisfy Φ.

Timed automaton *TA* satisfies TCTL state formula Φ whenever all its initial states satisfy Φ.

Definition 9.32. TCTL Semantics for Timed Automata

Let *TA* be a timed automaton with clocks C and locations *Loc*. For TCTL state formula Φ, the *satisfaction set* $Sat(\Phi)$ is defined by:

$$Sat(\Phi) \;=\; \{\, s \in Loc \times Eval(C) \mid s \models \Phi \,\}.$$

The timed automaton *TA* satisfies TCTL state formula Φ if and only if Φ holds in all initial states of *TA*:

$$TA \models \Phi \quad \text{if and only if} \quad \forall \ell_0 \in Loc_0 .\, \langle \ell_0, \eta_0 \rangle \models \Phi$$

where $\eta_0(x) = 0$ for all $x \in C$. ■

Example 9.33. TCTL Semantics

Consider the timed automaton $Switch_3$ depicted in Figure 9.17.

For the sake of convenience we only consider the reachable states in $TS(Switch_3)$. We then have:

$$\begin{aligned}
Sat(\forall\Diamond^{<1} off) &= \{\, \langle off, t\rangle \mid t \geqslant 0 \,\} \cup \{\, \langle on, t\rangle \mid 1 < t \leqslant 2 \,\} \\
Sat(\exists\Diamond^{<1} off) &= \{\, \langle off, t\rangle \mid t \geqslant 0 \,\} \cup \{\, \langle on, t\rangle \mid 0 < t \leqslant 2 \,\} \\
Sat(\forall\Diamond(on \wedge (x = 1))) &= \{\, \langle on, t\rangle \mid 0 \leqslant t \leqslant 1 \,\} \\
Sat(\forall\Diamond(on \wedge (x = 0))) &= \{\, \langle on, 0\rangle \,\} \\
Sat(\forall\Diamond(on \wedge (x \geqslant 3))) &= \varnothing
\end{aligned}$$

Since $Switch_3 \models \Phi$ if and only if the initial state $\langle off, 0\rangle$ of $TS(Switch_3)$ is in $Sat(\Phi)$, we have

$$Switch_3 \models \forall\Diamond^{<1} off \quad \text{and} \quad Switch_3 \models \exists\Diamond^{<1} off \quad \text{and} \quad Switch_3 \not\models \forall\Diamond(on \wedge (x \in J))$$

for any interval $J \subseteq \mathbb{R}_{\geqslant 0}$. Consider

$$\Phi \;=\; \forall\Box \Big(\; (on \wedge (x=0)) \;\longrightarrow\; \forall\Diamond(on \wedge (x=1)) \; \Big).$$

It follows that $Switch_3 \models \Phi$. It is, however, essential that universal path quantification only consider time-divergent paths. For example, the time-convergent path

$$\langle off, 0\rangle \, \langle on, 0\rangle \, \langle on, \frac{1}{2}\rangle \, \langle on, \frac{3}{4}\rangle \, \langle on, \frac{7}{8}\rangle \, \langle on, \frac{15}{16}\rangle \, \ldots$$

does not visit a state satisfying $on \wedge (x = 1)$.

Note that a (time-divergent) path does not have to explicitly visit a state satisfying $on \wedge (x = 1)$ to satisfy Φ, e.g., the path $\langle off, 0\rangle \, \langle on, 0\rangle \, \langle on, 2\rangle \, \langle off, 2\rangle \, \langle on, 0\rangle, \ldots$ satisfies $\Diamond on \wedge (x = 1)$ although a state with $x{=}1$ does not appear in its representation. The state $\langle on, 1\rangle$ is passed during the delay transition $\langle on, 0\rangle \xrightarrow{2} \langle on, 2\rangle$.

Finally, note that $Switch_3 \not\models \forall\Diamond on$, as $Inv(off) = \text{true}$, i.e., the timed automaton has a time-divergent path that resides in location *off* for ever. ■

Remark 9.34. TCTL vs. CTL

Any TCTL formula Φ in which all intervals are of the form $[0, \infty)$ may be considered as a CTL formula over the set of propositions AP and the atomic clock constraints in Φ. Due to time-divergent paths, though, there is a subtle difference between the interpretation of TCTL and CTL formulae. As a result, $TS(TA) \models_{\text{TCTL}} \forall\varphi$ and $TS(TA) \not\models_{\text{CTL}} \forall\varphi$ is possible: whereas $\models_{\text{TCTL}}$ ranges over all time-divergent paths, $\models_{\text{CTL}}$ considers all paths, in particular also the time-convergent paths. Consider, e.g., the light switch described in Example 9.33 and the TCTL formula:

$$\Phi \;=\; \forall\Box(on \;\longrightarrow\; \forall\Diamond off).$$

It follows that

$$\underbrace{TS(TA) \models_{\text{TCTL}} \Phi}_{\text{TCTL semantics}} \quad \text{and} \quad \underbrace{TS(TA) \not\models_{\text{CTL}} \Phi}_{\text{CTL semantics}}.$$

The fact that $TS(TA) \not\models_{\text{CTL}} \Phi$ results from the existence of time-convergent paths in which the location *on* is never left. ■

The semantics for TCTL is well-defined for timed automata that contain a timelock. Recall that timed automaton TA contains a timelock whenever there exists a state in $TS(TA)$ from which no time-divergent path emanates. A state is timelock-free if and only if it

satisfies $\exists\Box$true. The formula $\exists\Box$true holds in state s whenever some time-divergent path satisfies $\Box$true, i.e., whenever there is at least one time-divergent path. Note that for fair CTL, the states in which a fair path starts are also characterized by the formula $\exists\Box$true. This yields the following characterization of timelock freeness:

Lemma 9.35. Characterizing Timelock

Timed automaton TA is timelock-free *iff* $\forall s \in \mathit{Reach}(\mathit{TS}(\mathit{TA}))\,.\, s \models_{\mathit{TCTL}} \exists\Box \mathit{true}$.

$\mathit{TA} \models \forall\Box\Phi$ if and only if all reachable states on all time-divergent paths satisfy Φ. In general, from $\mathit{TA} \models \forall\Box\Phi$ it may not be concluded that all reachable states in $\mathit{TS}(\mathit{TA})$ satisfy Φ. This holds for timelock-free timed automata, but not for others. In particular, the TCTL formula

$$\forall\Box\exists\Box\text{true}$$

is a tautology in TCTL (and not characteristic of timelock-free timed automata).

9.3 TCTL Model Checking

The TCTL model-checking problem is to check for a given timed automaton TA and TCTL formula Φ whether $\mathit{TA} \models \Phi$. It is assumed that TA is non-zeno. The possible presence of timelocks is not relevant, as timelock freedom can be checked by a TCTL formula, see Remark 9.35. The main difficulty of the TCTL model-checking problem is that a transition system with uncountably many states has to be analyzed, since

$$\underbrace{\mathit{TA} \models \Phi}_{\text{timed automaton}} \quad \text{iff} \quad \underbrace{\mathit{TS}(\mathit{TA}) \models \Phi}_{\text{infinite transition system}} \ .$$

A naive graph analysis in the state graph of $\mathit{TS}(\mathit{TA})$ is therefore not feasible. Instead, the basic idea is to consider a finite quotient of this transition system, the so-called *region transition system*, which is obtained from the timed automaton TA and the TCTL formula Φ.[1] In essence, the region transition system $\mathit{RTS}(\mathit{TA}, \Phi)$ is the quotient of $\mathit{TS}(\mathit{TA})$ with respect to a bisimulation relation. The states in the region transition system are equivalence classes of states in $\mathit{TS}(\mathit{TA})$ that all satisfy the same atomic clock constraints, and from which "similar" time-divergent paths emanate, i.e., such states are TCTL equivalent. As the number of equivalence classes is finite, this provides a basis for TCTL model checking. In fact, rather than checking the TCTL formula Φ, it is checked whether a derived CTL formula holds in $\mathit{RTS}(\mathit{TA}, \Phi)$.

[1] In fact, the region transition system depends on the maximal constants with which clocks are compared in TA and the maximal timing constants in Φ.

To check whether a timed automaton satisfies a TCTL formula thus amounts to model-check its region transition system against a corresponding CTL formula. For the latter, traditional CTL model-checking algorithms can be exploited. Summarizing:

$$TA \models_{\text{TCTL}} \Phi \quad \text{iff} \quad \underbrace{RTS(TA,\Phi)}_{\text{finite transition system}} \models_{\text{CTL}} \widehat{\Phi}$$

where $\widehat{\Phi}$ is a CTL formula that is obtained from the TCTL formula Φ using the translation explained next. In summary, we obtain the scheme in Algorithm 43 where $\cong$ denotes the equivalence used to obtain the quotient $RTS(TA,\Phi)$.

Algorithm 43 Basic recipe of TCTL model checking

Input: timed automaton TA and TCTL formula Φ (both over AP and C)
Output: $TA \models \Phi$

$\widehat{\Phi}$:= eliminate the timing parameters from Φ;

determine the equivalence classes under $\cong$;

construct the region transition system $TS = RTS(TA)$;

apply the CTL model-checking algorithm to check $TS \models \widehat{\Phi}$;

$TA \models \Phi$ if and only if $TS \models \widehat{\Phi}$.

9.3.1 Eliminating Timing Parameters

We first explain how intervals $J \neq [0,\infty)$ that may appear in TCTL formulae as time bounds for path formulae are replaced by equivalent atomic clock constraints. Let $\text{TCTL}_{\Diamond}$ denote the set of TCTL formulae in which all intervals J are equal to $[0,\infty)$. That is, the only timing aspects that occur in $\text{TCTL}_{\Diamond}$ formulae are atomic clock constraints. As such constraints can be considered as atomic propositions, in fact, $\text{TCTL}_{\Diamond}$ is a subset of CTL. The resulting formulae provide the basis for the CTL formulae that are checked on the region transition system.

The basic idea of eliminating $J \neq [0,\infty)$ from TCTL formula Φ is to introduce a fresh clock, z, say, that neither occurs in Φ nor in the timed automaton under investigation,

and to enrich the formula Φ with atomic clock constraints that may refer to z. The clock z is used to measure the elapse of time until a certain property, i.e., sub formula of Φ, holds. For instance, to check the TCTL formula $\exists\Diamond^J\Phi$ in state s, clock z is reset in state s and Φ is checked whenever the current value of clock z lies in the interval J. In order to formalize this idea, the following auxiliary notations are helpful.

Notation 9.36. Clock Evaluation $\eta\{\ldots\}$

For clock evaluation $\eta \in \mathit{Eval}(C)$, $z \notin C$ and $d \in \mathbb{R}_{\geqslant 0}$, let $\eta\{z := d\}$ denote the clock valuation for $C \cup \{ z \}$ that extends η by setting z to d while keeping the value of all other clocks unchanged:

$$\eta\{z := d\}(x) \;=\; \begin{cases} \eta(x) & \text{if } x \in C \\ d & \text{if } x = z. \end{cases}$$

Let TA be a timed automaton over C. For state $s = \langle \ell, \eta \rangle$ in $TS(TA)$ let $s\{z := d\}$ denote the state $\langle \ell, \eta\{z := d\} \rangle$. Note that $s\{z := d\}$ is a state in $TS(TA \oplus z)$ where $TA \oplus z$ is the timed automaton TA with the set of clocks $C \cup \{ z \}$. ■

The following theorem provides a recipe to transform any TCTL formula into a timing parameter-free TCTL formula.

Theorem 9.37. Elimination of Timing Parameters

Let TA be timed automaton $(Loc, Act, C, \hookrightarrow, Loc_0, Inv, AP, L)$, and $\Phi \,\mathsf{U}^{\,J}\, \Psi$ a TCTL formula over C and AP. For clock $z \notin C$, let

$$TA \oplus z \;=\; (Loc, Act, C \cup \{ z \}, \hookrightarrow, Loc_0, Inv, AP, L).$$

For any state s of $TS(TA)$ it holds that

1. $s \models_{TCTL} \exists(\Phi \,\mathsf{U}^{\,J} \Psi)$ *iff* $\underbrace{s\{z := 0\}}_{\text{state in } TS(TA \oplus z)} \models_{TCTL} \exists\Big((\Phi \vee \Psi) \,\mathsf{U}\, ((z \in J) \wedge \Psi)\Big)$.

2. $s \models_{TCTL} \forall(\Phi \,\mathsf{U}^{\,J} \Psi)$ *iff* $\underbrace{s\{z := 0\}}_{\text{state in } TS(TA \oplus z)} \models_{TCTL} \forall\Big((\Phi \vee \Psi) \,\mathsf{U}\, ((z \in J) \wedge \Psi)\Big)$.

Proof: Since $TA \oplus z$ just extends TA with a fresh clock z (which is not used in TA), it follows that any path π in $TS(TA)$ uniquely corresponds to a path π' in $TS(TA \oplus z)$ such that

$$\pi \in s_0 \xRightarrow{d_0} s_1 \xRightarrow{d_1} s_2 \xRightarrow{d_2}$$

if and only if

$$\pi' \in s_0\{z := 0\} \stackrel{d_0}{\Longrightarrow} s_1\{z := d_0\} \stackrel{d_1}{\Longrightarrow} s_2\{z := d_0{+}d_1\} \stackrel{d_2}{\Longrightarrow} \ldots$$

It is easy to see that π is time-divergent if and only if π' is time-divergent. We now prove that $\pi \models \Phi \,\mathsf{U}^{\,J}\, \Psi$ iff $\pi' \models (\Phi \vee \Psi) \,\mathsf{U}\, ((z \in J) \wedge \Psi)$. We only consider the direction $\Rightarrow$; the proof for the other direction is similar. Assume $\pi \models \Phi \,\mathsf{U}^{\,J}\, \Psi$. From the TCTL semantics this is equivalent to

$$\begin{aligned} &\exists i \geqslant 0.\, s_i{+}d \models \Psi \text{ for some } d \in [0, d_i] \text{ with } \textstyle\sum_{k=0}^{i-1} d_k + d \in J \text{ and} \\ &\forall j \leqslant i.\, s_j{+}d' \models \Phi \vee \Psi \text{ for any } d' \in [0, d_j] \text{ with } \textstyle\sum_{k=0}^{j-1} d_k + d' \leqslant \sum_{k=0}^{i-1} d_k + d. \end{aligned}$$

As z is a fresh clock, this is equivalent to

$$\begin{aligned} &\exists i \geqslant 0.\, s'_i{+}d \models \Psi \text{ for some } d \in [0, d_i] \text{ with } \textstyle\sum_{k=0}^{i-1} d_k + d \in J \text{ and} \\ &\forall j \leqslant i.\, s'_j{+}d' \models \Phi \vee \Psi \text{ for any } d' \in [0, d_j] \text{ with } \textstyle\sum_{k=0}^{j-1} d_k + d' \leqslant \sum_{k=0}^{i-1} d_k + d. \end{aligned}$$

where $s'_i = s_i\{z := \sum_{k=0}^{i-1} d_k\}$ and $s'_j = s_j\{z := \sum_{k=0}^{j-1} d_k\}$. As clock z is never reset, the value of z in state $s'_i{+}d$ equals $\sum_{k=0}^{i-1} d_k + d$. As this sum lies in J, in the first conjunct Ψ may be strengthened by the atomic clock constraint $z \in J$. This yields

$$\begin{aligned} &\exists i \geqslant 0.\, s'_i{+}d \models (z \in J) \wedge \Psi \text{ for some } d \in [0, d_i] \text{ with } \underbrace{\sum_{k=0}^{i-1} d_k + d}_{=z} \in J \text{ and} \\ &\forall j \leqslant i.\, s'_j{+}d' \models \Phi \vee \Psi \text{ for any } d' \in [0, d_j] \text{ with } \textstyle\sum_{k=0}^{j-1} d_k + d' \leqslant \underbrace{\sum_{k=0}^{i-1} d_k + d}_{=z}. \end{aligned}$$

The constraint $\sum_{k=0}^{i-1} d_k + d \in J$ can now be omitted (as it is equivalent to $z \in J$), whereas in the second part, we may weaken $\Phi \vee \Psi$ into $(\Phi \vee \Psi) \vee (\Psi \wedge (z \in J))$ as for $d' = d$ and $i{=}j$, $\Psi \wedge (z \in J)$ holds. Applying the TCTL semantics yields that $\pi' \models (\Phi \vee \Psi) \,\mathsf{U}\, ((z \in J) \wedge \Psi)$. ■

Example 9.38. Eliminating Timing Parameters

Let Φ be a TCTL formula. According to the above mapping, the TCTL formula $\exists\Diamond^{\leqslant 2}\,\Phi$ is replaced by $\exists\Diamond((z \leqslant 2) \wedge \Phi)$. In a similar way, we replace: $\exists\Box^{\leqslant 2}\,\Phi \;=\; \neg\forall\Diamond^{\leqslant 2}\,\neg\Phi$ by

$$\begin{aligned} \neg\forall\Diamond\,((z \leqslant 2) \wedge \neg\Phi) \;&\equiv\; \exists\Box\,(\neg(z \leqslant 2) \vee \Phi) \\ &=\; \exists\Box\,((z \leqslant 2) \rightarrow \Phi)\,. \end{aligned}$$

Note that the resulting formulae are CTL formulae (or could be understood as such) provided Φ does not contain intervals different from $[0, \infty)$. ■

In order to verify whether $TA \models \Phi$ for TCTL formula Φ, the above result suggests equipping TA with a clock for each subformula of Φ of the form $\Psi \,\mathsf{U}^{J}\, \Psi'$ while replacing this subformula as indicated in Theorem 9.37. This yields $\mathrm{TCTL}_{\Diamond}$ formula $\widehat{\Phi}$. As $\widehat{\Phi}$ does not contain timing parameters, and any clock constraint can be considered as an atomic proposition, in fact, $\widehat{\Phi}$ is a CTL formula! Verifying a timed CTL formula on a timed automaton TA thus reduces to checking a CTL formula on a TA extended with a clock whose sole purpose is to measure the elapse of time that is referred to in the formula.

9.3.2 Region Transition Systems

Consider timed automaton TA and $\mathrm{TCTL}_{\Diamond}$ formula Φ. It is assumed that TA is equipped with an additional clock as explained in the previous section. The idea is impose an appropriate equivalence, denoted $\cong$, on the clock valuations—and implicitly on the states of $TS(TA)$ by letting $\langle \ell', \eta' \rangle \cong \langle \ell, \eta' \rangle$ if $\ell = \ell'$ and $\eta \cong \eta'$—such that:

(A) Equivalent clock valuations should satisfy the same clock constraints that occur in TA and Φ:

$$\eta \cong \eta' \;\Rightarrow\; \big(\eta \models g \quad \text{iff} \quad \eta' \models g \text{ for all } g \in ACC(TA) \cup ACC(\Phi)\big)$$

where $ACC(TA)$ and $ACC(\Phi)$ denote the set of atomic clock constraints that occur in TA and Φ, respectively. These constraints are either of the form $x \leqslant c$ or $x < c$.

(B) Time-divergent paths emanating from equivalent states should be "equivalent". This property guarantees that equivalent states satisfy the same path formulae.

(C) The number of equivalence classes under $\cong$ is finite.

In the sequel we adopt the following notation for clock values.

Notation 9.39. Integral and Fractional Part of Real Numbers

Let $d \in \mathbb{R}$. The *integral part* of d is the largest integer that is at most d:

$$\lfloor d \rfloor \;=\; \max\{\, c \in \mathbb{N} \mid c \leqslant d \,\}.$$

The *fractional part* of d is defined by $frac(d) \;=\; d - \lfloor d \rfloor$. For example, $\lfloor 17.59267 \rfloor = 17$, $frac(17.59267) = 0.59267$, $\lfloor 85 \rfloor = 85$, and $frac(85) = 0$. ■

The definition of clock equivalence is based on three observations that successively lead to a refined notion of equivalence. Let us discuss these observations in detail.

First observation. Consider atomic clock constraint g, and let η be a clock valuation (both over the set C of clocks with $x \in C$). As g is an atomic clock constraint, g is either of the form $x < c$ or $x \leqslant c$ for $c \in \mathbb{N}$. We have that $\eta \models x < c$ whenever $\eta(x) < c$, or equivalently, $\lfloor \eta(x) \rfloor < c$. The fractional part of $\eta(x)$ in this case is not relevant. Similarly, $\eta \models x \leqslant c$ whenever either $\lfloor \eta(x) \rfloor < c$, or $\lfloor \eta(x) \rfloor = c$ and $frac(x) = 0$. Therefore, $\eta \models g$ only depends on the integral part $\lfloor \eta(x) \rfloor$, and the fact whether $frac(\eta(x)) = 0$. This leads to the initial suggestion that clock valuations η and η' are equivalent (denoted $\cong_1$) whenever

$$\lfloor \eta(x) \rfloor \;=\; \lfloor \eta'(x) \rfloor \quad \text{and} \quad frac(\eta(x)) = 0 \text{ iff } frac(\eta'(x)) = 0. \tag{9.1}$$

This constraint ensures that equivalent clock valuations satisfy the clock constraint g provided g only contains atomic clock constraints of the form $x < c$ or $x \leqslant c$. (In case one would restrict all atomic clock constraints to be strict, i.e., of the form $x < c$, the fractional parts would not be of importance and the second conjunct in the above equation may be omitted.) Note that it is crucial for this observation that only *natural number* constants are permitted in the clock constraints. This equivalence notion is rather simple, leads to a denumerable (but still infinite) number of equivalence classes, but is too coarse.

Example 9.40. A First Partitioning for Two Clocks

To exemplify the kind of equivalence classes that one obtains, consider the set of clocks $C = \{ x, y \}$. The quotient space for C obtained by suggestion (9.1) is depicted in Figure 9.18) where the equivalence classes are

- the corner points (q, p)
- the line segments $\{ (q, y) \mid p < y < p{+}1 \}$ and $\{ (x, p) \mid q < x < q{+}1 \}$, and
- the content of the squares $\{ (x, y) \mid q < x < q{+}1 \ \wedge \ p < y < p{+}1 \}$

where $p, q \in \mathbb{N}$ and $\{ (x, p) \mid q < x < q{+}1 \}$ is a shorthand for the set of all clock evaluations η with $\eta(x) \in]q, q{+}1[$ and $\eta(y) = p$. ■

Second observation. We demonstrate the fact that $\cong_1$ is too coarse by means of a small example. Consider location ℓ whose two outgoing transitions are guarded with $x \geqslant 2$ (action α) and $y > 1$ (action β), respectively; see also Figure 9.19. Let state $s = \langle \ell, \eta \rangle$ with $1 < \eta(x) < 2$ and $0 < \eta(y) < 1$. Both transitions are disabled, so the only possibility is to let time advance. The transition that is enabled next depends on the ordering of the

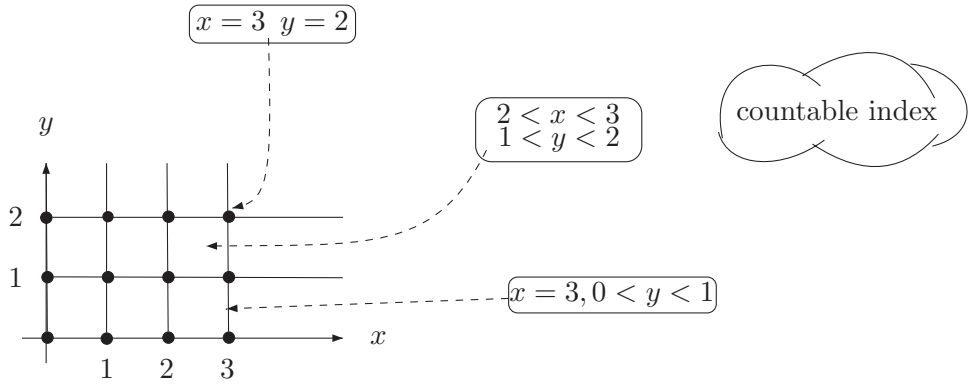


Figure 9.18: Initial partitioning for two clocks .

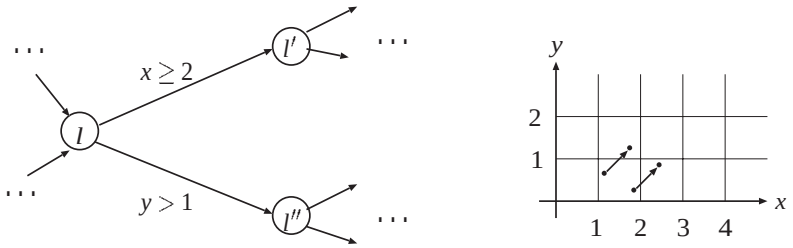


Figure 9.19: Fragment of timed automaton and time passage of two clock valuations.

fractional parts of the clocks x and y: if $frac(\eta(x)) < frac(\eta(y))$, then β is enabled before α; if $frac(\eta(x)) \geqslant frac(\eta(y))$, action α is enabled first. Time-divergent paths in s may thus start with α if $frac(\eta(x)) \geqslant frac(\eta(y))$, and with β otherwise. This is represented by the fact that delaying leads to distinct successor classes depending on the ordering of the fractional parts of clock, see Figure 9.19 (right part).

Thus, besides $\lfloor \eta(x) \rfloor$ and the fact whether $frac(\eta(x)) = 0$, apparently the *order of the fractional parts* of $\eta(x)$, $x \in C$ is important as well, i.e., whether for $x, y \in C$:

$$frac(\eta(x)) < frac(\eta(y)) \text{ or } frac(\eta(x)) > frac(\eta(y)) \text{ or } frac(\eta(x)) = frac(\eta(y)).$$

This suggests extending the initial proposal (9.1) for all $x, y \in C$ by

$$frac(\eta(x)) \leqslant frac(\eta(y)) \quad \text{if and only if} \quad frac(\eta'(x)) \leqslant frac(\eta'(y)), \tag{9.2}$$

i.e., $\eta_1 \cong_2 \eta_2$ iff $\eta_1 \cong_1 \eta_2$ and (9.2) holds. This strengthening will ensure that equivalent states $\langle \ell, \eta \rangle$ and $\langle \ell, \eta' \rangle$ have similar time-divergent paths.

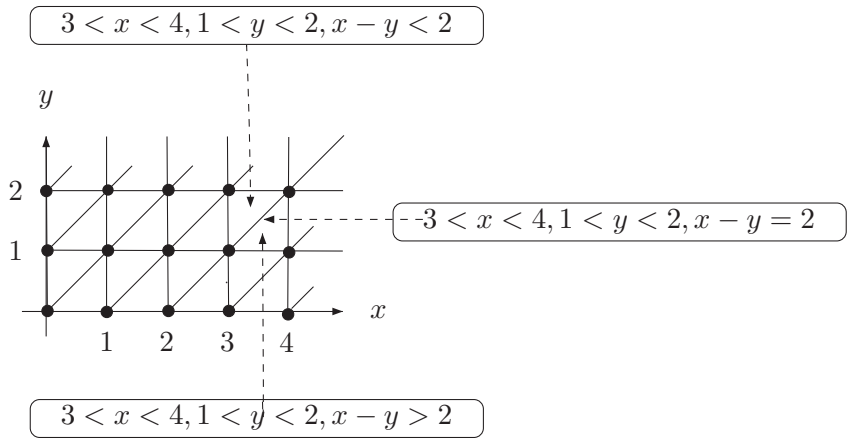


Figure 9.20: Refining the initial partitioning for two clocks.

Example 9.41. A Second Partitioning for Two Clocks

This observation suggests to decompose the squares $\{\,(x,y) \mid q < x < q{+}1 \ \wedge\ p < y < p{+}1\,\}$ into a line segment, an upper and lower triangle, i.e., the following three parts:

$$\{\,(x,y) \mid q < x < q{+}1 \ \wedge\ p < y < p{+}1 \ \wedge\ x{-}y < q{-}p\,\},$$
$$\{\,(x,y) \mid q < x < q{+}1 \ \wedge\ p < y < p{+}1 \ \wedge\ x{-}y > q{-}p\,\}, \text{and}$$
$$\{\,(x,y) \mid q < x < q{+}1 \ \wedge\ p < y < p{+}1 \ \wedge\ x{-}y = q{-}p\,\}.$$

Figure 9.20 illustrates the resulting partitioning for two clocks. ■

Final observation. The above constraints on clock equivalence yield a denumerable though not finite quotient. To obtain an equivalence with a *finite* quotient, we exploit the fact that in order to decide whether $TA \models \Phi$ only the clock constraints occurring in TA and Φ are relevant. As there are only finitely many clock constraints, we can determine for each clock $x \in C$ the maximal clock constraint, $c_x \in \mathbb{N}$, say, with which x is compared in some clock constraint in either TA (as guard or location invariant) or Φ. Since c_x is the largest constant with which clock x is compared it follows that if $\eta(x) > c_x$, the actual value of x is irrelevant. (Clock x that occurs neither in TA nor in Φ is superfluous and can be omitted; for these clocks we set $c_x = 0$.) As a consequence, the constraints (9.1) are only relevant if $\eta(x) \leqslant c_x$ and $\eta'(x) \leqslant c_x$, while for (9.2) in addition $\eta(y) \leqslant c_y$ and $\eta'(y) \leqslant c_y$.

The above considerations suggest the following notion of clock equivalence.

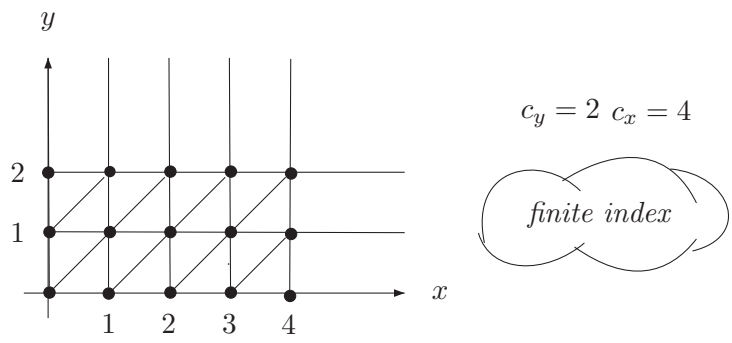


Figure 9.21: Third (and final) partitioning for two clocks (for $c_x = 4$ and $c_y = 2$).

Definition 9.42. Clock Equivalence $\cong$

Let TA be a timed automaton, Φ a TCTL$_\Diamond$ formula (both over set C of clocks), and c_x the largest constant with which $x \in C$ is compared with in either TA or Φ. Clock valuations $\eta, \eta' \in \mathit{Eval}(C)$ are *clock-equivalent*, denoted $\eta \cong \eta'$ if and only if either

- for any $x \in C$ it holds that $\eta(x) > c_x$ and $\eta'(x) > c_x$, or
- for any $x, y \in C$ with $\eta(x), \eta'(x) \leqslant c_x$ and $\eta(y), \eta'(y) \leqslant c_y$ all the following conditions hold:
  - $\lfloor \eta(x) \rfloor = \lfloor \eta'(x) \rfloor$ and $\mathit{frac}(\eta(x)) = 0$ iff $\mathit{frac}(\eta'(x)) = 0$,
  - $\mathit{frac}(\eta(x)) \leqslant \mathit{frac}(\eta(y))$ iff $\mathit{frac}(\eta'(x)) \leqslant \mathit{frac}(\eta'(y))$.

■

As the clock equivalence $\cong$ depends on TA and Φ, strictly speaking one should write $\cong_{TA,\Phi}$ instead of $\cong$. The dependency of $\cong$ on TA and Φ is limited to the largest constants c_x; that is to say, neither the structure of TA nor that of Φ is of relevance to clock equivalence. The equivalence $\cong$ is lifted to states of the transition system $TS(TA)$ as follows. For states $s_i = \langle \ell_i, \eta_i \rangle$, $i = 1, 2$, in $TS(TA)$:

$$s_1 \cong s_2 \quad \text{iff} \quad \ell_1 = \ell_2 \quad \text{and} \quad \eta_1 \cong \eta_2.$$

Equivalence classes under $\cong$ are called *clock regions*.

Definition 9.43. Clock and State Region

Let $\cong$ be a clock equivalence on C. The *clock region* of $\eta \in \mathit{Eval}(C)$, denoted $[\eta]$, is defined by

$$[\eta] \;=\; \{\, \eta' \in \mathit{Eval}(C) \mid \eta \cong \eta' \,\}.$$

The *state region* of $s = \langle \ell, \eta \rangle \in \mathit{TS}(\mathit{TA})$, denoted $[s]$, is defined by

$$[s] \;=\; \langle \ell, [\eta] \rangle \;=\; \{\, \langle \ell, \eta' \rangle \mid \eta' \in [\eta] \,\}.$$

■

In the sequel, state and clock regions are often indicated as regions whenever it is clear from the context what is meant. Clock regions will be denoted by r, r', and so forth. We often use casual notations to denote clock regions or clock valuations. For a timed automaton with two clocks, x and y say,

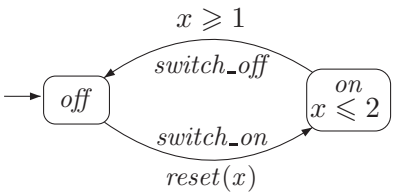
$$\{(x, y) \mid 1 < x < 2, 0 < y < 1, x - y < 1\}$$

denotes the clock region of all clock valuations $\eta \in \mathit{Eval}(\{\, x, y \,\})$ with

$$1 < \eta(x) < 2 \quad \text{and} \quad 0 < \eta(y) < 1 \quad \text{and} \quad \mathit{frac}(\eta(x)) < \mathit{frac}(\eta(y)).$$

Example 9.44. Light Switch

Consider the timed automaton over $C = \{\, x \,\}$ for the light switch and the $\mathrm{TCTL}_{\Diamond}$ formula $\Phi = \mathrm{true}$. It follows that the largest constant with which x is compared is $c_x = 2$; this is due to the location invariant $x \leqslant 2$.



We gradually construct the regions for this timed automaton by considering each of the constraints in Definition 9.42 separately. Clock valuations η, η' are equivalent if $\eta(x)$ and $\eta'(x)$ belong to the same equivalence class along the real line. (In general, for n clocks this amounts to considering an n-dimensional hyperspace on $\mathbb{R}_{\geqslant 0}$.)

1. The requirement that $\eta(x) > 2$ and $\eta'(x) > 2$ or $\eta(x) \leqslant 2$ and $\eta'(x) \leqslant 2$ yields the partitioning into the intervals $[0, 2]$ and $(2, \infty)$.

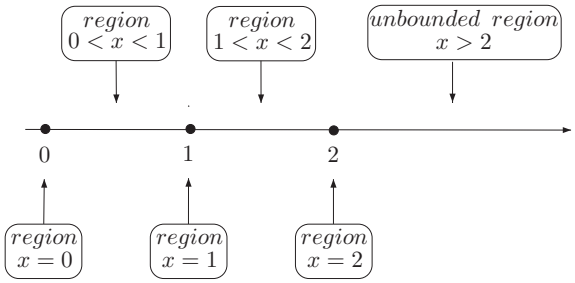


Figure 9.22: Clock regions for the light switch timed automaton.

2. The requirement that whenever $\eta(x) \leqslant 2$ and $\eta'(x) \leqslant 2$, the integral parts of $\eta(x)$ and $\eta'(x)$ agree and $\mathit{frac}(\eta(x)) = 0$ iff $\mathit{frac}(\eta'(x)) = 0$ yields the partitioning into the intervals
$$[0,0], (0,1), [1,1], (1,2), [2], (2,\infty).$$

3. As there is only a single clock, the third constraint of Definition 9.42 trivially holds.

We thus obtain six clock regions (see Figure 9.22), and as there are two locations, twelve state regions. ■

Example 9.45. Two Clocks

Consider the set of clocks $C = \{ x, y \}$ and assume $c_x = 2$ and $c_y = 1$. As in the previous example, we gradually construct the clock regions. Clock valuations $\eta, \eta' \in \mathit{Eval}(\{ x, y \})$ are equivalent if the real-valued pairs $(\eta(x), \eta(y))$ and $(\eta'(x), \eta'(y))$ are elements of the same clock region.

1. The requirement that $\eta(x) > c_x$ and $\eta'(x) > c_x$ or $\eta(x) \leqslant c_x$ and $\eta'(x) \leqslant c_x$ for any clock $x \in C$ yields four classes:
$$\{ (x,y) \mid 0 \leqslant x \leqslant 2, 0 \leqslant y \leqslant 1 \}, \quad \{ (x,y) \mid 0 \leqslant x \leqslant 2, y > 1 \},$$
$$\{ (x,y) \mid x > 2, 0 \leqslant y \leqslant 1 \}, \quad \text{and } \{ (x,y) \mid x > 2, y > 1 \}$$

2. The second requirement of Definition 9.42 yields a refinement of the first three classes obtained in the previous step. For instance, the rectangle $[(0 \leqslant x \leqslant 2), (0 \leqslant y \leqslant 1)]$ is decomposed into the six corner points:
$$(0,0), (0,1), (1,0), (1,1), (2,0) \text{ and, } (2,1),$$

the (open) line segments:

$$\{\,(0,y) \mid 0 < y < 1\,\},\quad \{\,(1,y) \mid 0 < y < 1\,\},\quad \{\,(2,y) \mid 0 < y < 1\,\},$$
$$\{\,(x,0) \mid 0 < x < 1\,\},\quad \{\,(x,0) \mid 1 < x < 2\,\},$$
$$\{\,(x,1) \mid 0 < x < 1\,\},\quad \{\,(x,1) \mid 1 < x < 2\,\},$$

and the (open) squares:

$$\{\,(x,y) \mid 0 < x < 1, 0 < y < 1\,\} \quad \text{and} \quad \{\,(x,y) \mid 1 < x < 2, 0 < y < 1\,\}.$$

Similarly, $[(0 \leqslant x \leqslant 2), (y > 1)]$ is decomposed into

$$\begin{array}{lll} \{\,(0,y) \mid y > 1\,\} & \{\,(1,y) \mid y > 1\,\} & \{\,(2,y) \mid y > 1\,\} \\ \{\,(x,y) \mid 0 < x < 1, y > 1\,\} & \{\,(x,y) \mid 1 < x < 2, y > 1\,\} & \end{array}$$

In a similar way $\{\,(x,y) \mid x > 2, 0 \leqslant y \leqslant 1\,\}$ is decomposed into three classes.

3. Finally, we apply the ordering constraint, see the third constraint of Definition 9.42 to $\{\,(x,y) \mid 1 < x < 2, 0 < y < 1\,\}$. Since the ordering of clocks now becomes important, this class is split into

$$\{\,(x,y) \mid 1 < x < 2, 0 < y < 1, \mathit{frac}(x) < \mathit{frac}(y)\,\},$$
$$\{\,(x,y) \mid 1 < x < 2, 0 < y < 1, \mathit{frac}(x) > \mathit{frac}(y)\,\},$$
$$\{\,(x,y) \mid 1 < x < 2, 0 < y < 1, x - y = 1\,\}.$$

A similar reasoning applies to $\{\,(x,y) \mid 0 < x < 1, 0 < y < 1\,\}$. The other classes are not further partitioned. For instance, $\{\,(x,y) \mid 1 < x < 2, y > 1\,\}$ is not further split as $y > c_y$.

Summarizing, we obtain twenty-eight clock regions: six corner points, fourteen open line segments, four open triangles, and four open clock regions. ■

Even for apparently simple timed automata, a large number of regions can arise. For this reason, we abstain from indicating the regions in more complex examples such as the railroad crossing and real-time mutual exclusion examples. The number of clocks, as well as the constants c_x, are essential factors that determine the number of regions. The number of clock regions and state regions is finite, i.e., consraint (C) holds. The following theorem contains an estimate for the number of clock regions. The number of state regions is a factor $|\mathit{Loc}|$ larger.

Theorem 9.46. Number of Regions

The number of clock regions is bounded from below and above as follows:

$$|C|! * \prod_{x \in C} c_x \quad \leqslant \quad | \mathit{Eval}(C)/\cong | \quad \leqslant \quad |C|! * 2^{|C|-1} * \prod_{x \in C} (2c_x + 2)$$

where for the upper bound it is assumed that $c_x \geqslant 1$ for all $x \in C$.

Proof: The lower and upper bounds are determined by considering a representation of clock regions such that there is a one-to-one relationship between the representation of a clock region and the clock region itself. This representation allows derivation of the bounds.

Let C be a set of clocks and $\eta \in \mathit{Eval}(C)$. Every clock region r can be represented by a tuple $\langle J, \wp, D \rangle$ where J is a family of intervals, $\wp$ is a permutation of a subset of clocks in C, and $D \subseteq C$ is a set of clocks such that

- $J = (J_x)_{x \in C}$ is a family of intervals with

  $$J_x \in \Big\{ [0,0],\ ]0,1[,\ [1,1],\ ]1,2[,\ \ldots,\ ]c_x{-}1, c_x[,\ [c_x, c_x],\ ]c_x, \infty[\Big\},$$

  such that $\eta(x) \in J_x$ for all clocks $x \in C$ and clock evaluations $\eta \in r$.

- Let C_{open} be the set of clocks $x \in C$ such that J_x is an open interval, i,e,

  $$C_{open} = \Big\{ x \in C \mid J_x \in \{\ ]0,1[,\ ]1,2[,\ \ldots,\ ]c_x{-}1, c_x[,\ ]c_x, \infty[\ \} \Big\}.$$

  $\wp = \{ x_{i_1}, \ldots, x_{i_k} \}$ is a permutation of $C_{open} = \{ x_1, \ldots, x_k \}$ such that for any $\eta \in r$ the clocks are ordered according to their fractional parts, i.e.,

  $$i_h < i_j \quad \text{imples} \quad \mathit{frac}(\eta(x_{i_h})) \leqslant \mathit{frac}(\eta(x_{i_j})).$$

- $D \subseteq C_{open}$ contains all clocks in C_{open} such that for all clock evaluations $\eta' \in [\eta]$ the fractional part for clock x_{i_j} corresponds to the fractional part for its predecessor $x_{i_{j-1}}$ in the permutation $\wp$:

  $$x_{i_j} \in D \text{ implies } \mathit{frac}(\eta(x_{i_{j-1}})) = \mathit{frac}(\eta(x_{i_j})).$$

There is a one-to-one relation between the clock regions and triples $\langle J, \wp, D \rangle$.

The indicated upper bound for the number of clock regions is obtained by the following combinatorial observation that there are

- exactly $\prod_{x \in C} (2c_x + 2)$ different interval families J,
- maximally $|C_{open}|! \leqslant |C|!$ different permutations over C_{open}, and
- maximally $2^{|C_{open}|-1} \leqslant 2^{|C|-1}$ different choices for $D \subseteq C \setminus \{ x_1 \}$.

The indicated lower bound is obtained when all clocks have a value in an open interval (though not the unbounded interval $]c_x, \infty[$), and all have different fractional parts. In this case $D = \varnothing$, and

$$J_x \in \Big\{ \,]0,1[, \;]1,2[, \; \ldots, \;]c_x{-}1, c_x[\, \Big\}.$$

As there are exactly $\prod_{x \in C} c_x$ possibilities for J and maximally $|C|!$ different permutations, the lower bound follows. ■

Example 9.47. Number of Regions

Let us illustrate the number of regions for a reasonable small timed automaton. Assume $|C| = n$ such that $c_x = 2$ for all $x \in C$. The lower bound for the number of clock regions indicated in Theorem 9.46 is $n! \cdot 2^n$. The minimal number of clock regions for n=2 equals 8; for n=3 and n=4 this rises to 48 and 384 respectively. For n=5, there are at least 3840 clock regions. ■

Lemma 9.48.

Let TA be a timed automaton and Φ a $TCTL_\Diamond$ formula both over the set C of clocks and $\cong$ the clock equivalence induced by TA and Φ. Then:

1. *For $\eta, \eta' \in Eval(C)$ such that $\eta \cong \eta'$:*

$$\eta \models g \quad \textit{if and only if} \quad \eta' \models g \textit{ for all } g \in ACC(TA) \cup ACC(\Phi)$$

2. *For $s, s' \in TS(TA)$ such that $s \cong s'$:*

$$s \models a \quad \textit{if and only if} \quad s' \models a \textit{ for any } a \in AP'.$$

where $AP' = AP \cup ACC(TA) \cup ACC(\Phi)$.

The first part of this lemma follows directly from the observations that justified the definition of clock equivalence. Using this result, the satisfaction relation of clock constraints

(see Definition 9.10) may now be used for clock regions; $[\eta] \models g$ denotes that $\eta' \models g$ for any $\eta' \in [\eta]$. As equivalent states have the same location, the second part of the lemma follows directly from the first part. All states of a state region thus satisfy the same clock constraints that occur in TA and Φ. This proves constraint (A) mentioned before.

It has been argued before that atomic clock constraints can in fact be considered as atomic propositions. Under this view, clock equivalence between states of $TS(TA)$ is in fact a *bisimulation*. In the following, again let $AP' = AP \cup ACC(TA) \cup ACC(\Phi)$. We lift the notion of clock reset to regions as follows.

Notation 9.49. Region Reset Operator

For $r \in Eval(C)/\cong$ and $D \subseteq C$ let

$$\mathsf{reset}\ D\ \mathsf{in}\ r\ =\ \{\, \mathsf{reset}\ D\ \mathsf{in}\ \eta \mid \eta\ \in r\,\}.$$

Since for $\eta, \eta' \in Eval(C)$ we have

$$\eta \cong \eta'\ \wedge\ D \subseteq C\ \Rightarrow\ \mathsf{reset}\ D\ \mathsf{in}\ \eta \cong \mathsf{reset}\ D\ \mathsf{in}\ \eta',$$

it follows that $\mathsf{reset}\ D\ \mathsf{in}\ r \in Eval(C)/\cong$. That is to say, resetting the clocks D in region r can be considered as a transition between state regions. ■

Theorem 9.50. Clock Equivalence is a Bisimulation

Clock equivalence is a bisimulation equivalence over AP'.

Proof: We prove that $\cong$ is a bisimulation (over AP') by checking the conditions of a bisimulation (see Definition 7.1, page 451). Let $s_1, s_2 \in TS(TA)$ such that $s_1 \cong s_2$, that is, $s_1 = \langle \ell, \eta_1 \rangle$ and $s_2 = \langle \ell, \eta_2 \rangle$ such that $\eta_1 \cong \eta_2$.

1. From the second part of Lemma 9.48 (page 718), it follows that $s_1 \models a$ if and only if $s_2 \models a$ for any $a \in AP'$.
2. To show that any transition emanating from s_1 can be mimicked by s_2, distinguish between discrete and delay transitions.
   (a) (Discrete transition). Assume $\langle \ell, \eta_1 \rangle = s_1 \xrightarrow{\alpha} s_1' = \langle \ell', \eta_1' \rangle$. By the semantics of timed automata, this means that there is a transition $\ell \overset{g:\alpha,D}{\hookrightarrow} \ell'$ in TA such that

$$\eta_1 \models g \quad \text{and} \quad \eta_1' = \mathsf{reset}\ D\ \mathsf{in}\ \eta_1 \models Inv(\ell').$$

Since $\eta_1 \cong \eta_2$ and $\eta_1 \models g$, it follows from the first part of Lemma 9.48 that $\eta_2 \models g$. Similarly, since $\eta_1 \cong \eta_2$, it follows that $\mathsf{reset}\ D\ \mathsf{in}\ \eta_1 \cong \mathsf{reset}\ D\ \mathsf{in}\ \eta_2$. As $\mathsf{reset}\ D\ \mathsf{in}\ \eta_1 \models \mathit{Inv}(\ell')$ we have $\mathsf{reset}\ D\ \mathsf{in}\ \eta_2 \models \mathit{Inv}(\ell')$. Thus:

$$s_2 \xrightarrow{\alpha} s_2' \;=\; \langle \ell', \mathsf{reset}\ D\ \mathsf{in}\ \eta_2 \rangle.$$

As the states s_1' and s_2' are in the same state region, it follows that $s_1' \cong s_2'$.

(b) (Delay transition). Assume $s_1 \xrightarrow{d} s_1' = s_1{+}d$ for some $d \in \mathbb{R}_{\geqslant 0}$. It is not difficult to see that for any d there exists d' such that $\eta_1{+}d \cong \eta_2{+}d'$. From $\eta_1 \models \mathit{Inv}(\ell)$ and $\eta_1{+}d \models \mathit{Inv}(\ell)$, it follows by Lemma 9.48 (page 718) that $\eta_2 \models \mathit{Inv}(\ell)$ and $\eta_2{+}d' \models \mathit{Inv}(\ell)$. But then $s_2 \xrightarrow{d'} s_2{+}d' = s_2'$ and $s_1' \cong s_2'$.

For transitions emanating from s_2' an analogous reasoning applies.

■

Note that in the delay transitions, the amount of delaying is ignored. Instead, only the fact that some delay may take place is of importance. Such bisimulation is also called *time abstract*.

Remark 9.51. The Need for Ordering the Fractional Parts of Clocks

For $\eta_1 \cong \eta_2$ it holds that whenever $\eta_1(x), \eta_2(x) \leqslant c_x$ and $\eta_1(y), \eta_2(y) \leqslant c_y$ then:

$$\mathit{frac}(\eta_1(x)) \leqslant \mathit{frac}(\eta_1(y)) \quad \text{if and only if} \quad \mathit{frac}(\eta_2(x)) \leqslant \mathit{frac}(\eta_2(y)).$$

Let us explain by means of a timed automaton with $C = \{ x, y \}$ and $c_x = 3$, $c_y = 1$ that without this constraint, $\cong$ would *not* be a bisimulation.

Assume for location ℓ, $\mathit{Inv}(\ell) = y < 1$. Consider state $s_1 = \langle \ell, \eta_1 \rangle$ with

$$1 < \eta_1(x) < 2, \quad 0 < \eta_1(y) < 1, \quad \eta_1(x) - \eta_1(y) > 1$$

and state $s_2 = \langle \ell, \eta_2 \rangle$ with

$$1 < \eta_2(x) < 2, \quad 0 < \eta_2(y) < 1, \quad \eta_2(x) - \eta_2(y) < 1.$$

The only difference between s_1 and s_2 is the ordering of the clocks. According to the first two constraints of the definition of $\cong$ (see Definition 9.42 on page 713), s_1 and s_2 would be equivalent. But the successor state regions of s_1 and s_2 after delaying are distinct.

There exists a delay transition from s_1 to state $s_1' = \langle \ell, \eta_1' \rangle$ with $\eta_1'(x) = 2$ and $\eta_1'(y) < 1$, i.e., the state region $\langle \ell, [x = 2, y < 1] \rangle$. As clocks proceed at the same rate, clocks x and

y both advanced by an equal amount. Due to $Inv(\ell) = y < 1$, any delay transition from state s_2 yields a state in the state region:

$$\langle \ell, \{\, (x, y) \mid 1 < x < 2, 0 < y < 1, x - y < 1 \,\} \rangle.$$

The state region $\langle \ell, [x = 2, y < 1] \rangle$ cannot be reached. States s_1 and s_2 have no corresponding delay transitions, and thus are not bisimilar.

Due to the constraint on the ordering of the fractional parts of the clocks (see the third constraint in Definition 9.42), this is avoided and $s_1 \ncong s_2$. ■

Theorem 9.50 ensures that the partitioning given by the state regions represents a refinement of the bisimulation quotient and allows defining a quotient transition system in which any edge $\langle \ell, [\eta] \rangle \to \langle \ell', [\eta'] \rangle$ is mimicked by $\langle \ell, \eta \rangle \to \langle \ell', \eta' \rangle$. States in the quotient transition system are state regions. Transitions between state regions are either delay or discrete transitions. The following example illustrates this by means of a small example.

Example 9.52. Region Transition System

Consider the simple timed automaton in Figure 9.23 (left) and let $\Phi = \text{true}$. As the largest constant with which x is compared is 2, $c_x = 2$. The region transition system is depicted in Figure 9.23 (lower part). Since there is only one location ℓ, in each state region the location is ℓ. All τ-labeled transitions are delay transitions. There are two discrete transitions in the region automaton, both labeled with α, that lead to the initial state. The only state region equipped with a τ-self-loop is called an *unbounded* region, as time may progress without bound while remaining in the same state region.

Let us now consider $\Phi = \Diamond(z \leqslant 2)$, i.e., $c_z = 2$. The region transition system for Φ and the timed automaton of Figure 9.23 (upper part) is depicted in Figure 9.24. Note that it is in fact the region transition system before (see Figure 9.23 (lower part)) extended with two "copies" of it. These "copies" are introduced for the constraints $x - z = 2$ and $z - x > 2$. Note that the clock z is never reset. This is typical for clocks occurring in Φ as there is no means in a formula to reset clocks. ■

To define the quotient transition system with respect to $\cong$, some auxiliary notions will be introduced. A clock region is unbounded whenever all the value of any clock exceeds its maximum constant.

Definition 9.53. Unbounded Clock Region

The clock region $r_\infty = \{\, \eta \in Eval(C) \mid \forall x \in C.\, \eta(x) > c_x \,\}$ is *unbounded*. ■

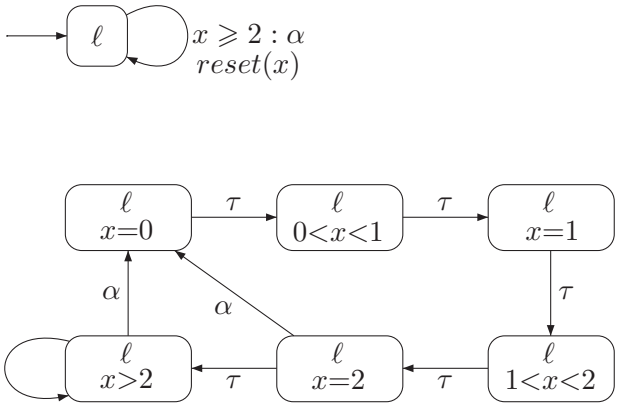


Figure 9.23: Region transition system for a simple timed automaton with $\Phi = \text{true}$.

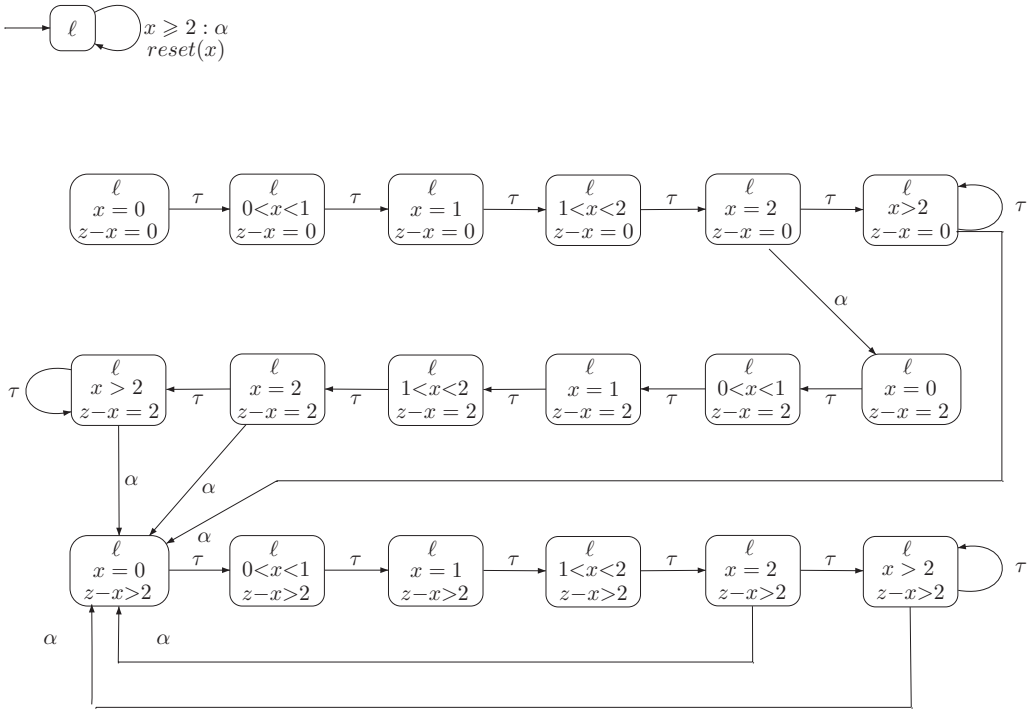


Figure 9.24: Region transition system for a simple timed automaton with Φ with $c_z = 2$.

The following definition defines the successor region of a region that is obtained by delaying.

Definition 9.54. Successor Region

Let $r, r' \in \mathit{Eval}(C)/\cong$. r' is the *successor* (clock) region of r, denoted $r' = \mathit{succ}(r)$, if either

1. $r = r_\infty$ and $r = r'$, or
2. $r \neq r_\infty$, $r \neq r'$ and for all $\eta \in r$:

$$\exists d \in \mathbb{R}_{>0}.\ \left(\eta{+}d \in r' \text{ and } \forall 0 \leqslant d' \leqslant d.\, \eta{+}d' \in r \cup r'\right).$$

The *successor* state region is defined as $\mathit{succ}(\langle \ell, r \rangle) = \langle \ell, \mathit{succ}(r) \rangle$. ■

Stated in words, the progress of time in the unbounded region yields the unbounded region. The clock region r' is the delay successor of $r \neq r_\infty$ if any clock valuation in r can delay to a clock valuation in r', without having the possibility of leaving the regions r and r' at any earlier time instant. Observe that the delay successor of a region is unique; this corresponds to the deterministic nature of progressing time. For any delay, the progress of time in an unbounded region is possible. This is not applicable to the other regions: after delaying sufficiently long, the region is left. Therefore, $\mathit{succ}(r) \neq r$ for $r \neq r_\infty$.

Note that $\mathit{succ}(\langle \ell, r \rangle)$ just defines the possible successor regions of a state region, and does not consider the location invariant of ℓ.

Example 9.55. Successor Clock Regions

For $C = \{ x \}$ with $c_x = 2$, the successor region are given by

$$\begin{aligned}
\mathit{succ}(\{0\}) &=]0,1[, \\
\mathit{succ}(]0,1[) &= \{1\}, \\
\mathit{succ}(\{1\}) &=]1,2[, \\
\mathit{succ}(]1,2[) &= \{2\}, \\
\mathit{succ}(\{2\}) &=]2,\infty[= r_\infty, \text{ and} \\
\mathit{succ}(]2,\infty[) &=]2,\infty[.
\end{aligned}$$

For $C = \{ x, y \}$ with $c_x = 2$ and $c_y = 1$, $r_\infty = \{ (x, y) \mid x > 2, y > 1 \}$. The following regions are unbounded with respect to y, but bounded for x:

$$\begin{array}{rclrcl} r_1 & = & \{ (x, y) \mid 0 < x < 1, y > 1 \} & r_2 & = & \{ (1, y) \mid y > 1 \}, \\ r_3 & = & \{ (x, y) \mid 1 < x < 2, y > 1 \} & r_4 & = & \{ (2, y) \mid y > 1 \}. \end{array}$$

It follows that $succ(r_i) = r_{i+1}$ for $0 < i \leqslant 3$ and $succ(r_4) = r_\infty$. For the bounded clock regions, the successor regions are defined by

$$\begin{array}{rcl} succ(\{(x, y) \mid 0 < x < 1, 0 < y < 1, x < y\}) & = & \{(x, 1) \mid 0 < x < 1\}, \\ succ(\{(x, y) \mid 0 < x < 1, 0 < y < 1, x > y\}) & = & \{(1, y) \mid 0 < y < 1\}, \\ succ(\{(x, y) \mid 0 < x < 1, 0 < y < 1, x = y\}) & = & \{(1, 1)\}, \\ succ(\{(x, y) \mid 1 < x < 2, 0 < y < 1, x < y\}) & = & \{(x, 1) \mid 1 < x < 2\}, \\ succ(\{(x, y) \mid 1 < x < 2, 0 < y < 1, x > y\}) & = & \{(2, y) \mid 0 < y < 1\}, \\ succ(\{(x, y) \mid 1 < x < 2, 0 < y < 1, x = y\}) & = & \{(2, 1)\}. \end{array}$$

Besides, $succ(\{(0, 0)\}) = \{(x, y) \mid 0 < x < 1, 0 < y < 1, x = y\})$ and $succ(\{(2, 1)\}) = r_\infty$. ■

Remark 9.56. Time Passage

The successor region $succ(r)$ denotes the region that is obtained from r by delaying for some amount. Not any delay in r, however, results in $succ(r)$. A delay within a region or to some (not necessarily direct) successor region of $succ(r)$ is possible too. Delaying within a region is only possible if there is no clock that has a fixed value. This applies to regions like $]0, 1[$ (for $|C|{=}1$) or $\{ (x, y) \mid 0 < x < 1, 0 < y < 1, x{-}y = 0 \}$ for $C = \{ x, y \}$. Delaying within e.g., the regions $\{ 0 \}$ or $\{ (x, 1) \mid 0 < x < 1 \}$ is impossible—any delay in these regions implies changing region.

The passage of time over several regions implies a change from r to $succ^n(r)$ for $n > 1$. For example, the delay transition in a timed automaton

$$\langle \mathit{off}, 0 \rangle \xrightarrow{1.578} \langle \mathit{off}, 1.578 \rangle$$

can be decomposed into three transitions, in which any delay is made to a state of the successor region

$$\underbrace{\langle \mathit{off}, 0 \rangle}_{\text{region } \{ 0 \}} \xrightarrow{0.5} \underbrace{\langle \mathit{off}, 0.5 \rangle}_{\text{region }]0, 1[} \xrightarrow{0.5} \underbrace{\langle \mathit{off}, 1 \rangle}_{\text{region } \{ 1 \}} \xrightarrow{0.578} \underbrace{\langle \mathit{off}, 1.578 \rangle}_{\text{region }]1, \infty[}.$$

These considerations show that any time passage can be described by successor regions. ■

Recall that the interpretation of TCTL formulae was provided by considering time-divergent paths. The following result provides a characterization of time-convergent paths in a non-zeno timed automaton in terms of state regions. Given that the timed automaton is non-zeno, any time-convergent path contains finitely many delay transitions. That is to say, from some point on, a time-convergent path stays within a certain state region. Vice versa, a path that from some point on does not change state region, is time-convergent (except for paths that reside in the unbounded region r_∞).

Lemma 9.57. Time Convergence

For non-zeno TA and $\pi = s_0\, s_1\, s_2 \ldots$ an initial, infinite path fragment in TS(TA):

(a) if π is time-convergent, then there exists a state region $\langle \ell, r \rangle$ such that for some j

$$s_i \in \langle \ell, r \rangle \quad \textit{for all } i \geqslant j.$$

(b) if there exists a state region $\langle \ell, r \rangle$ with $r \neq r_\infty$ and an index j such that

$$s_i \in \langle \ell, r \rangle \quad \textit{for all } i \geqslant j,$$

then π is time-convergent.

Proof:

(a) Assume $\pi = s_0\, s_1\, s_2 \ldots$ is a time-convergent path in $TS(TA)$. Since π is time-convergent, finitely many actions occur in π, i.e., from some point, say from index k on, only delays occur. Thus there exists a state $\langle \ell, \eta \rangle$ and a sequence of non-negative real numbers $d_{k+1}, d_{k+2} \ldots$ such that

$$s_k \;=\; \langle \ell, \eta \rangle \quad \text{and} \quad s_i \in \langle \ell, \eta + d_{k+1} + d_{k+2} + \ldots + d_i \rangle \;\text{ for all } i > k.$$

Recall that the number of clock regions is finite, and observe that apart from the self-loop at r_∞, cycles (of delay successors) of clocks, regions do not exist. Thus, as π is infinite, and from some point only delays occur in π, some region must be visited continuously from some index on. That is, there exists a clock region r and an index $j \geqslant k$ such that

$$s_i \;=\; \langle \ell, \eta + d_{k+1} + d_{k+2} + \ldots + d_i \rangle \;\in\; \langle \ell, r \rangle \quad \text{for all } i \geqslant j.$$

(b) Assume there exists a state region $\langle \ell, r \rangle$ with $r \neq r_\infty$ and some j such that $s_i \in \langle \ell, r \rangle$ for all $i \geqslant j$. We show that $\pi = s_0\, s_1 \ldots$ is time-convergent by contraposition.

Assume π is time-divergent. That is, infinitely many delay transitions d_i occur in π and $\sum_i d_i$ is not converging. In particular, infinitely many delay transitions occur in $\langle \ell, r \rangle$. Thus, no evaluation $\eta \in r$ exists with $\eta(x) = 0$ for some x. Distinguish two cases.

(i) Infinitely many actions $\alpha \in Act$ occur in π. We show that this is impossible. Assume action α occurs infinitely often in π. Then π traverses some loop $\ell \overset{g:\alpha,D}{\hookrightarrow} \ell$ in TA infinitely often. As no evaluation $\eta \in r$ exists with $\eta(x) = 0$ for some x, it follows that $D = \varnothing$. As $r \models g$, the action α can be executed infinitely often in every state $\langle \ell, \eta \rangle$ of $\langle \ell, r \rangle$, i.e., the transition system $TS(TA)$ contains a loop $\langle \ell, \eta \rangle \xrightarrow{\alpha} \langle \ell, \eta \rangle$ for any state $\langle \ell, \eta \rangle$ of $\langle \ell, r \rangle$. This, however, contradicts the assumption that TA is non-zeno.

(ii) Finitely many actions $\alpha \in Act$ are executed along π. Then there is an index j such that only delay transitions $d_{j+1}, d_{j+2}, \ldots$ occur in the path fragment $s_j\, s_{j+1} \ldots$. Let $s_j = \langle \ell, \eta \rangle$. Since π is time-divergent, $\sum_{i=j+1}^{\infty} d_i$ does not converge. That is, there exists an index $k \geqslant j$ such that $\sum_{i=j+1}^{k} d_i + \eta \in r_\infty$. This, however, contradicts the assumption that $r \neq r_\infty$ is not left.

■

Lemma 9.57 suggests that in order to indicate time-divergent paths it suffices to ignore delay transitions within a region.

Definition 9.58. Region Transition System

Let $TA = (Loc, Act, C, \hookrightarrow, Loc_0, Inv, AP, L)$ be a non-zeno timed automaton and let Φ be a $\mathrm{TCTL}_\Diamond$ formula. Then the region transition system of TA for Φ is defined as follows.

$$RTS(TA, \Phi) = (S', Act \cup \{\tau\}, \rightarrow', I', AP', L') \quad \text{where}$$

- if S is the set of all states in $TS(TA)$, then the state space of $RTS(TA, \Phi)$ is

$$S' = S/\cong \; = \; \{\, [s] \mid s \in S \,\},$$

the set of all state regions,

- $I' = \{\, [s] \mid s \in I \,\}$,
- $AP' = ACC(TA) \cup ACC(\Phi) \cup AP$,
- $L'(\langle \ell, r \rangle) \; = \; L(\ell) \cup \{\, g \in AP' \setminus AP \mid r \models g \,\}$,

- the transition relation $\rightarrow'$ is defined by:

$$\frac{\ell \overset{g:\alpha,D}{\hookrightarrow} \ell' \quad \wedge \quad r \models g \quad \wedge \quad \mathsf{reset}\ D\ \mathsf{in}\ r \models \mathit{Inv}(\ell')}{\langle \ell, r\rangle \xrightarrow{\alpha}{}' \langle \ell', \mathsf{reset}\ D\ \mathsf{in}\ r\rangle} \quad \text{and}$$

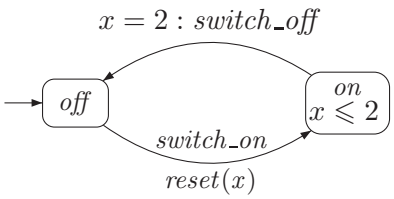
$$\frac{r \models \mathit{Inv}(\ell) \quad \wedge \quad \mathit{succ}(r) \models \mathit{Inv}(\ell)}{\langle \ell, r\rangle \xrightarrow{\tau}{}' \langle \ell, \mathit{succ}(r)\rangle}.$$

■

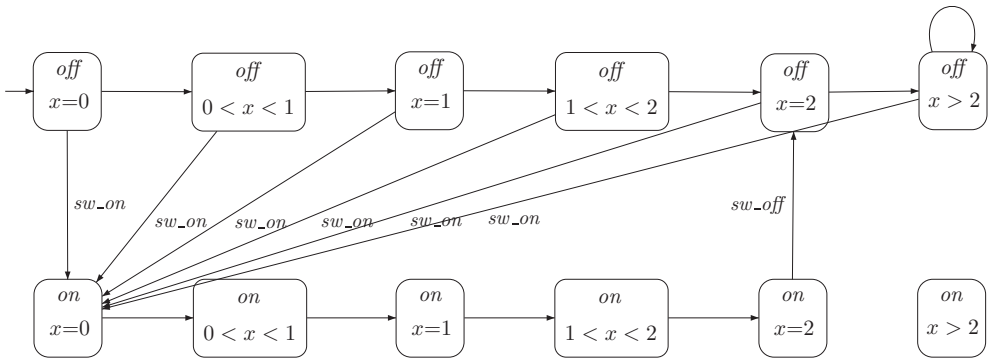
The dependence on formula Φ is implicit; its maximal constants are of relevance to the clock equivalence only. In case the region transition system does not depend on (the maximal constants occurring in) Φ, we simply write $\mathit{RTS}(\mathit{TA})$. This applies, e.g., when $\Phi = \text{true}$ or the constants in Φ for clock x do not exceed the maximal constants for x in TA, for any x.

Example 9.59. Light Switch

Consider the timed automaton modeling a light switch:



Let $\Phi = \text{true}$. Then $c_x = 2$, and the region transition system $\mathit{RTS}(\mathit{Switch}) = \mathit{RTS}(\mathit{Switch}, \text{true})$ is as follows, where the τ action labels are omitted:



Although there exists a timelock in state $\langle on, [x > 2]\rangle$, the timed automaton *Switch* is timelock-free, since $\langle on, [x > 2]\rangle$ is unreachable.

Any path in the (infinite) transition system of timed automaton *Switch* has a corresponding path in $RTS(Switch)$. The following path in $TS(Switch)$:

$$\langle off, 0\rangle \langle off, 2.5\rangle \langle off, 2.7\rangle \langle on, 0\rangle \langle on, 1.456\rangle \langle on, 1.7\rangle \langle on, 2\rangle \ \ldots$$

corresponds to the path in $RTS(TA)$:

$$\langle off, [x = 0]\rangle \langle off, [0 < x < 1]\rangle \langle off, [x = 1]\rangle \langle off, [1 < x < 2]\rangle$$
$$\langle off, [x = 2]\rangle \langle off, [x > 2]\rangle \langle on, [x = 0]\rangle \langle on, [0 < x < 1]\rangle$$
$$\langle on, [x = 1]\rangle \langle on, [1 < x < 2]\rangle \langle on, [x = 2]\rangle \ldots$$

(In fact, both paths are stutter trace equivalent with respect to the set of propositions AP'.) Conversely, every path in $RTS(Switch)$ represents a *set* of paths in $TS(Switch)$. For example, the path fragment

$$\langle off, [x = 0]\rangle \langle off, [0 < x < 1]\rangle \langle on, [x = 0]\rangle \langle on, [0 < x < 1]\rangle \ldots$$

in $RTS(Switch)$ is representative for all path fragments in $TS(Switch)$ which reside in location *off* for t time units with $0 < t < 1$ and then change to location *on*; e.g.,

$$\langle off, 0\rangle \quad \langle off, 0.231\rangle \quad \langle off, 0.5788\rangle \quad \langle off, 0.98\rangle \quad \langle on, 0\rangle \qquad \ldots \text{and}$$
$$\langle off, 0\rangle \quad \langle off, 0.001\rangle \quad \langle on, 0\rangle \qquad \langle on, 0.789\rangle \quad \langle on, 0.79\rangle \quad \ldots$$

Although these path fragments change from *off* to *on* at different time instants, they are stutter trace equivalent with respect to the set AP' of propositions.

The time-divergent path

$$\pi \ = \ \langle off, 0\rangle \langle off, 1\rangle \langle off, 2\rangle \langle off, 3\rangle \ \ldots,$$

in which the light is never switched on, finally reaches the unbounded state region $\langle off, [x > 2]\rangle$, in which time passage is represented by the self-loop:

$$\langle off, [x > 2]\rangle \ \xrightarrow{\tau} \langle off, [x > 2]\rangle.$$

Thus, in the region transition system, π is represented by the path

$$\langle off, [x = 0]\rangle \qquad \langle off, [0 < x < 1]\rangle \quad \langle off, [x = 1]\rangle$$
$$\langle off, [1 < x < 2]\rangle \quad \langle off, [x = 2]\rangle \qquad \langle off, [x > 2]\rangle \quad \langle off, [x > 2]\rangle \ldots$$

■

Example 9.60. Region Transition System with Two Clocks

Consider the timed automaton *TA*:

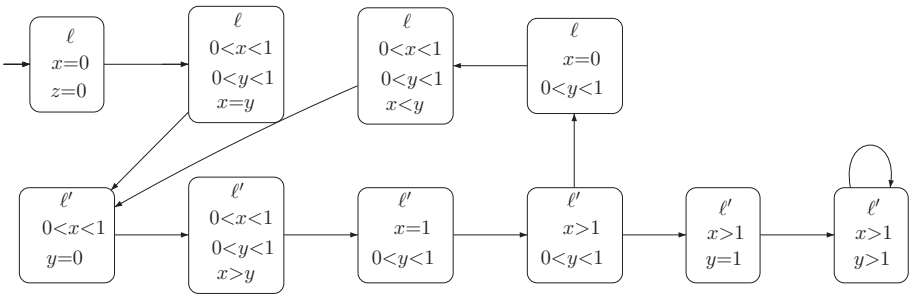
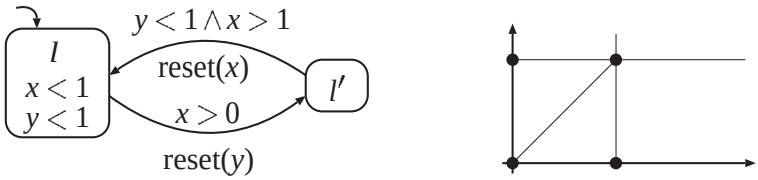


Figure 9.25: Region transition system for example of a example timed automaton.



where $c_x = c_y = 1$. The reachable part of $RTS(TA)$ is indicated in Figure 9.25. It is not difficult to see that the region transition system exhibits a path satisfying $\Box(y < 1)$. The following result asserts that this yields $TA \models \exists\Box(y < 1)$. ■

The following result establishes the correctness of the model-checking approach of timed automata via region transition systems.

Theorem 9.61. Correctness of TCTL Model Checking

For non-zeno timed automaton TA and $TCTL_\Diamond$ formula Φ:

$$\underbrace{TA \models \Phi}_{\text{TCTL semantics}} \quad \textit{if and only if} \quad \underbrace{RTS(TA, \Phi) \models \Phi}_{\text{CTL semantics}}.$$

Proof: Let TA be a non-zeno timed automaton. By structural induction over the structure of Φ, it is proven that any state subformula Ψ (of Φ) and any state s of $TS(TA)$:

$$s \models_{\text{TCTL}} \Psi \quad \text{if and only if} \quad [s] \models_{\text{CTL}} \Psi.$$

The base cases are straightforward. For the induction step, it is assumed that the maximal state subformulae of Ψ have been replaced by atomic propositions. (This assumption is justified by the fact that CTL model checking is a recursive descent procedure over the parse tree of Ψ.) Consider the proof for $\Psi = \forall(a \,\mathsf{U}\, b)$ for $a, b \in AP'$; the proof for the existentially quantified until-operator goes along similar lines.

- ($\Rightarrow$): Assume $s \models_{\text{TCTL}} \forall(a \,\mathsf{U}\, b)$ and $s \in TS(TA)$. Then $\pi \models_{\text{TCTL}} a \,\mathsf{U}\, b$ for any time-divergent path $\pi \in Paths_{div}(s)$. Let

$$\pi' = \underbrace{\langle \ell_0, r_0 \rangle}_{[s]} \langle \ell_1, r_1 \rangle \langle \ell_2, r_2 \rangle \ldots$$

be a path in $RTS(TA, \forall\Diamond a)$ starting in the state region $[s]$. Then there exists a corresponding path $\pi = s_0\, s_1\, s_2 \ldots$ in $TS(TA)$ such that $s_0 = s$ and $s_i = \langle \ell_i, \eta_i \rangle$ where $\eta_i \in r_i$ for all $i \geqslant 0$. As none of the regions $r_i \neq r_\infty$ is equipped with a self-loop, π' either traverses a cycle (of length at least two) in $RTS(TA)$, or the self-loop at r_∞, infinitely often. That is, there does not exist a state region $\langle \ell_i, r_i \rangle$ such that for some j:

$$s_i \in \langle \ell_i, r_i \rangle \quad \text{for all } i \geqslant j.$$

By Lemma 9.57(a) (page 725), it follows that π is time-divergent.

Since $s \models_{\text{TCTL}} \forall(a \,\mathsf{U}\, b)$, it follows that $\pi \models a \,\mathsf{U}\, b$. In particular, there exists a state s_j with $s_j \models b$ such that $s_i \models a \vee b$ for all $i \leqslant j$. As in π each visited region in π' is represented by a state, it follows that $\langle \ell_j, r_j \rangle \models b$ and $\langle \ell_i, r_i \rangle \models a \vee b$ for any $i \leqslant j$. Hence, $\pi' \models_{\text{CTL}} a \,\mathsf{U}\, b$.

- ($\Leftarrow$): Assume $[s] \models_{\text{CTL}} \forall(a \,\mathsf{U}\, b)$. Let $\pi = s_0\, s_1\, s_2 \ldots$ be a time-divergent path in $TS(TA)$ with $s = s_0$ and

$$s_i \;=\; \langle \ell_i, \eta_i \rangle, \quad i = 0, 1, 2, \ldots$$

We show that $\pi \models_{\text{TCTL}} \Diamond a$. Assume w.l.o.g. that delay transitions in π are between successor regions. (Any delay transition can be decomposed into delay transitions between successor regions.) As π is time-convergent, it follows from Lemma 9.57(b) (page 725), that delay transitions within a state region $\langle \ell, r \rangle$ with $r \neq r_\infty$ do not occur in π. Then

$$\pi' \;=\; [s_0]\,[s_1]\,[s_2] \ldots \;=\; \langle \ell_0, [\eta_0] \rangle \langle \ell_1, [\eta_1] \rangle \langle \ell_2, [\eta_2] \rangle \ldots$$

is a path in $RTS(TA, \Phi)$ starting in the state region $[s_0] = [s]$. As $[s] \models_{\text{CTL}} \forall(a \,\mathsf{U}\, b)$ it follows that $\pi' \models_{\text{CTL}} a \,\mathsf{U}\, b$. Thus, there exists an index j such that $[s_j] \models b$ and $[s_i] \models a$ for $i \leqslant j$. Hence, $s_j \models b$ and $s_i \models a \vee b$ for any $i \leqslant j$, and $\pi \models_{\text{TCTL}} \Diamond a$. It follows that $s \models_{\text{TCTL}} \forall(a \,\mathsf{U}\, b)$.

■

The region transition system $RTS(TA)$ provides a simple and effective criterion to check whether non-zeno timed automaton TA is timelock-free.

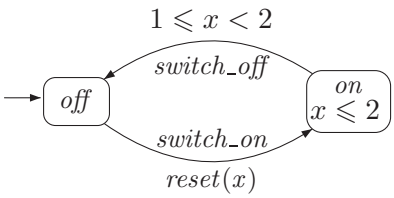
Theorem 9.62. Timelock Freedom

Non-zeno timed automaton TA is timelock-free iff RTS(TA) does not have any reachable terminal states.

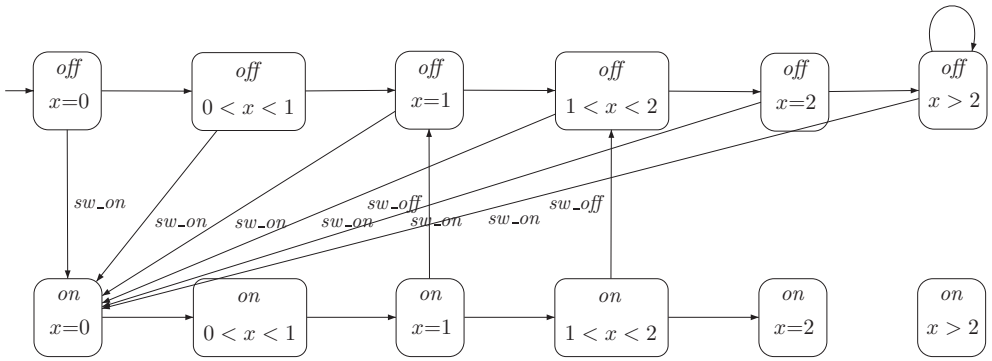
Proof: Essentially, the proposition follows from Lemma 9.57 (page 725). ■

Example 9.63. Modified Light Switch (Timelock)

Consider the timed automaton $Switch_{timelock}$:



which has a timelock in state $\langle on, \eta \rangle$ with $\eta(x) > 1$. The transition system $RTS(Switch_{timelock})$ is of the following form.



The state region $\langle on, [x{=}2] \rangle$ is reachable and does not have any outgoing transitions. Thus, $Switch_{timelock}$ is not timelock-free.

The light switch described in Example 9.12 (page 688) is timelock-free, since the associated region transition system does not contain a reachable terminal state, see Example 9.59 (page 727). ■

Timelock freedom can thus be checked by reachability analysis of the region transition system.

Remark 9.64. Bisimulation, Time Divergence, and Fairness

The region transition system $RTS(TA)$ is *not* bisimilar to $TS(TA)$, since neither delay transitions within a state region nor delays over several regions are explicitly represented in $RTS(TA)$. $RTS(TA)$ and $TS(TA)$ are, however, stutter-equivalent for the set of atomic propositions AP'. This follows from the fact that delay transitions over several state regions in $TS(TA)$ can be simulated in $RTS(TA)$ by a sequence of delay transitions between successor regions, and that delay transitions in $TS(TA)$ within a state region may be considered as stutter steps. Equivalent states satisfy the same $a \in AP'$ (see Lemma 9.48, page 718) and every path $\pi' = r_0\, r_1\, r_2 \ldots$ in $RTS(TA, \Phi)$ can be lifted to a path $\pi = s_0\, s_1\, s_2 \ldots$ with $s_i \in r_i$ for all $i \geqslant 0$.

The reader may wonder whether region transition systems can be enriched with self-loops (for bounded regions), and by incorporating delay transitions that jump over successor regions. Let us discuss both enrichments. Self-loops could be added to region transition systems to mimic the delay within a region. In order to only consider time-divergent paths, fairness assumptions need to be imposed, however, to rule out infinite (time-convergent) behaviors that take a self-loop of a bounded region infinitely often. Incorporating delay transitions is more problematic, though. This is due to the fact that in order to determine the truth of a TCTL path formula in a given path, states that are not explicitly on that path may be relevant (and thus should not be skipped). For example, any path corresponding to an execution fragment of the form

$$\langle \ell, [x = 0] \rangle \underbrace{\xrightarrow{d_1} \ldots \xrightarrow{d_1}}_{\text{delay of 2 time units}} \langle \ell, [x = 2] \rangle \longrightarrow \ldots$$

satisfies $\Diamond(x = 1)$, since the interim state $\langle \ell, [x = 1] \rangle$ (which does not explicitly occur in the path representation) is passed and satisfies the clock constraint $x{=}1$. Delay transitions that take over successor regions are therefore not represented in region transition systems. ■

9.3.3 The TCTL Model-Checking Algorithm

The previous sections have described a technique to reduce the model-checking problem $TA \models \Phi$ for non-zeno timed automaton TA and TCTL formula Φ to a CTL model-checking problem on the region transition system for the clocks and their maximal constants in TA and Φ. TCTL formulae have been assumed to be of the form $\exists(\Phi\, \mathsf{U}^J\, \Psi)$ with $J \neq [0, \infty)$ and such that Φ and Ψ do not contain any until formulae equipped with intervals different from $[0, \infty)$. Such formulae are transformed into CTL formulae using a fresh clock z, say. Before explaining on how this approach can be generalized toward nested formulae of the form U^J with $J \neq [0, \infty)$, we summarize the approach so far by means of a small example.

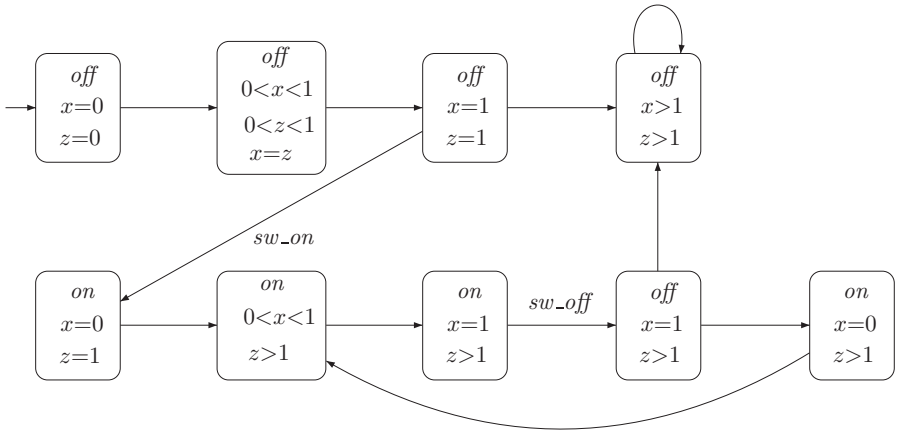
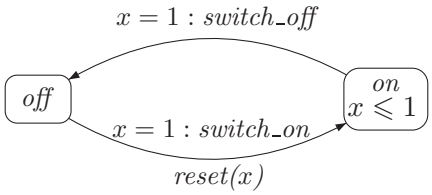


Figure 9.26: Region transition system for example of a light switch timed automaton.

Example 9.65. Formulae with a Single Time Bound

Consider the following timed automaton TA and the TCTL formula $\Phi = \exists\Diamond^{\leqslant 1} on$.



As a first step, Φ is replaced by $\widehat{\Phi} = \exists\Diamond((z \leqslant 1) \wedge on)$ and TA is equipped with an additional clock z. The maximal constants for the clocks x and z are $c_x = 1$ and $c_z = 1$. The region transition system $TS = RTS(TA \oplus z, \widehat{\Phi})$ is depicted in Figure 9.26.

The state region $\langle on, [x{=}0, z{=}1]\rangle \models (z \leqslant 1) \wedge on$ and is reachable from the initial state region. Therefore $TS \models_{CTL} \exists\Diamond\,((z \leqslant 1) \wedge on)$, and thus $TA \models \exists\Diamond^{\leqslant 1} on$. ■

It remains to clarify how formulae with nested time bounds, such as, e.g.,

$$\Phi \;=\; \forall\Box^{\geqslant 3}\; \exists\Diamond^{]1,2]}\; on$$

are treated. As before, we assume a non-zeno timed automaton $TA = (Loc,\, Act,\, C,\, \hookrightarrow,\, Loc_0,\, Inv, AP, L)$. Without loss of generality, TA is assumed to be timelock-free. A simple way of treating formulae with nested time bounds is to introduce a fresh clock for each

subformula. This amounts to, e.g., transforming the example formula Φ just above into:

$$\widehat{\Phi} \;=\; \forall\Box\Big((z_1 \geqslant 3) \rightarrow \exists\Diamond\,(z_2 \in]1,2] \wedge on)\Big).$$

To check whether $TA \models \Phi$ amounts to applying a CTL model checker to $RTS(TA \oplus \{\, z_1, z_2 \,\})$ and $\widehat{\Phi}$. Although this is a straightforward approach, the number of clocks grows linearly with the number of time bounds. As the size of the region automaton is exponential in the number of clocks, it is beneficial to keep the number of (additional) clocks minimal. In the sequel it will be shown that a *single* additional clock suffices for TCTL formulae with arbitrary many (nested) time bounds.

As before, the timed automaton TA is equipped with a single fresh clock z, say, that does not occur in TA. The principle of the model-checking algorithm is similar in spirit to that of CTL—a recursive descent procedure over the parse tree of TCTL formula Φ. The clock z is of interest only for subformulae Ψ of Φ that contain a time bound. Once $Sat(\Psi)$ is determined, the clock z is available for checking another subformula of Φ which contains a time bound.

Let $TS = TS(TA \oplus z)$, the transition system of TA equipped with the additional clock z which will be used to evaluate the time-constrained formulae. The state space of TS is

$$S_{ts} \;=\; Loc \times Eval(C \cup \{\, z \,\}).$$

For state $s = \langle \ell, \eta \rangle \in S_{ts}$ and $d \in \mathbb{R}_{\geqslant 0}$, let state $s\{z := d\} \in S_{ts}$ which originates from s by assigning value d to clock z and leaving all other clocks unaffected:

$$s\{z := d\} \;=\; \langle \ell, \eta\{z := d\} \rangle \quad \text{where} \quad \eta\{z := d\}(x) \;=\; \begin{cases} d & \text{if } z = x \\ \eta(x) & \text{otherwise} \end{cases}$$

Let $R = RTS(TA \oplus z, \Phi)$ and let S_{rts} be the state space of R. For state region $[s] = \langle \ell, r \rangle \in S_{rts}$, let $[s]\{z := 0\}$ denote the state region $\langle \ell, r\{z := 0\} \rangle$ in S_{rts} where

$$r\{z := 0\} \;=\; \{\, \eta\{z := 0\} \mid \eta \in r \,\}.$$

Example 9.66. Regions in $RTS(TA \oplus z, \Phi)$

The states in the region transition systems $R = RTS(TA \oplus z, \Phi)$ and $RTS(TA, \Phi)$ are strongly related; in fact, their only difference is the clock evaluation for clock z. Let $C = \{\, x \,\}$. The state region $\langle \ell, [0 < x < 1] \rangle$ in $RTS(TA, \Phi)$ corresponds to the following state regions in S_{rts}:

- $\langle \ell, [0 < x < 1, z = c] \rangle$ for $c = 0, 1, \ldots, c_z$,

- $\langle \ell, [0 < x < 1, z > c_z] \rangle$,
- $\langle \ell, [0 < x < 1, c < z < c{+}1, \mathit{frac}(x) \bowtie \mathit{frac}(z)] \rangle$ for $c = 0, 1, \ldots, c_z$ and $\bowtie \in \{<, >, =\}$.

■

In the sequel, we present an algorithm to check whether $TA \models \Phi$ for TCTL formula Φ that may contain *several* (possibly nested) time bounds. The basic idea is to exploit the CTL model-checking algorithm on the region transition system of $TA \oplus z$ and the CTL formula $\widehat{\Phi}$. This algorithm computes the satisfaction sets:

$$Sat_R(\widehat{\Phi}) = \{\, [s] \in S_{rts} \mid [s] \models \widehat{\Phi} \,\}.$$

Due to the bottom-up nature of the CTL model-checking algorithm, on computing $Sat_R(\Psi)$ for subformula Ψ of $\widehat{\Phi}$, the satisfaction sets $Sat_R(\Psi_i)$ for any subformula Ψ_i of Ψ have been determined and are at our disposal. Any subformula Ψ_i can thus be replaced by (fresh) atomic propositions a_{Ψ_i}. Accordingly, states in the region transition system R are either labeled with atomic propositions that occur in TA, atomic clock constraints in $\widehat{\Phi}$, or propositions a_{Ψ_i} for any subformula Ψ_i of Φ. States in R may, in addition, be labeled with propositions of the form $z \in J$ in case $\Psi_i \,\mathsf{U}^J\, \Psi_j$ is a subformula of $\widehat{\Phi}$. These propositions are the atomic clock constraints of clock z. The following example illustrates this.

Example 9.67. Propositions for the Region Transition System

Consider the TCTL formula:

$$\Phi \;=\; \forall\Box^{\leqslant 3}\,\Big(\underbrace{\exists\Diamond^{[2,6]} a}_{=\Psi_1} \;\wedge\; \underbrace{\exists\Box^{]2,5[}\underbrace{\forall\Diamond^{\geqslant 3}\underbrace{(b \wedge (x=9))}_{=\Psi_2}}_{=\Psi_3}}_{=\Psi_4}\Big)}_{=\Psi_5},$$

The set of propositions of R contains the propositions a and b, the clock constraint $x{=}9$, and the propositions a_{Ψ_1} through a_{Ψ_5}, and a_Φ. In addition, the clock constraints $z \leqslant 3$, $z \in [2,6]$, $z \in]2,5[$ and $z \geqslant 3$ are included; these clock constraints correspond to the time bounds of the subformulae of Φ. ■

As stated above, verifying whether $TA \models \Phi$ boils down to applying a CTL model checker to the region transition system R and $\widehat{\Phi}$. For subformula Ψ of Φ we have

$$\underbrace{s \in Sat(\Psi)}_{\text{satisfaction relation in } TS(TA)} \qquad \text{iff} \qquad \underbrace{a_\Psi \in L_{rts}([s])}_{\text{label in } RTS(TA,\,\Phi)}.$$

Algorithm 44 (page 737) summarizes the essential steps of the TCTL model-checking algorithm. First, the region transition system for $TA \oplus z$ and Φ is determined. By means of a recursive descent procedure over the parse tree of Φ, the satisfaction set $Sat_R(\Psi) = \{\, [s] \in S_{rts} \mid [s] \models \Psi \,\}$ is determined for each subformula Ψ of Φ. For propositional logical formula Ψ, the treatment is straightforward. The interesting cases are the path formulae. Let us explain the treatment for $\Psi = \exists(a \,\mathsf{U}^J\, b)$; for universally quantified formulae, the procedure is similar. A CTL model checker is applied on the region transition system and the CTL formula $\widehat{\Psi} = \exists(a \,\mathsf{U}\, ((z \in J) \wedge b))$. Note that $\widehat{\Psi}$ is a CTL formula over the set of atomic propositions that occur in Φ, clock constraints in Φ, and clock constraints on z (like $z \in J$). Theorems 9.37 (page 707) and 9.61 (page 729) yield

$$s \models_{\text{TCTL}} \Psi \quad \text{iff} \quad s\{z := 0\} \models_{\text{TCTL}} \widehat{\Psi} \quad \text{iff} \quad [s\{z := 0\}] \models_{\text{TCTL}} \widehat{\Psi}$$

where $[s\{z := 0\}]$ is a state in the region transition system R. Accordingly, all states $[s]$ in R for which

$$[s\{z := 0\}] \models_{\text{TCTL}} \widehat{\Psi}$$

are labeled with proposition a_Ψ once $Sat_R(\Psi)$ has been determined. Once all subformulae have been treated, $TA \models \Phi$ if and only if all initial states in the region transition system R are labeled with a_Φ. In case TA refutes Φ, the path fragment that is returned as a counterexample by the CTL model checker for the region transition system can be returned. The same applies to the generation of witnesses.

Given the fact that $TA \models \Phi$ can be decided by means of a CTL-like model-checking algorithm on the region transition system, we obtain:

Theorem 9.68. Time Complexity of TCTL Model Checking

For timed automaton TA and TCTL formula Φ, the TCTL model-checking problem $TA \models \Phi$ can be determined in time $\mathcal{O}\,((N+K) \cdot |\,\Phi\,|)$, where N and K are the number of states and transitions in the region transition system $RTS(TA, \Phi)$, respectively.

The worst-case time complexity of TCTL model checking is thus linear in the size of Φ—due to the recursive descent over the parse tree of Φ—and in the size of the region transition system. As the state-space size of the region transition system grows exponentially in the number of clocks (and maximal constants c_x) (see Theorem 9.46, page 717), the time complexity of TCTL model checking is exponential in the number of clocks. Although this does not affect the worst-case time complexity, it provides ample means to significantly reduce the number of states in the transition system (like the region transition system) that symbolically represent the behavior of the timed automaton.

We state here without proof that the verification problem for real-time systems is computationally hard, as even for TCTL there is a PSPACE lower bound.

Algorithm 44 TCTL model checking (basic idea)

Input: non-zeno, timelock-free timed automaton TA and TCTL formula Φ
Output: "yes" if $TA \models \Phi$, "no" otherwise.

$R := RTS(TA \oplus z, \Phi)$; (* with state space S_{rts} and labeling L_{rts} *)

for all $i \leqslant |\,\Phi\,|$ **do**
  for all $\Psi \in Sub(\Phi)$ with $|\,\Psi\,| = i$ **do**

    switch(Ψ):

true	:	$Sat_R(\Psi) := S_{rts}$;
a	:	$Sat_R(\Psi) := \{\, s \in S_{rts} \mid a \in L_{rts}(s) \,\}$;
$\Psi_1 \ \wedge \ \Psi_2$	:	$Sat_R(\Psi) := \{\, s \in S_{rts} \mid \{a_{\Psi_1}, a_{\Psi_2}\} \subseteq L_{rts}(s) \,\}$;
$\neg\Psi'$	:	$Sat_R(\Psi) := \{\, s \in S_{rts} \mid a_{\Psi'} \notin L_{rts}(s) \,\}$;
$\exists(\Psi_1 \,\mathsf{U}^J\, \Psi_2)$	:	$Sat_R(\Psi) := Sat_{\mathrm{CTL}}\big(\exists(\,(a_{\Psi_1} \vee a_{\Psi_2})\,\mathsf{U}\,((z \in J) \wedge a_{\Psi_2})\,)\big)$;
$\forall(\Psi_1 \,\mathsf{U}^J\, \Psi_2)$	:	$Sat_R(\Psi) := Sat_{\mathrm{CTL}}\big(\forall(\,(a_{\Psi_1} \vee a_{\Psi_2})\,\mathsf{U}\,(\,(z \in J) \wedge a_{\Psi_2})\,)\big)$;

    end switch

(* add a_Ψ to the labeling of all state regions where Ψ holds *)

    forall $s \in S_{rts}$ with $s\{z := 0\} \in Sat_R(\Psi)$ **do** $L_{rts}(s) := L_{rts}(s) \cup \{\, a_\Psi \,\}$ **od**;

  od
od
if $I_{rts} \subseteq Sat_R(\Phi)$ **then return** "yes" **else return** "no" **fi**

Theorem 9.69. Complexity of TCTL Model Checking

The TCTL model-checking problem is PSPACE*-complete.*

The proof of the PSPACE-completeness is not provided here; it can be found in [9].

9.4 Summary

- Timed automata are program graphs in which clock variables are used to measure the elapse of time. Guards on edges determine when an edge may be taken, whereas (location) invariants describe how long the system may reside in a location.
- A timed automaton describes an infinite transition system, and thus is a model to finitely model a continuous-time system.
- A path of a timed automaton is time-divergent if time may pass without bound. A timed automaton is timelock-free whenever, from all states in its transition system, at least one time-divergent path can start. That is, all its states satisfy $\exists\Box$true (under the TCTL semantics).
- A path of a timed automaton is zeno if it is not time-divergent and infinitely many actions are performed on it. A non-zeno timed automaton does not have initial zeno paths. A sufficient criterion for non-zenoness is that in all control cycles at least a single time unit elapses.
- Timed CTL is a real-time variant of CTL in which clock constraints may act as atomic propositions and the until operator is equipped with a time interval. The semantics of universal and existential path formulae is defined in terms of time-divergent paths. This treatment is similar to fair CTL where only fair paths are considered.
- Model-checking whether timed automaton TA satisfies TCTL formula Φ can be reduced to checking a CTL formula derived from Φ on a finite transition system (determined by both TA and Φ), the so-called region transition system. The key to this concept is a clock equivalence relation, a bisimulation relation on propositions and clock constraints.
- The size of the region transition system is exponential in the number of clocks and the constants with which clocks are compared.
- A timed automaton is timelock-free whenever its region transition system does not have reachable terminal states.

9.5 Bibliographic Notes

Timed automata. The distinction between instantaneous activities (like changing a state) and delays originates from real-time process algebras such as timed CSP [366], timed CCS [427], and ATP [315]. The resulting two-phase behavior in which discrete phases – a state change due to some activity – and continuous phases – passage of time – alternate has been adopted in timed automata. Timed automata have been introduced by Alur and Dill [10] (the journal version is published as [11]), Dill [131] and H. R. Lewis [271]. Henzinger et al. [200] introduced the idea of invariants and called this safety timed automata. Extensions of timed automata include, e.g., timed automata with deadlines (as opposed to invariants) by Bornot and Sifakis [58], timed automata with drifting clocks – associating an interval to each clock that specifies the relative speed with respect to an exact reference clock – by Olivero, Sifakis, and Yovine [316], and hybrid automata for describing a mixture of discrete and continous behavior of a more general nature by Maler, Manna, and Pnueli [281]. D'Argenio and Brinksma [111] defined a process algebra for describing safety timed automata.

Model-checking timed automata. The decidability of model-checking timed CTL has been shown by Alur, Courcoubetis, and Dill [9], using clock equivalence and region transition systems. The symbolic manipulation of timed automata by means of sets of linear inequations over pairs of clocks has been brought up by Henzinger et. al. [200]. Bouyer has shown that symbolic backward reachability in the setting where timed automata are equipped with clock difference constraints (i.e., $x - y \prec c$) calls for a special treatment [61]. Survey papers on model-checking timed automata have been provided by Bengtsson and Yi [43] and Yovine [430]. Time-abstract bisimulation originates from Larsen and Yi [265] who studied this equivalence in the context of a real-time process algebra. Decidability of time-abstract bisimulation has been shown by Cerans [78]. Tripakis and Yovine [394] introduced the use of time-abstract bisimulation for timed automata verification. In the journal paper [395], these authors describe linear-time (using timed Büchi automata) and branching-time (timed CTL) verification algorithms based on time-abstract bisimulation. They also present a quotienting algorithm with respect to time-abstract bisimulation. Difference bound matrices have been proposed for timed automata by Dill [131]. Berthomieu and Menasche [48] have used these data structures for the analysis of time Petri nets and Bellman [40] exploited these structures for constraint graphs. Larsen et al. [263] introduced a symbolic data structure akin to binary decision diagrams for dealing with zones in a compact manner. The edited volume [294] contains an overview of techniques and tools for the verification of real-time systems.

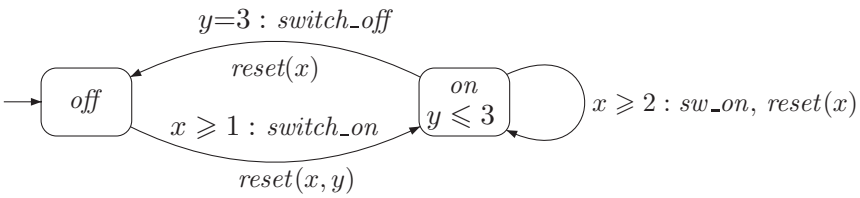
Linear-time properties. Timed automata with final locations accept timed languages, i.e., sets of infinite sequences of pairs consisting of a symbol and a real value representing its time of occurrence. Alur and Dill [11] proved that the class of languages accepted

by timed automata is not closed under complementation, and hence no simple logical characterization of this class exists. Alur and Dill [12] present an algorithm to check the emptiness of the language accepted by a timed automaton. Alur, Fix, and Henzinger [13] proposed event-recording automata which are closed under all Boolean operations, including complementation. An event-recording automaton is a timed automaton that contains for every event (i.e., action) a clock that records the time of the last occurrence of the event. Asarin, Caspi, and Maler [20] defined *timed regular expressions* and proved that, à la Kleene's theorem for finite automata, its expressive power is equivalent to timed automata (without location invariants). Intersection and renaming are essential operators for timed regular expressions to obtain this result. As in classical automata theory, the construction of automata from expressions is rather straightforward, but the reverse direction is much more involved. The authors extend their results also to timed ω-regular expressions.

Model checkers for timed automata. Behrmann, Larsen, Petterson, and Yi and colleagues have developed the model checker UPPAAL [35] since the mid-nineties. It supports a safety fragment of timed CTL, and allows bounded liveness properties to be checked using test automata. Around the same time, Yovine *et al.* developed the model checker KRONOS [429] which supports timed CTL. During recent years, UPPAAL has been extended with branch-and-bound algorithms for dealing with priced timed automata where locations are equipped with a cost rate [353]. Wang [417] has developed a timed automata model checker using a symbolic data structure akin to binary decision diagrams.

9.6 Exercises

EXERCISE 9.1. Consider the timed automaton *LightSwitch* illustrated below.



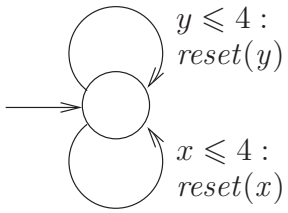
Questions:

(a) Determine the transition system $TS(LightSwitch)$.

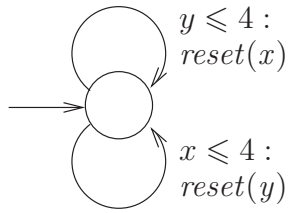
(b) Determine the region transition system $RTS(LightSwitch, \text{true})$.

(c) Check whether *LightSwitch* is timelock-free and non-zeno.

EXERCISE 9.2. Consider the following two timed automata:



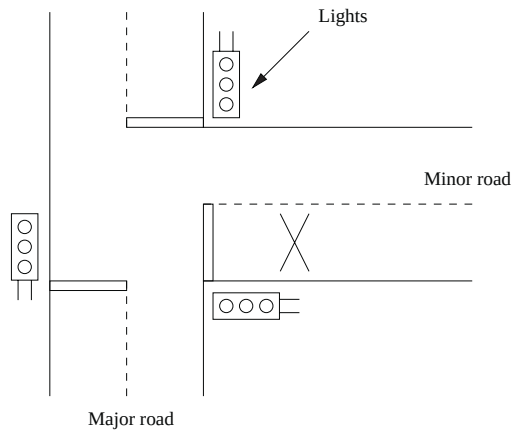
(a)



(b)

As these timed automata have a single location only, the *state* of these timed automata can be considered as just a point in the real plane. A point (d, e) (with $d, e \geqslant 0$) thus represents that clock x has value d and clock y has value e. Determine the reachable state space of each of these timed automata. Justify your answers.

EXERCISE 9.3. (Modeling exercise). A control system must ensure the safe and correct functioning of a set of traffic lights at a T-junction between a major and a minor road. The lights will be set on green on the major road and red on the minor road unless a vehicle is detected by a sensor in the road just before the lights on the minor road. In this case the lights will be switchable in the standard manner and allow traffic to leave the minor road. After a suitable interval the lights will revert to their default position to allow traffic to flow on the major road again. Once a vehicle is detected the sensor will be disabled until the minor-road lights are set to red again. A sketch of the T-junction is provided below.



Questions:

1. First we ignore all timing issues involved and concentrate on the qualitative aspects of the behavior of the traffic lights. Model the above system as a network of (timed) automata. For convenience, you may assume that the two major-road lights are fully synchronized and can be modeled as a single light. Complement your system model by adding a process that regulates the arrival of cars in the minor road.

2. Adapt your model so as to incorporate the following timing constraints. Deal with each timing constraint separately so as reduce the complexity. Indicate for each timing constraint the necessary adaptations to your un-timed model:

   (a) a minor-road light stays on green for 30 seconds,

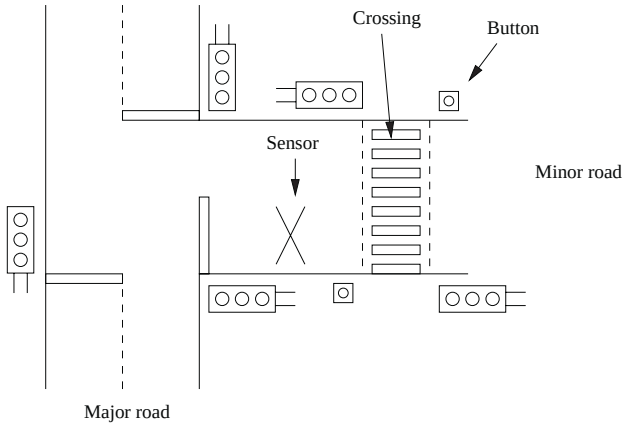
   (b) all interim lights stay on for 5 seconds,

   (c) there is a 1 second delay between switching one light off and another on (e.g., switching from red to yellow),

   (d) the major-road lights must be on green for at least 30 seconds in each cycle,

   (e) (more involved) but must respond to the sensor immediately after that.

3. We extend the T-junction in the following way. Suppose there is a pedestrian crossing a short distance down the minor road but beyond the sensor. There is a button on each side of the road for pedestrians to indicate they wish to cross. The crossing should only allow people to cross when the "minor lights" are set to red in order to minimize waiting times for traffic on the minor road. The new situation is sketched below.

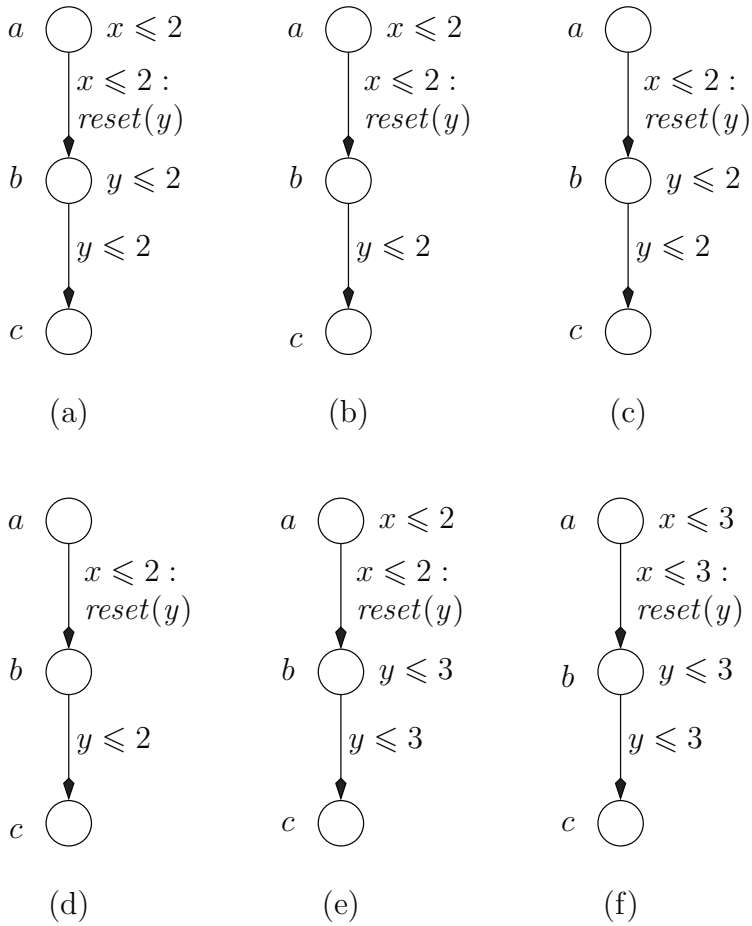


   Extend your timed model of the previous question in order to cope with this new situation.

4. Does the crossing indeed only allow pedestrians to cross when the "minor lights" are set to red?

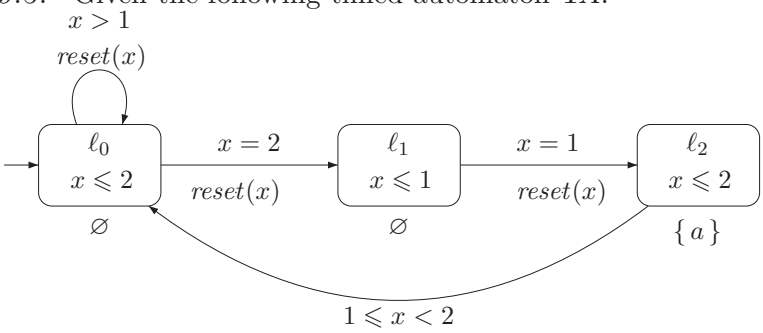
(This exercise is inspired by [151].)

Exercise 9.4. Consider the following six timed automata:



Give for each individual timed automaton a TCTL formula that distinguishes this timed automaton from all other ones. It is allowed to use only the atomic propositions a, b, and c and clock constraints in the TCTL formulae. Location identifiers are not allowed. (This exercise is due to Pedro D'Argenio.)

EXERCISE 9.5. Given the following timed automaton TA:



Questions:

(a) Determine the transition system $TS(TA)$.

(b) Determine the set of states $Sat(\exists \Diamond^{<4}\ a)$.

(c) Determine the region transition system $RTS(TA, \text{true})$.

Chapter 10

Probabilistic Systems

Whereas model-checking techniques focus on the absolute guarantee of correctness — "it is impossible that the system fails" — in practice such rigid notions are hard, or even impossible, to guarantee. Instead, systems are subject to various phenomena of a stochastic nature, such as message loss or garbling and the like, and correctness — "with 99% chance the system will not fail" — is becoming less absolute. This chapter considers the automated verification of *probabilistic* systems, i.e., systems that exhibit probabilistic aspects[1]. Probabilistic aspects are essential for, among others:

- Randomized algorithms. Typical examples are distributed algorithms like leader election or consensus algorithms where coin-tossing experiments are used to break the "symmetry" between the processes such that, e.g., consensus will eventually be reached with probability 1.
- Modeling unreliable and unpredictable system behavior. Phenomena like message loss, processor failure, and the like may be modeled by nondeterminism. This is often appropriate in early system design phases where systems are considered at a high level of abstraction and where information about the likelihood is (sometimes deliberately) left unspecified. In later design stages, though, where the internal system characteristics become more dominant, probabilities are a useful vehicle to quantify and thus refine this information.
- Model-based performance evaluation. As performance evaluation is aimed at forecasting system performance and dependability, probabilistic information — what is

[1]Note: verifying probabilistic systems should not be confused with *probabilistic verification*, a model-checking technique that is based on a partial state-space exploration.

the distribution of the message transmission delay or what is the failure rate of a processor? — needs to be present in order to evaluate *quantitative* properties like waiting time, queue length, time between failure, and so on.

In order to model random phenomena, transition systems are enriched with probabilities. This can be done in different ways. In *discrete-time Markov chains* (MCs), all choices are probabilistic. Markov chains are the most popular operational model for the evaluation of performance and dependability of information-processing systems. Roughly speaking, Markov chains are transition systems with probability distributions for the successors of each state. That is, instead of a nondeterministic choice, the next state is chosen probabilistically. Markov chains are not appropriate for modeling randomized distributed systems, since they cannot model the interleaving behavior of the concurrent processes in an adequate manner. For this purpose, *Markov decision processes* (MDPs) are used. In MDPs, both nondeterministic and probabilistic choices coexist. Put in a nutshell, MDPs are transition systems in which in any state a nondeterministic choice between probability distributions exists. Once a probability distribution has been chosen nondeterministically, the next state is selected in a probabilistic manner—as in MCs. Any MC is thus an MDP in which in any state the probability distribution is uniquely determined. Randomized distributed algorithms are typically appropriately modeled by MDPs, as probabilities affect just a small part of the algorithm and nondeterminism is used to model concurrency between processes by means of interleaving.

The verification of probabilistic systems can be focused on either *quantitative properties* or *qualitative properties* (or both). Quantitative properties typically put constraints on the probability or expectation of certain events. Instances of quantitative properties are, e.g., the requirement that the probability for delivering a message within the next t time units is at least 0.98, or that the expected number of unsuccessful attempts to find a leader in a concurrent system is at most seven. Qualitative properties, on the other hand, typically assert that a certain (good) event will happen almost surely, i.e., with probability 1, or dually, that a certain (bad) event almost never occurs, i.e., with zero probability. Typical qualitative properties for Markov models are reachability, persistence (does eventually an event always hold?), and repeated reachability (can certain states be repeatedly reached?).

The purpose of this chapter is to present the main verification principles for qualitative and quantitative properties of MCs and MDPs. These range from techniques to analyze reachability, persistence, and repeated reachability properties, to model-checking algorithms for a probabilistic variant of CTL, called *Probabilistic Computation Tree Logic*, PCTL for short. This logic is appropriate for expressing a large class of properties in a rather elegant manner. For instance, the property "eventually a leader will be elected with probability at least $\frac{4}{5}$" for a randomized leader election protocol is expressed in PCTL by

$$\mathbb{P}_{\geqslant 0.8}(\Diamond \mathit{leader}).$$

Alternatively,

$$\mathbb{P}_{\leqslant 0.015}(\neg c.\mathit{empty}\ \mathsf{U}^{\leqslant 6} c.\mathit{full})$$

asserts that the probability of a channel c being fully occupied within the next six steps, while in all intermediate configurations c is nonempty, is bounded above by 0.015.

Apart from branching-time properties in PCTL, this chapter also covers linear-time properties. As opposed to PCTL where probabilities are expressed syntactically by means of the $\mathbb{P}$ operator, in the linear-time setting probabilistic concepts only appear on the semantic level. That is, in the probabilistic linear time setting, LTL formulae serve to specify the desired or bad behaviors and the aim is to establish the probability that a given LTL formula holds. Thus, the satisfaction relation over states is no longer Boolean—a formula holds for a state or not—but assigns probability values to states. This chapter deals with the verification of linear-time properties such as regular safety properties and ω-regular safety properties.

In line with the rest of this monograph, we adopt a *state-based* view of probabilistic models. This means that Markov chains and Markov decision processes are treated as variants of directed graphs (read: transition systems) where the edges (i.e., transitions) are augmented with randomness information. This is in contrast to many textbooks where MCs are defined as sequences of random variables. The state-based approach with atomic propositions for the states appears to be more natural, when viewing MCs and MDPs as operational models for reactive systems and structures for temporal logics (as we do). As compositional approaches for Markov models are outside the scope of this monograph, actions are irrelevant in this chapter and are therefore omitted.

10.1 Markov Chains

Markov chains behave as transition systems with the only difference that nondeterministic choices among successor states are replaced by probabilistic ones. That is to say, the successor state of state s, say, is chosen according to a probability distribution. This probability distribution only depends on the current state s, and not on, e.g., the path fragment that led to state s from some initial state. Accordingly, the system evolution does not depend on the history (i.e., the path fragment that has been executed so far), but only on the current state s. This is known as the *memoryless property*.

Definition 10.1. (Discrete-Time) Markov Chain (MC)

A *(discrete-time) Markov chain* is a tuple $\mathcal{M} = (S, \mathbf{P}, \iota_{\text{init}}, AP, L)$ where

- S is a countable, nonempty set of states,
- $\mathbf{P} : S \times S \to [0,1]$ is the *transition probability function* such that for all states s:

$$\sum_{s' \in S} \mathbf{P}(s, s') = 1,$$

- $\iota_{\text{init}} : S \to [0,1]$ is the *initial distribution*, such that $\sum_{s \in S} \iota_{\text{init}}(s) = 1$, and
- AP is a set of atomic propositions and $L : S \to 2^{AP}$ a labeling function.

$\mathcal{M}$ is called *finite* if S and AP are finite. For finite $\mathcal{M}$, the *size* of $\mathcal{M}$, denoted *size*$(\mathcal{M})$, is the number of states plus the number of pairs $(s, s') \in S \times S$ with $\mathbf{P}(s, s') > 0$. ■

The transition probability function $\mathbf{P}$ specifies for each state s the probability $\mathbf{P}(s, s')$ of moving from s to s' in one step, i.e., by a single transition. The constraint imposed on $\mathbf{P}$ ensures that $\mathbf{P}$ is a distribution. For the mathematical treatment of Markov chains, it is irrelevant whether the nonzero transition probabilities are rational or not. However, for algorithmic purposes the values $\mathbf{P}(s, s')$ are supposed to be rational for all states s, $s' \in S$.

The value $\iota_{\text{init}}(s)$ specifies the probability that the system evolution starts in state s. The states s with $\iota_{\text{init}}(s) > 0$ are considered as the *initial states*. In a similar way, the states s' for which $\mathbf{P}(s, s') > 0$ are viewed as the possible successors of s. For state s and $T \subseteq S$, let $\mathbf{P}(s, T)$ denote the probability of moving from s to some state $t \in T$ in a single step. That is,

$$\mathbf{P}(s, T) \ = \sum_{t \in T} \mathbf{P}(s, t).$$

In the sequel, we often identify the transition probability function $\mathbf{P} : S \times S \to [0,1]$ with the matrix $\big(\, \mathbf{P}(s,t) \,\big)_{s,t \in S}$. The row $\mathbf{P}(s, \cdot)$ for state s in this matrix contains the probabilities of moving from s to its successors, while the column $\mathbf{P}(\cdot, s)$ for state s specifies the probabilities of entering state s from any other state. Similarly, the initial distribution ι_{init} is often viewed as a vector $\big(\, \iota_{\text{init}}(s) \,\big)_{s \in S}$.

The use of atomic propositions and the labeling function L is the same as for transition systems. In the remainder of this chapter, we will often treat the state names as atomic propositions, i.e., $AP = S$ and $L(s) = \{\, s \,\}$.

A Markov chain induces an *underlying digraph* where states act as vertices and there is an edge from s to s' if and only if $\mathbf{P}(s, s') > 0$. *Paths* in Markov chains are maximal (i.e.,

infinite) paths in the underlying digraph. They are defined as infinite state sequences $\pi = s_0\, s_1\, s_2 \cdots \in S^\omega$ such that $\mathbf{P}(s_i, s_{i+1}) > 0$ for all $i \geqslant 0$. For path π in $\mathcal{M}$, $\inf(\pi)$ denotes the set of states that are visited infinitely often in π. For finite Markov chains, $\inf(\pi)$ is nonempty for all paths π.

Markov chains are depicted by their underlying digraph where edges are equipped with the transition probabilities in]0,1]. If a state s has a unique successor s', i.e., $\mathbf{P}(s, s') = 1$, the transition probability may be omitted.

Example 10.2. A Simple Communication Protocol

Consider a simple communication protocol operating with a channel. It is error-prone in the sense that messages may be lost, see the Markov chain depicted in Figure 10.1. Here, $\iota_{\text{init}}(\mathit{start}) = 1$ and $\iota_{\text{init}}(s) = 0$ for $s \neq \mathit{start}$, i.e., *start* is the unique initial state. In the state *start*, a message is generated that is senf off along the channel in its unique successor state *try*. The message is lost with probability $\frac{1}{10}$, in which case the message will be sent off again, until it is eventually delivered. As soon as the message has been delivered correctly, the system returns to its initial state.

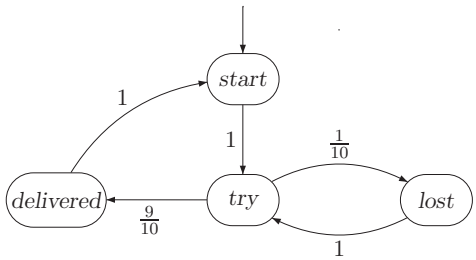


Figure 10.1: Markov chain for a simple communication protocol.

Using the enumeration *start*, *try*, *lost*, *delivered* for the states, the transition probability function $\mathbf{P}$ viewed as a 4×4 matrix and the initial distribution viewed as a column vector are

$$\mathbf{P} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{10} & \frac{9}{10} \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \qquad \iota_{\text{init}} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

An example of a path is

$$\pi = (\mathit{start}\ \mathit{try}\ \mathit{lost}\ \mathit{try}\ \mathit{lost}\ \mathit{try}\ \mathit{delivered})^\omega.$$

Along this path each message has to be retransmitted two times before delivery. It follows that $\inf(\pi) = S$. For $T = \{\mathit{lost}, \mathit{delivered}\}$, we have $\mathbf{P}(\mathit{try}, T) = 1$. ■

Example 10.3. Simulating a Die by a Fair Coin

Consider simulating the behavior of a standard six-sided die by a fair coin, as originally proposed by Knuth and Yao [242], see the Markov chain depicted in Figure 10.2.

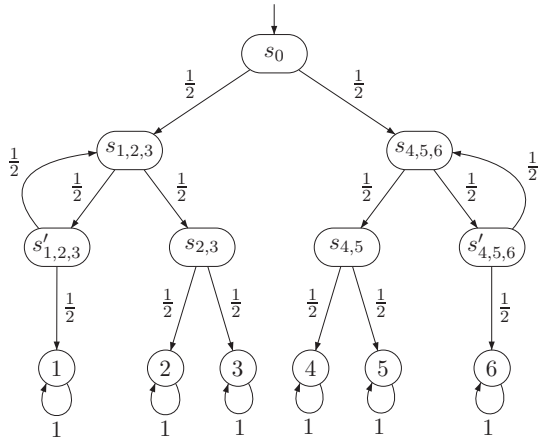


Figure 10.2: Markov chain for simulating a die by a fair coin.

The computation starts in the initial state s_0, i.e., we have $\iota_{\text{init}}(s_0) = 1$ and $\iota_{\text{init}}(s) = 0$ for all states $s \neq s_0$. The states 1, 2, 3, 4, 5, and 6 at the bottom stand for the possible die outcomes. Each inner node stands for tossing a fair coin. If the outcome is *heads*, the left branch determines the next state; if the outcome is *tails*, the right branch determines the next state.

If the coin-tossing experiment in state s_0 yields *heads*, the system moves to state $s_{1,2,3}$. Tossing the coin again leads with equal probability to either state $s_{2,3}$ (from which the die-outcomes 2 or 3 are possible with equal probability) or to state $s'_{1,2,3}$. From the latter state, a coin flipping yields with probability $\frac{1}{2}$ the outcome 1, or with probability $\frac{1}{2}$ a return to state $s_{1,2,3}$. The behavior for outcome *tails* in the initial state is symmetric. We will establish later that, in fact, this Markov chain indeed adequately models a die, i.e., the outcomes are equally likely. ■

Example 10.4. The Craps Gambling Game

The game craps is based on betting on the outcome of the roll of two dice. The outcome of the first roll—the "come-out" roll—determines whether there is a need for any further rolls. On outcome 7 or 11, the game is over and the player wins. The outcomes 2, 3, or 12, however, are "craps"; the player loses. On any other outcome, the dice are rolled again, but the outcome of the come-out roll is remembered (the "point"). If the next roll yields

7 or the point, the game is over. On 7, the player loses, on point the player wins. In any other case, the dice are rolled until eventually either 7 or the point is obtained. Figure 10.3 depicts the Markov chain that describes the behavior of the craps game. State *start* is the unique initial state. We have $\mathbf{P}(\mathit{start}, \mathit{won}) = \frac{2}{9}$ as there are eight combinations of the come-out roll that are successful: (1,6), (2,5), (3,4), (5,6) and all symmetric counterparts. The other transition probabilities are determined in a similar way. The self-loops at the states 4, 5, 6, 8, 9, and 10 model the rerolling of the dice. For $T = \{4, 5, 6\}$, we have $\mathbf{P}(s_0, T) = \frac{1}{3}$. ■

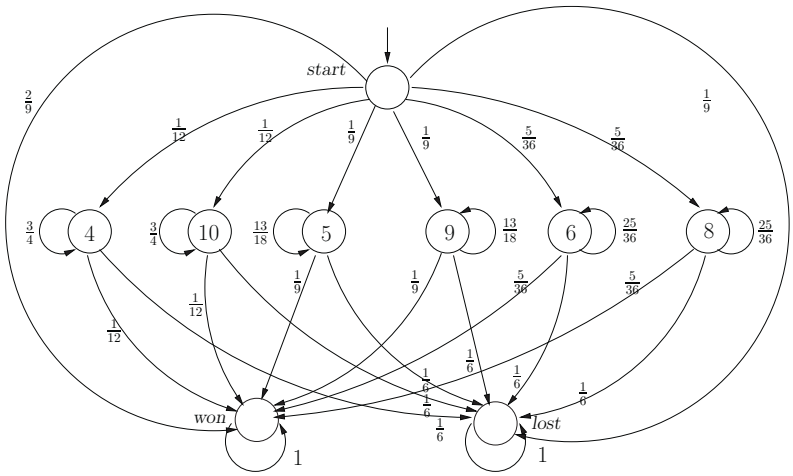


Figure 10.3: Markov chain for the behavior of the craps game.

Example 10.5. Zeroconf Protocol

The IPv4 zeroconf protocol is designed for a home local network of appliances (microwave oven, laptop, VCR, DVD player, etc.) each of which is supplied with a network interface to enable mutual communication. Such adhoc networks must be hot-pluggable and self-configuring. Among others, this means that when a new appliance (interface) is connected to the network, it must be configured with a *unique* IP address automatically. The zeroconf protocol solves this task in the following way. A host that needs to be configured randomly selects an IP address, U say, out of the 65,024 available addresses and broadcasts a message (called *probe*) saying "Who is using the address U?" If the probe is received by a host that is already using the address U, it replies by a message indicating that U is in use. After receiving this message the host to be configured will restart: it randomly selects a new address, broadcasts a probe, etc.

Due to message loss or a busy host, a probe or a reply message may not arrive at some (or all) other hosts. In order to increase the reliability of the protocol, a host is required

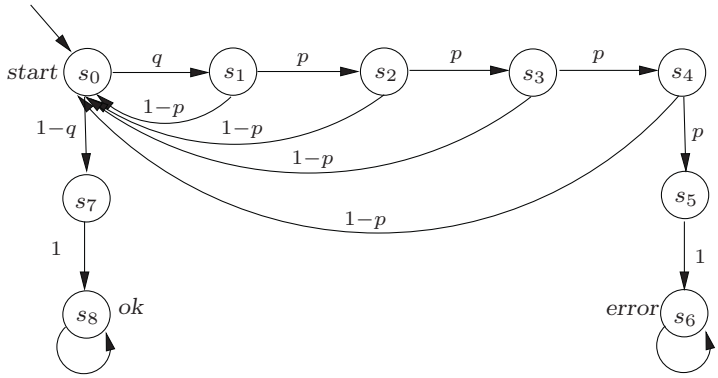


Figure 10.4: Markov chain of the IPv4 zeroconf protocol (for n=4 probes).

to send n probes, each followed by a listening period of r time units. Therefore, the host can start using the selected IP address only after n probes have been sent and no reply has been received during $n{\cdot}r$ time units. Note that after running the protocol a host may still end up using an IP address already in use by another host, e.g., because all probes were lost. This situation, called *address collision*, is highly undesirable since it may force a host to kill active TCP/IP connections.

The protocol behavior of a single host is modeled by a Markov chain consisting of $n{+}5$ states (see Figure 10.4 for $n = 4$) where n is the maximal number of probes needed (as above). The initial state is s_0 (labeled *start*). In state s_{n+4} (labeled *ok*) the host finally ends up with an unused IP address; in state s_{n+2} (labeled *error*) it ends up with an address that is already in use, i.e., an address collision. State s_i ($0 < i \leqslant n$) is reached after issuing the ith probe. In state s_0 the host randomly chooses an IP address. With probability $q = m/65024$, where m is the number of hosts in the network when connecting the host to the network, this address is already in use. With probability $1{-}q$ the host chooses an unused address and ends up in state s_{n+3}. Then it issues $n{-}1$ probes and waits $n{\cdot}r$ time units before using this address. (The sending of these probes and the waiting time are abstracted from in the MC.) If the chosen IP address is already in use, state s_1 is reached. Now two situations are possible. With probability p, no reply is received during r time units (as either the probe or its reply has been lost), and a next probe is sent, resulting in state s_2. If, however, a reply has arrived in time, the host returns to the initial state and restarts the protocol. The behavior in state s_i ($2 \leqslant i < n$) is similar. If in state s_n, however, no reply has received within r time units after sending the nth probe, an address collision occurs. ■

We adopt the notions of direct successor and direct predecessor from transition systems. Let $Paths(\mathcal{M})$ denote the set of paths in $\mathcal{M}$, and $Paths_{fin}(\mathcal{M})$ denote the set of finite

path fragments $s_0\, s_1 \ldots s_n$ where $n \geqslant 0$ and $\mathbf{P}(s_i, s_{i+1}) > 0$ for $0 \leqslant i < n$. $\mathit{Paths}(s)$ denotes the set of all paths in $\mathcal{M}$ that start in state s. Similarly, $\mathit{Paths}_{\mathit{fin}}(s)$ denotes the set of all path fragments $s_0\, s_1 \ldots s_n$ such that $s_0 = s$. The set of direct successors and direct predecessors are defined as follows. Let $\mathit{Post}(s)$ denote the set of successors of s, i.e., $\mathit{Post}(s) = \{\, s' \in S \mid \mathbf{P}(s, s') > 0 \,\}$. Similarly, $\mathit{Pre}(s) = \{\, s' \in S \mid \mathbf{P}(s', s) > 0 \,\}$. $\mathit{Post}^*(s)$ denotes the set of all states that are reachable from s via a finite path fragment and $\mathit{Pre}^*(s) = \{\, s' \in S \mid s \in \mathit{Post}^*(s') \,\}$. For $B \subseteq S$, let

$$\mathit{Post}^*(B) = \bigcup_{s \in B} \mathit{Post}^*(s) \text{ and } \mathit{Pre}^*(B) = \bigcup_{s \in B} \mathit{Pre}^*(s).$$

Notation 10.6. Absorbing State

State s of MC $\mathcal{M}$ is called *absorbing* if $\mathit{Post}^*(s) = \{\, s \,\}$. Since $\mathbf{P}$ is a stochastic matrix, i.e., each row sum equals 1, s is an absorbing state if and only if $\mathbf{P}(s, s) = 1$, and $\mathbf{P}(s, t) = 0$ for all states $t \neq s$. ■

Remark 10.7. Discrete-Time Markov Chains

Markov chains as in Definition 10.1 are often called *discrete-time Markov chains*. This has mainly historical reasons. In many cases, Markov chains are used as a *time-abstract* model, like transition systems provide a time-abstract operational model for reactive systems. In fact, as for transition systems, a timed interpretation of Markov chains is only adequate when the underlying time domain is discrete and each transition is assumed to take a single time unit. ■

The underlying graph of a Markov chain forms the basis for reasoning about qualitative properties, such as LTL or CTL formulae. To that end, define for MC $\mathcal{M} = (S, \mathbf{P}, \iota_{\text{init}}, AP, L)$ the transition system $TS(\mathcal{M}) = (S, \{\, \tau \,\}, \rightarrow, I, AP, L)$ where $I = \{\, s \in S \mid \iota_{\text{init}}(s) > 0 \,\}$ and $s \xrightarrow{\tau} t$ iff $\mathbf{P}(s, t) > 0$. LTL or CTL formulae abstract away from the probabilities; e.g., the CTL formula $\exists\Diamond a$ may hold even if it is very unlikely that a state where a holds will eventually be visited. For instance, for the communication protocol of Example 10.2 (page 749), the event to reach state *delivered* within the next 100 steps, formalized in LTL by $\varphi = \bigvee_{0 \leqslant i \leqslant 100} \bigcirc^i \mathit{delivered}$, does not hold for the starting state, but gives no quantitative information about the fraction of paths for which φ is violated. In fact, the chance that φ does not hold is extremely small (namely 10^{-50}), and hence, might be classified as negligible. It even holds that

$$\mathit{start} \models \exists\Box\neg \mathit{delivered},$$

although it seems to be impossible that a message never will be delivered. In fact, the event $\Box\neg\mathit{delivered}$ holds with zero probability, as we will see later.

The goal is now to specify and check *quantitative properties*. Such properties may, e.g., assert that the probability of reaching a certain bad state is sufficiently small, or that the probability of achieving a certain desired system behavior is above a given threshold. *Qualitative properties* arise as a special case of quantitative properties where the probability bounds are the trivial bounds zero or 1. That is, typical qualitative properties require that the probability of reaching a bad state is zero, or dually, that a certain desired system behavior appears with probability 1.

Although Markov chains yield a rather intuitive probabilistic model, reasoning about quantitative or qualitative properties requires a formalization of the probabilities for sets of paths. This formalization is based on measure theory; in particular probability spaces and σ-algebras. Readers familiar with the main concepts of this field may skip the following short summary of the main concepts. Further details can be found, e.g., in [21, 150, 319].

Excursus on Probability Spaces A *σ-algebra* is a pair $(Outc, \mathfrak{E})$ where $Outc$ is a nonempty set and $\mathfrak{E} \subseteq 2^{Outc}$ a set consisting of subsets of $Outc$ that contains the empty set and is closed under complementation and countable unions, i.e.,

- $\varnothing \in \mathfrak{E}$,
- if $E \in \mathfrak{E}$, then $\overline{E} = Outc \setminus E \in \mathfrak{E}$,
- if $E_1, E_2, \ldots \in \mathfrak{E}$, then $\bigcup\limits_{n \geqslant 1} E_n \in \mathfrak{E}$.

Note that the conditions of σ-algebras yield that $Outc \in \mathfrak{E}$, as $Outc = \overline{\varnothing}$, and that $\mathfrak{E}$ is closed under countable intersections, as

$$\bigcap_{n \geqslant 0} E_n \;=\; \overline{\bigcup_{n \geqslant 0} \overline{E_n}}.$$

Occasionally, the set $Outc$ is supposed to be fixed and $\mathfrak{E}$ is called a σ-algebra. The elements of $Outc$ are often called *outcomes*, while the elements of $\mathfrak{E}$ are called *events*.

For any set $Outc$, the powerset $\mathfrak{E} = 2^{Outc}$ yields a σ-algebra over $Outc$. In this σ-algebra, all subsets of $Outc$ are events. The other extreme is the σ-algebra consisting of the empty set and $Outc$, i.e., $\mathfrak{E} = \{\, \varnothing, Outc \,\}$. Here, no nonempty proper subset of $Outc$ is an event.

A *probability measure* on $(Outc, \mathfrak{E})$ is a function $Pr : \mathfrak{E} \to [0,1]$ such that $Pr(Outc) = 1$, and if $(E_n)_{n \geqslant 1}$ is a family of pairwise disjoint events $E_n \in \mathfrak{E}$, then:

$$Pr\Big(\bigcup_{n \geqslant 1} E_n\Big) \;=\; \sum_{n \geqslant 1} Pr(E_n).$$

A *probability space* is a σ-algebra equipped with a probability measure, i.e., it is a triple $(Outc, \mathfrak{E}, Pr)$ where $(Outc, \mathfrak{E})$ is a σ-algebra and Pr a probability measure on $(Outc, \mathfrak{E})$. The value $Pr(E)$ is called the *probability measure* of E, or simply the probability of E. In the context of probability measures, the events (i.e., the elements of $\mathfrak{E}$) are often said to be *measurable*. That is, measurability of a set $E \subseteq Outc$ means that $E \in \mathfrak{E}$, and hence, it makes sense to speak about the probability measure of E.

Example 10.8. Tossing a Fair Coin

Consider the experiment in which a fair coin is tossed once. The possible outcomes are *heads* and *tails*, i.e., $Outc = \{ heads, tails \}$. For the events to be considered let us assume the singleton events $\{ heads \}$ and $\{ tails \}$ suffice. The smallest σ-algebra that contains these events is the powerset of $\{ heads, tails \}$. So, let $\mathfrak{E} = 2^{Outc}$. Since the coin is assumed to be fair, the probability measure Pr is given by

$$Pr(\varnothing) = 0, \quad Pr(\{ heads \}) = Pr(\{ tails \}) = \tfrac{1}{2}, \quad Pr(\{ heads, tails \}) = 1.$$

■

In general, whenever $Outc$ is countable, then a probability measure on the powerset of $Outc$ can be obtained by fixing a function $\mu : Outc \to [0, 1]$ such that

$$\sum_{e \in Outc} \mu(e) \;=\; 1.$$

Such functions μ are called *distributions* on $Outc$. Any distribution μ induces a probability measure on the σ-algebra $\mathfrak{E} = 2^{Outc}$ in the following way. For subset E of $Outc$, $Pr_\mu(E)$ is defined by $\sum_{e \in E} \mu(e)$. In fact, it is easy to check that μ satisfies the conditions of a probability measure. In the sequel, $Pr_\mu(E)$ is often abbreviated by $\mu(E)$ and $Distr(Outc)$ is used to denote the set of distributions on $Outc$.

Let us summarize some fundamental properties of probability measures. Since $E \cup \overline{E} = Outc$ and E and $\overline{E}$ are disjoint, the above conditions imply

$$Pr(\overline{E}) \;=\; 1 - Pr(E).$$

In particular, $Pr(\varnothing) = Pr(\overline{Outc}) = 1 - Pr(Outc) = 1 - 1 = 0$. Probability measures are *monotonic*, i.e., for events E and E' such that $E \subseteq E'$ it holds that

$$Pr(E') \;=\; Pr(E) + Pr(E' \setminus E) \;\geqslant\; Pr(E).$$

Furthermore, if $(E_n)_{n \geqslant 1}$ is a family of events, possibly not pairwise disjoint, then: $\bigcup_{n \geqslant 1} E_n = \bigcup_{n \geqslant 1} E'_n$ where $E'_1 = E_1$ and $E'_n = E_n \setminus (E_1 \cup \ldots \cup E_{n-1})$ for $n \geqslant 2$. Since $E'_n \cap E'_m = \varnothing$

if $n \neq m$ we get

$$Pr(\bigcup_{n\geqslant 1} E_n) \;=\; Pr(\bigcup_{n\geqslant 1} E'_n) \;=\; \sum_{n\geqslant 1} Pr(E'_n).$$

If $E_1 \subseteq E_2 \subseteq E_3 \subseteq \ldots$ and E'_n is as above, then we have $E'_n = E_n \setminus E_{n-1}$ for $n \geqslant 2$, which yields

$$\begin{aligned} Pr(\bigcup_{n\geqslant 1} E_n) &= \overbrace{Pr(E_1)}^{=Pr(E'_1)} + \sum_{n=2}^{\infty} \overbrace{(Pr(E_n) - Pr(E_{n-1}))}^{=Pr(E'_n)} \\ &= \lim_{N\to\infty} \left(Pr(E_1) + \sum_{n=2}^{N} (Pr(E_n) - Pr(E_{n-1})) \right) \\ &= \lim_{N\to\infty} Pr(E_N). \end{aligned}$$

Note that this limit exists and agrees with the supremum of $\{ Pr(E_1), Pr(E_2), \ldots \}$, as the monotonicity of Pr yields $Pr(E_1) \leqslant Pr(E_2) \leqslant \ldots \leqslant 1$. For countable intersections analogous results apply. That is, if $E_1 \supseteq E_2 \supseteq \ldots$, then

$$Pr(\bigcap_{n\geqslant 1} E_n) \;=\; \lim_{n\to\infty} Pr(E_n) \;=\; \inf_{n\geqslant 1} Pr(E_n).$$

This follows from the fact that the sequence $(\,\overline{E_n}\,)_{n\geqslant 1}$ of the complements $\overline{E_n} = Outc \setminus E_n$ is decreasing. Thus, the above yields

$$\begin{aligned} Pr(\bigcap_{n\geqslant 1} E_n) &= Pr(\overline{\bigcup_{n\geqslant 1} \overline{E_n}}) \\ &= 1 - Pr(\bigcup_{n\geqslant 1} \overline{E_n}) \\ &= 1 - \lim_{n\to\infty} Pr(\overline{E_n}) \\ &= 1 - \lim_{n\to\infty} (1 - Pr(E_n)) \\ &= \lim_{n\to\infty} Pr(E_n). \end{aligned}$$

Any event E with $Pr(E) = 1$ is said to hold *almost surely*. Note that if E holds almost surely, then $Pr(D) = Pr(E \cap D)$ for all events D as $Pr(D \setminus E) = 0$ since $D \setminus E$ is a subset of $\overline{E}$ and $Pr(\overline{E}) = 1 - Pr(E) = 1 - 1 = 0$. This yields

$$Pr(D) \;=\; Pr(E \cap D) + \underbrace{Pr(D \setminus E)}_{=0} \;=\; Pr(E \cap D).$$

In particular, the event $E_1 \cap E_2$ of events E_1 and E_2 that hold almost surely, holds almost surely. As can be shown by induction, this carries over to any event that can be written as a finite intersection $\bigcap_{1 \leqslant i \leqslant n} E_i$ of events $E_1, \ldots, E_n$ that hold almost surely. By taking the limit of such (finite) intersections, we obtain that $\bigcap_{i \geqslant 1} E_i$ holds almost surely if $Pr(E_i) = 1$ for all $i \geqslant 0$.

To define an appropriate σ-algebra for a given Markov chain $\mathcal{M}$ (see below), we will use the fact that for each set *Outc* and each subset Π of 2^{Outc} there exists a *smallest σ-algebra* that contains Π. This is due to the observations that

- the powerset 2^{Outc} of *Outc* is a σ-algebra, and
- the intersection of σ-algebras is a σ-algebra.

Thus, the intersection $\mathfrak{E}_\Pi = \bigcap_{\mathfrak{E}} \mathfrak{E}$ where $\mathfrak{E}$ ranges over all σ-algebras on *Outc* that contain Π is a σ-algebra and is contained in any σ-algebra $\mathfrak{E}$ such that $\Pi \subseteq \mathfrak{E}$. $\mathfrak{E}_\Pi$ is called the σ-algebra *generated by* Π, and Π the *basis* for $\mathfrak{E}_\Pi$.

Probability Measure of a Markov Chain In order to be able to associate probabilities to events in Markov chains, the intuitive notion of probabilities in the Markov chain $\mathcal{M}$ is formalized by associating a probability space with $\mathcal{M}$. The infinite paths of $\mathcal{M}$ play the role of the outcomes. That is, $Outc^{\mathcal{M}} = Paths(\mathcal{M})$. Recall that $Paths(\mathcal{M})$ denotes the set of all infinite sequences $s_0\, s_1\, s_2 \ldots \in S^\omega$ such that $\mathbf{P}(s_i, s_{i+1}) > 0$ for all $i \geqslant 0$. (The requirement that $\iota_{\text{init}}(s_0) > 0$ could be added, but is irrelevant here.) The σ-algebra associated with $\mathcal{M}$ is generated by the *cylinder sets* spanned by the finite path fragments in $\mathcal{M}$.

Definition 10.9. Cylinder Set

The *cylinder set* of $\widehat{\pi} = s_0 \ldots s_n \in Paths_{fin}(\mathcal{M})$ is defined as

$$Cyl(\widehat{\pi}) = \{\, \pi \in Paths(\mathcal{M}) \mid \widehat{\pi} \in pref(\pi) \,\}.$$

■

The cylinder set spanned by the finite path $\widehat{\pi}$ thus consists of all infinite paths that start with $\widehat{\pi}$. The cylinder sets serve as basis events of the σ-algebra $\mathfrak{E}^{\mathcal{M}}$ associated with $\mathcal{M}$.

Definition 10.10. σ-Algebra of a Markov Chain

The σ-algebra $\mathfrak{E}^{\mathcal{M}}$ associated with MC $\mathcal{M}$ is the smallest σ-algebra that contains all cylinder sets $Cyl(\widehat{\pi})$ where $\widehat{\pi}$ ranges over all finite path fragments in $\mathcal{M}$. ■

From classical concepts of probability theory (see e.g. [21, 150]), it follows that there exists a unique probability measure $Pr^{\mathcal{M}}$ (or, briefly, Pr) on the σ-algebra $\mathfrak{E}^{\mathcal{M}}$ associated with $\mathcal{M}$ where the probabilities for the cylinder sets (i.e., the events) are given by

$$Pr^{\mathcal{M}}(Cyl(s_0 \dots s_n)) \;=\; \iota_{\text{init}}(s_0) \cdot \mathbf{P}(s_0 \dots s_n)$$

where

$$\mathbf{P}(s_0\, s_1 \dots s_n) \;=\; \prod_{0 \leqslant i < n} \mathbf{P}(s_i, s_{i+1}).$$

For path fragments of length zero, let $\mathbf{P}(s_0) = 1$.

For paths starting in a certain (possibly noninitial) state s, the same construction is applied to the MC $\mathcal{M}_s$ that results from $\mathcal{M}$ by declaring s as the unique initial state. Formally, for $\mathcal{M} = (S, \mathbf{P}, \iota_{\text{init}}, AP, L)$ and $s \in S$, the MC $\mathcal{M}_s$ is defined by $\mathcal{M}_s = (S, \mathbf{P}, \iota_s^1, AP, L)$ where

$$\iota_s^1(t) \;=\; \begin{cases} 1 & \text{if } s = t \\ 0 & \text{otherwise.} \end{cases}$$

We often write $Pr_s^{\mathcal{M}}$ for $Pr^{\mathcal{M}_s}$. If $\mathcal{M}$ is clear from the context, then $\mathcal{M}$ is omitted and we simply write Pr_s.

Example 10.11. Cylinder Sets

Consider the Markov chain for the craps game, see Figure 10.3. The state sequence $\widehat{\pi} = \mathit{start}\,6\,6\,6 \in \mathit{Paths}_{\mathit{fin}}(s_0)$. Its probability is given by

$$\iota_s(\mathit{start}) \cdot \mathbf{P}(\mathit{start}, 6) \cdot \mathbf{P}(6,6) \cdot \mathbf{P}(6,6) \;=\; \tfrac{5}{36} \cdot \left(\tfrac{25}{36}\right)^2.$$

The cylinder set of $\widehat{\pi}$ is given by

$$\{\, \mathit{start}\, 6^n\, \mathit{won}^\omega \mid n > 2 \,\} \cup \{\, \mathit{start}\, 6^n\, \mathit{lost}^\omega \mid n > 2 \,\} \cup \{\, \mathit{start}\, 6^\omega \,\}.$$

■

Notation 10.12. LTL-Style Notations for Events

In the remainder of this chapter, LTL-like notations are frequently used to describe events in Markov chains. For example, for set $B \subseteq S$ of states, $\Diamond B$ denotes the event to reach

(some state in) B eventually, and $\Box\Diamond B$ describes the event that B should be visited infinitely often. For singleton sets, the set brackets are typically omitted; e.g., $\Box\Diamond t$ means that $\{\, t \,\}$ is visited infinitely often. In line with the notations for LTL, we often write $\pi \models \varphi$ rather than $\pi \in \varphi$ if φ is an LTL-style notation for an event (i.e., an LTL formula with sets of states as atomic propositions) and π a path in $\mathcal{M}$. The probability for φ to hold in state s is denoted by $Pr^{\mathcal{M}}(s \models \varphi)$, i.e.,

$$Pr^{\mathcal{M}}(s \models \varphi) \;=\; Pr_s^{\mathcal{M}}\{\, \pi \in Paths(s) \mid \pi \models \varphi \,\}.$$

If $\mathcal{M}$ is clear from the context the superscript $\mathcal{M}$ is omitted. ■

10.1.1 Reachability Probabilities

One of the elementary questions for the quantitative analysis of systems modeled by Markov chains is to compute the probability of reaching a certain set B of states. The set B may represent a set of certain *bad* states which should be visited only with some small probability, or dually, a set of *good* states which should rather be visited frequently.

In the sequel, let $\mathcal{M} = (S, \mathbf{P}, \iota_{\text{init}}, AP, L)$ be a Markov chain and $B \subseteq S$ a set of states. The event of interest is denoted $\Diamond B$. In order to be able to reason about the probability of this event, we need to characterize the event $\Diamond B$ as a measurable set of paths. In fact, this is not difficult because the event $\Diamond B$ agrees with the union of all basic cylinders $Cyl(s_0 \ldots s_n)$ where $s_0 \ldots s_n$ is an initial path fragment in $\mathcal{M}$ such that $s_0, \ldots, s_{n-1} \notin B$ and $s_n \in B$. The set of all such paths is given by $Paths_{\mathit{fin}}(\mathcal{M}) \cap (S \setminus B)^* B$. As the set of finite path fragments is countable, this set is measurable, i.e., it is an element of the σ-algebra $\mathfrak{E}^{\mathcal{M}}$ on $\mathcal{M}$. Moreover, since these cylinder sets are pairwise disjoint, the probability of eventually reaching B is given by

$$\begin{aligned} Pr^{\mathcal{M}}(\Diamond B) &= \sum_{s_0 \ldots s_n \in Paths_{\mathit{fin}}(\mathcal{M}) \cap (S \setminus B)^* B} Pr^{\mathcal{M}}(Cyl(s_0 \ldots s_n)) \\ &= \sum_{s_0 \ldots s_n \in Paths_{\mathit{fin}}(\mathcal{M}) \cap (S \setminus B)^* B} \iota_{\text{init}}(s_0) \cdot \mathbf{P}(s_0 \ldots s_n). \end{aligned}$$

Example 10.13. Computing Reachability Probabilities by Infinite Series

Consider the Markov chain in Figure 10.1 on page 749, and suppose our interest is to compute the probability of reaching the state *delivered*. For this event, the finite initial path fragments $s_0 \ldots s_n$ with $s_i \neq \mathit{delivered}$ for $0 \leqslant i < n$ and $s_n = \mathit{delivered}$ are of

interest. These path fragments are of the form

$$\widehat{\pi}_n \;=\; \mathit{start}\,\mathit{try}\,(\mathit{lost}\,\mathit{try})^n\,\mathit{delivered}$$

where n is an arbitrary natural number. The probability of the cylinder set spanned by $\widehat{\pi_n}$ is $\left(\frac{1}{10}\right)^n \cdot \frac{9}{10}$. Hence,

$$Pr^{\mathcal{M}}(\Diamond \mathit{delivered}) \;=\; \sum_{n=0}^{\infty} \left(\frac{1}{10}\right)^n \cdot \frac{9}{10} \;=\; \frac{\frac{9}{10}}{1-\frac{1}{10}} \;=\; \frac{\frac{9}{10}}{\frac{9}{10}} \;=\; 1.$$

The event $\Diamond \mathit{delivered}$ thus holds almost surely. This is, in fact, rather intuitive since in the absence of a bound on the number of (re)transmissions it is to be expected that any message is delivered eventually. Let us now impose an upper bound, say three, for the number of transmission of a message. In order to determine the probability of the event to deliver the message within three trials, we sum the probabilities for the cylinder sets of the path fragments $\widehat{\pi}_0, \widehat{\pi}_1$, and $\widehat{\pi}_2$ which yields

$$\tfrac{9}{10} \;+\; \tfrac{1}{10} \cdot \tfrac{9}{10} \;+\; \tfrac{1}{10} \cdot \tfrac{1}{10} \cdot \tfrac{9}{10} \;=\; 0.999.$$

■

This example shows how the probability of reaching a certain set of states can be calculated by means of infinite sums. In general, this technique is rather involved and cumbersome. For instance, the computation of the event $\Diamond \mathit{won}$ in the craps game (see Figure 10.3, page 751) is much more involved. Let us now explain how probabilities of reaching a certain set B of states in a finite MC can be computed in a more efficient way, i.e., without considering infinite sums.

Let variable x_s denote the probability of reaching B from s, for arbitrary $s \in S$. The goal is to compute $x_s = Pr(s \models \Diamond B)$ for all states s. Clearly, if B is not reachable from s in the underlying directed graph of $\mathcal{M}$, then $x_s = 0$. Also the converse holds, i.e., if $x_s > 0$, then B is reachable from s. Moreover, $x_s = 1$ if $s \in B$. For the states $s \in S \setminus B$ for which B is reachable, it holds that

$$x_s \;=\; \sum_{t \in S \setminus B} \mathbf{P}(s,t) \cdot x_t \;+\; \sum_{u \in B} \mathbf{P}(s,u).$$

This equation states that either B is reached within one step, i.e., by a finite path fragment $s\,u$ with $u \in B$ (second summand), or first a state $t \in S \setminus B$ is reached from which B is reached—this corresponds to the path fragments $s\,t \ldots u$ of length $\geqslant 2$ where all states (except the last one) do not belong to B (first summand). Let $\tilde{S} = \mathit{Pre}^*(B) \setminus B$ denote

the set of states $s \in S \setminus B$ such that there is a path fragment $s_0\, s_1 \ldots s_n$ $(n > 0)$ with $s_0 = s$ and $s_n \in B$. For the vector $\mathbf{x} = (x_s)_{s \in \tilde{S}}$, we have

$$\mathbf{x} \;=\; \mathbf{A}\mathbf{x} \;+\; \mathbf{b},$$

where the matrix $\mathbf{A}$ contains the transition probabilities for the states in $\tilde{S}$, i.e., $\mathbf{A} = (\,\mathbf{P}(s,t)\,)_{s,t \in \tilde{S}}$, and the vector $\mathbf{b} = (b_s)_{s \in \tilde{S}}$ contains the probabilities of reaching B from $\tilde{S}$ within one step, i.e., $b_s = \mathbf{P}(s, B) = \sum_{u \in B} \mathbf{P}(s,u)$. The above equation system can be rewritten into a (heterogeneous) *linear equation system*

$$(\mathbf{I} - \mathbf{A}){\cdot}\mathbf{x} \;=\; \mathbf{b}$$

where $\mathbf{I}$ is the identity matrix of cardinality $|\tilde{S}| \times |\tilde{S}|$.

Example 10.14. Simple Communication Protocol

Consider the simple communication protocol in Figure 10.1 on page 749 and the event $\Diamond B$ for $B = \{\, \mathit{delivered} \,\}$. Then $x_s > 0$ for all states s, since *delivered* is reachable from all states. In this case, $\tilde{S} = \{\, \mathit{start}, \mathit{try}, \mathit{lost} \,\}$ and we obtain the equations

$$\begin{array}{lcl} x_{\mathit{start}} & = & x_{\mathit{try}} \\ x_{\mathit{try}} & = & \frac{1}{10} \cdot x_{\mathit{lost}} \;+\; \frac{9}{10} \\ x_{\mathit{lost}} & = & x_{\mathit{try}}. \end{array}$$

These equations can be rewritten as

$$\begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & -\frac{1}{10} \\ 0 & -1 & 1 \end{pmatrix} \cdot \mathbf{x} \;=\; \begin{pmatrix} 0 \\ \frac{9}{10} \\ 0 \end{pmatrix}$$

which yields the (unique) solution $x_{\mathit{start}} = x_{\mathit{try}} = x_{\mathit{lost}} = 1$. Thus, the event of eventually reaching the state *delivered* is almost sure for any state. ■

The above technique yields the following two-phase algorithm to compute reachability probabilities in finite Markov chains: first, perform a graph analysis to compute the set $\tilde{S}$ of all states that can reach B (e.g., by a backward DFS- or BFS-based search from B), then generate the matrix $\mathbf{A}$ and the vector $\mathbf{b}$, and solve the linear equation system $(\mathbf{I} - \mathbf{A}){\cdot}\mathbf{x} = \mathbf{b}$. However, $(\mathbf{I} - \mathbf{A}){\cdot}\mathbf{x} = \mathbf{b}$ might have more than one solution. This is the case if $\mathbf{I} - \mathbf{A}$ is singular, i.e., does not have an inverse. This problem is addressed below by characterizing the desired probability vector as the *least solution* in $[0,1]^{\tilde{S}}$. This characterization enables computing the probability vector by an iterative approximation method. (To apply direct methods, like Gaussian elimination, several rows and columns of the matrix $\mathbf{I} - \mathbf{A}$ have to be removed such that the remaining linear equation system

has a unique solution.) In fact, we present a characterization for a slightly more general problem, viz. *constrained reachability* (i.e., until properties). The obtained characterization also partly applies to infinite Markov chains.

Let $\mathcal{M} = (S, \mathbf{P}, \iota_{\text{init}}, AP, L)$ be a (possibly infinite) MC and B, $C \subseteq S$. Consider the event of reaching B via a finite path fragment which ends in a state $s \in B$, and visits only states in C prior to reaching s. Using LTL-like notations, this event is denoted by $C \,\mathsf{U}\, B$. The event $\Diamond B$ considered above agrees with $S \,\mathsf{U}\, B$. For $n \geqslant 0$, the event $C \,\mathsf{U}^{\leqslant n} B$ has the same meaning as $C \,\mathsf{U}\, B$, except that it is required to reach B (via states in C) within n steps. Formally, $C \,\mathsf{U}^{\leqslant n} B$ is the union of the basic cylinders spanned by path fragments $s_0\, s_1 \ldots s_k$ such that $k \leqslant n$ and $s_i \in C$ for all $0 \leqslant i < k$ and $s_k \in B$.

Let $S_{=0}$, $S_{=1}$, $S_?$ a partition of S such that

- $B \subseteq S_{=1} \subseteq \{\, s \in S \mid Pr(s \models C \,\mathsf{U}\, B) = 1 \,\}$,
- $S \setminus (C \cup B) \subseteq S_{=0} \subseteq \{\, s \in S \mid Pr(s \models C \,\mathsf{U}\, B) = 0 \,\}$, and
- $S_? = S \setminus (S_{=1} \cup S_{=0})$.

For all states in the set $S_{=1}$, the event $C \,\mathsf{U}\, B$ almost surely holds. All states in $S_?$ belong to $C \setminus B$. Let the matrix $\mathbf{A}$ be a quadratic matrix with rows and columns for the states in $S_?$. This matrix is obtained from the transition probability matrix $\mathbf{P}$ by omitting the rows and columns for the states $s \in S_{=0} \cup S_{=1}$ from $\mathbf{P}$. That is

$$\mathbf{A} \;=\; (\,\mathbf{P}(s,t)\,)_{s,t \in S_?}$$

Similarly, let vector $\mathbf{b}$ be defined as $(\,b_s\,)_{s \in S_?}$ where $b_s = \mathbf{P}(s, S_{=1})$.

We now provide a least fixed point characterization for the probability vector $(\,Pr(s \models C \,\mathsf{U}\, B)\,)_{s \in S_?}$. Note that the set $[0,1]^{S_?}$ consists of all vectors $\mathbf{y} = (y_s)_{s \in S_?}$ with $y_s \in [0,1]$ for all $s \in S_?$. To obtain the least fixed point characterization, the set $[0,1]^{S_?}$ is equipped with the partial order $\leqslant$ given by $\mathbf{y} \leqslant \mathbf{y}'$ if and only if $y_s \leqslant y'_s$ for all $s \in S_?$, where $\mathbf{y} = (y_s)_{s \in S_?}$ and $\mathbf{y}' = (y'_s)_{s \in S_?}$.

Theorem 10.15. Least Fixed Point Characterization

The vector $\mathbf{x} = (\,Pr(s \models C \,\mathsf{U}\, B)\,)_{s \in S_?}$ *is the* least fixed point *of the operator* $\Upsilon : [0,1]^{S_?} \to [0,1]^{S_?}$ *which is given by*

$$\Upsilon(\mathbf{y}) \;=\; \mathbf{A} \cdot \mathbf{y} + \mathbf{b}.$$

Furthermore, if $\mathbf{x}^{(0)} = \mathbf{0}$ *is the vector consisting of zeros only, and* $\mathbf{x}^{(n+1)} = \Upsilon(\mathbf{x}^{(n)})$ *for* $n \geqslant 0$, *then*

- $\mathbf{x}^{(n)} = (x_s^{(n)})_{s \in S_?}$ *where* $x_s^{(n)} = Pr(s \models C \,\mathsf{U}^{\leqslant n} S_{=1})$ *for each state* $s \in S_?$,
- $\mathbf{x}^{(0)} \leqslant \mathbf{x}^{(1)} \leqslant \mathbf{x}^{(2)} \leqslant \ldots \leqslant \mathbf{x}$, *and*
- $\mathbf{x} = \lim_{n \to \infty} \mathbf{x}^{(n)}$.

Before proving Theorem 10.15, let us first comment on the well-definedness of Υ as a function from $[0,1]^{S_?}$ to $[0,1]^{S_?}$. For $\mathbf{y} = (y_s)_{s \in S_?}$, the vector $\Upsilon(\mathbf{y}) = (y'_s)_{s \in S_?}$ has the entries

$$y'_s \;=\; \sum_{t \in S_?} \mathbf{P}(s,t) \cdot y_t \;+\; \mathbf{P}(s, S_{=1}).$$

Since $0 \leqslant y_t \leqslant 1$ for all $t \in S_?$, $\mathbf{P}(s,s') \geqslant 0$ and $\sum_{s' \in S} \mathbf{P}(s,s') = 1$, this implies $0 \leqslant y'_s \leqslant 1$. Therefore, $\Upsilon(\mathbf{y}) \in [0,1]^{S_?}$.

Proof: Let $x_s = Pr(s \models C \,\mathsf{U}\, B)$ for each state $s \in S$. Then, it follows from the definitions of $S_{=0}$ and $S_{=1}$ that $\mathbf{x} = (x_s)_{s \in S_?}$, $x_s = 0$ for $s \in S_{=0}$, and $x_s = 1$ for $s \in S_{=1}$.

1. Let us first show that $\mathbf{x}$ agrees with the limit of the vectors $\mathbf{x}^{(n)} = (x_s^{(n)})_{s \in S_?}$. By induction on n, one can show that $x_s^{(n)} = Pr(\, s \models C \,\mathsf{U}^{\leqslant n} S_{=1}\,)$. Since $C \,\mathsf{U}\, S_{=1}$ is the countable union of the events $C \,\mathsf{U}^{\leqslant n} S_{=1}$, we obtain

$$\lim_{n \to \infty} x_s^{(n)} \;=\; Pr(s \models C \,\mathsf{U}\, S_{=1}) \;=\; x_s.$$

2. The fixed point property $\mathbf{x} = \Upsilon(\mathbf{x})$ is a consequence of the fact that $x_s = Pr(s \models C \,\mathsf{U}\, B) = 0$ if $s \in S_{=0}$ and $x_s = Pr(s \models C \,\mathsf{U}\, B) = 1$ if $s \in S_{=1}$. For $s \in S_?$ we derive that

$$\begin{aligned} x_s \;&=\; \sum_{t \in S} \mathbf{P}(s,t) \cdot x_t \\ &=\; \sum_{t \in S_{=0}} \mathbf{P}(s,t) \cdot \underbrace{x_t}_{=0} \;+\; \sum_{t \in S_?} \mathbf{P}(s,t) \cdot x_t \;+\; \sum_{t \in S_{=1}} \mathbf{P}(s,t) \cdot \underbrace{x_t}_{=1} \\ &=\; \sum_{t \in S_?} \mathbf{P}(s,t) \cdot x_t \;+\; \underbrace{\sum_{t \in S_{=1}} \mathbf{P}(s,t)}_{=b_s} \end{aligned}$$

   is the component for state s in the vector $\Upsilon(\mathbf{x})$.

3. It remains to show that $\mathbf{x} \leqslant \mathbf{y}$ for each fixed point $\mathbf{y}$ of Υ. This follows from the fact that $\mathbf{x}^{(n)} \leqslant \mathbf{y}$ for all n, which can be shown by induction on n. But then $\mathbf{x} = \lim_{n \to \infty} \mathbf{x}^{(n)} \leqslant \mathbf{y}$.

This completes the proof. ■

Remark 10.16. Expansion Law

The statement of the above theorem with $S_{=1} = B$ and $S_{=0} = S \setminus (C \cup B)$ can be considered as the probabilistic counterpart to the characterization of the CTL formula $\exists(C \,\mathsf{U}\, B)$ as the least solution of the expansion law:

$$\exists(C \,\mathsf{U}\, B) \;\equiv\; B \;\vee\; (C \;\wedge\; \exists \bigcirc \exists(C \,\mathsf{U}\, B)).$$

To make this clear, let us rewrite this expansion law in the following way. The set $X = \mathit{Sat}(\exists(C \,\mathsf{U}\, B))$ is the least set such that

$$B \;\cup\; \{\, s \in C \setminus B \;\mid\; \mathit{Post}(s) \cap X \neq \varnothing \,\} \;\subseteq\; X.$$

In Theorem 10.15, this set-based least fixed point characterization is phrased for the quantitative setting where instead of truth values for "$s \in X$" we deal with values x_s in $[0,1]$. If $s \in B$, then $x_s = 1$. This corresponds to the statement that $s \in B$ implies $s \in X$. If $s \in C \setminus B$, then $x_s = \sum_{t \in C \setminus B} \mathbf{P}(s,t) \cdot x_t + \sum_{t \in B} \mathbf{P}(s,t)$. This corresponds to the statement that $s \in C \setminus B$ and $\mathit{Post}(s) \cap X \neq \varnothing$ implies $s \in X$. Finally, if $s \in S \setminus (C \cup B)$, then $x_s = 0$. This corresponds to the statement that $s \notin X$ if $s \notin C \cup B$. ■

The last part of Theorem 10.15 in fact provides a recipe for computing an approximation of the desired probability vector $\mathbf{x}$. It suggests to use the following iterative approach:

$$\mathbf{x}^{(0)} \;=\; \mathbf{0} \quad \text{and} \quad \mathbf{x}^{(n+1)} \;=\; \mathbf{A}\mathbf{x}^{(n)} + \mathbf{b} \quad \text{for } n \geqslant 0.$$

This method, often called *power method*, provides a simple iterative algorithm which computes the vectors $\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots$ by matrix-vector multiplication and vector addition. It aborts as soon as $\max_{s \in S_?} |x_s^{(n+1)} - x_s^{(n)}| < \varepsilon$ for some small user-defined tolerance ε. Although the convergence of the power method is guaranteed, it is often less efficient than other iterative methods for solving linear equation systems of large dimensions, such as the Jacobi or Gauss-Seidel method. Explanations of such numerical methods for linear equation systems go beyond the scope of this monograph and can be found in textbooks such as [196, 248, 344].

Remark 10.17. Choosing $S_{=0}$ and $S_{=1}$

The constraints on $S_{=0}$ and $S_{=1}$

$$B \;\subseteq\; S_{=1} \;\subseteq\; \{\, s \in S \mid \mathit{Pr}(s \models C \,\mathsf{U}\, B) = 1 \,\}$$

and

$$S \setminus (C \cup B) \;\subseteq\; S_{=0} \;\subseteq\; \{\, s \in S \mid \mathit{Pr}(s \models C \,\mathsf{U}\, B) = 0 \,\},$$

do not uniquely characterize $S_{=0}$ and $S_{=1}$. For instance, $S_{=0} = S \setminus (C \cup B)$ and $S_{=1} = B$ would suffice. For efficiency reasons, it is advantageous to deal with the largest possible sets. The larger the sets $S_{=0}$ and $S_{=1}$, the smaller is their complement $S_?$, and the smaller the linear equation system that needs to be solved—as there is a variable for each $s \in S_?$. A reasonable choice is

$$\begin{aligned} S_{=0} &= \{\, s \in S \mid Pr(s \models C \,\mathsf{U}\, B) = 0 \,\} \quad \text{and} \\ S_{=1} &= \{\, s \in S \mid Pr(s \models C \,\mathsf{U}\, B) = 1 \,\}. \end{aligned}$$

These sets can be computed by simple graph algorithms that have a linear-time complexity in the size of the MC $\mathcal{M}$, i.e., the number of states plus the number of nonzero entries in the transition probability matrix. The computation of $S_{=0}$ is straightforward since

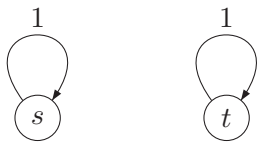
$$Pr(s \models C \,\mathsf{U}\, B) = 0 \;\text{ if and only if }\; s \not\models \exists(C \,\mathsf{U}\, B)$$

where $\exists(C \,\mathsf{U}\, B)$ is viewed as a CTL formula. Thus, $S_{=0}$ can be calculated by means of a backward search starting from B in time $\mathcal{O}(size(\mathcal{M}))$. The computation of $S_{=1}$ can also be realized by means of simple graph traversal techniques, as will be explained in the next section; see page 775ff. ■

The results we have established so far permit computing approximations of the probability $Pr(s \models C \,\mathsf{U}\, B)$ by iterative methods. In order to apply direct methods for a linear equation system, like Gaussian elimination, the uniqueness of the linear equation system to be solved is required. Without any additional assumptions on the choice of $S_{=0}$, the uniqueness can, however, not be guaranteed. This even applies to finite Markov chains as shown by the following example.

Remark 10.18. Several Fixed Points

Under the assumption that $S_{=0}$ is a proper subset of $\{\, s \in S \mid Pr(s \models C \,\mathsf{U}\, B) = 0 \,\}$, the operator Υ may have more than one fixed point. Consider, e.g., the Markov chain:



Consider the event $\Diamond s$. A possible choice is $S_{=0} = \varnothing$ and $S_{=1} = \{\, s \,\}$, i.e., $S_? = \{\, t \,\}$. Matrix $\mathbf{A}$ is the identity matrix of cardinality one and $\mathbf{b}$ is the vector with the single entry 0 for state t. Thus, the equation system $\mathbf{x} = \mathbf{A}\mathbf{x} + \mathbf{b}$ represents the trivial equation $x_t = x_t$ and the operator $\Upsilon : [0,1] \to [0,1]$ is given by $\Upsilon(y_t) = y_t$. Clearly, Υ has infinitely many fixed points, namely all values $y_t \in [0,1]$. However, the probabilities for the event $\Diamond s$ are given by the least fixed point $x_t = 0$ of Υ. ■

Unique fixed points are guaranteed if $\mathcal{M}$ is finite and if $S_{=0}$ covers all states s with $s \not\models \exists(C \,\mathsf{U}\, B)$. As mentioned before, these are exactly the states s for which $Pr(s \models C \,\mathsf{U}\, B) = 0$. This fact is captured by the following theorem:

Theorem 10.19. Unique Solution

Let $\mathcal{M}$ be a finite Markov chain with state space S, and B, $C \subseteq S$,

$$S_{=0} \;=\; Sat(\neg\exists(C \,\mathsf{U}\, B)) \text{ and } B \subseteq S_{=1} \subseteq \{\, s \in S \mid Pr(s \models C \,\mathsf{U}\, B) = 1 \,\}$$

and $S_? \;=\; S \setminus (S_{=0} \cup S_{=1})$. Then, the vector $(\, Pr(s \models C \,\mathsf{U}\, B) \,)_{s \in S_?}$ is the unique *solution of the equation system $\mathbf{x} = \mathbf{A}\mathbf{x} + \mathbf{b}$ where $\mathbf{A} = (\, \mathbf{P}(s,t) \,)_{s,t \in S_?}$, and $\mathbf{b} = (\, \mathbf{P}(s, S_{=1}) \,)_{s \in S_?}$.*

Proof: Suppose there are two solutions of the equation system, say $\mathbf{x} = \mathbf{A}\mathbf{x} + \mathbf{b}$ and $\mathbf{y} = \mathbf{A}\mathbf{y} + \mathbf{b}$. Thus $\mathbf{x} - \mathbf{y} = \mathbf{A}(\mathbf{x} - \mathbf{y})$. Assume $\mathbf{A}\mathbf{x} = \mathbf{x}$ implies $\mathbf{x} = \mathbf{0}$, where $\mathbf{0}$ is a vector just consisting of zeros. Then $\mathbf{x} - \mathbf{y} = \mathbf{A}(\mathbf{x} - \mathbf{y})$ yields $\mathbf{x} - \mathbf{y} = \mathbf{0}$, and therefore $\mathbf{x} = \mathbf{y}$.

We now prove that indeed $\mathbf{A}\mathbf{x} = \mathbf{x}$ implies $\mathbf{x} = \mathbf{0}$. This is proven by contraposition. Assume $\mathbf{x} = (x_s)_{s \in S_?}$ is a vector such that $\mathbf{A}\mathbf{x} = \mathbf{x}$ and $\mathbf{x} \neq \mathbf{0}$. Since $\mathcal{M}$ is finite, the maximum of the values $|x_s|$ is well-defined. Let x be this maximum, and T the set of states for which $x_s = x$, i.e.:

$$x \;=\; \max\{\, |x_s| \mid s \in S_? \,\} \quad \text{and} \quad T \;=\; \{\, s \in S_? \mid |x_s| = x \,\}.$$

As $\mathbf{x} \neq \mathbf{0}$, it follows that $x > 0$. Furthermore, $T \neq \varnothing$. Since the values $\mathbf{P}(s,t)$ are non-negative and $\sum_{t \in S_?} \mathbf{P}(s,t) \leqslant 1$, we obtain for each $s \in T$:

$$x \;=\; |x_s| \;\leqslant\; \sum_{t \in S_?} \mathbf{P}(s,t) \cdot \underbrace{|x_t|}_{\leqslant x} \;\leqslant\; x \cdot \sum_{t \in S_?} \mathbf{P}(s,t) \;\leqslant\; x.$$

This yields

$$x \;=\; |x_s| \;=\; \sum_{t \in S_?} \mathbf{P}(s,t) \cdot |x_t| \;=\; x \cdot \sum_{t \in S_?} \mathbf{P}(s,t).$$

As $x > 0$, this implies $\sum\limits_{t \in S_?} \mathbf{P}(s,t) \;=\; 1$ and $|x_t| = x$ for all states in $Post(s) \cap S_?$. But then for any $s \in T$:

$$Post(s) \;=\; \{\, t \in S \mid \mathbf{P}(s,t) > 0 \,\} \;\subseteq\; T.$$

We conclude that $Post^*(s) \subseteq T$ for all $s \in T$. In particular, $Post^*(s) \subseteq S_?$ and $Post^*(s) \cap B = \varnothing$ (recall that $B \subseteq S_{=1}$ and therefore $B \cap S_? = \varnothing$). Thus, none of the states $s \in T$ can reach B. Therefore,

$$T \;\subseteq\; \{\, s \in S \mid s \not\models \exists(C \,\mathsf{U}\, B) \,\} \;\subseteq\; S_{=0}.$$

But, as $T \subseteq S_?$ and $S_{=0} \cap S_? = \varnothing$, T must be empty. Contradiction. ■

Remark 10.20. Nonsingularity of Matrix $\mathbf{I} - \mathbf{A}$

A short remark for readers familiar with matrix norms. By roughly the same arguments as in the proof of Theorem 10.15 (page 762) it follows that the matrix $\mathbf{A}$ does not have an Eigenvalue λ with $|\lambda| \geqslant 1$. Then, the *spectral norm* of $\mathbf{A}$, defined as

$$\max\{\, |\lambda| \mid \lambda \text{ is a (complex) Eigenvalue of } \mathbf{A} \,\},$$

is strictly less than 1. Thus, the infinite series:

$$\mathbf{I} + \mathbf{A} + \mathbf{A}^2 + \mathbf{A}^3 + \ldots \;=\; \sum_{n \geqslant 0} \mathbf{A}^n$$

converges and its limit is the inverse of $\mathbf{I} - \mathbf{A}$ where $\mathbf{I}$ is the identity matrix (of the same cardinality as $\mathbf{A}$). Note that

$$\begin{aligned} & (\mathbf{I} - \mathbf{A}) \cdot (\mathbf{I} + \mathbf{A} + \mathbf{A}^2 + \mathbf{A}^3 + \ldots) \\ = \;& (\mathbf{I} + \mathbf{A} + \mathbf{A}^2 + \mathbf{A}^3 + \ldots) \;-\; (\mathbf{A} + \mathbf{A}^2 + \mathbf{A}^3 + \mathbf{A}^4 + \ldots) \\ = \;& \mathbf{I}. \end{aligned}$$

This yields another argument why $\mathbf{x} = (\mathbf{I} - \mathbf{A})^{-1}\mathbf{b}$ is a unique solution of $\mathbf{A}\mathbf{x} + \mathbf{b} = \mathbf{x}$. ■

As a result, Theorem 10.15 yields an iterative method for computing *step-bounded constrained reachability probabilities*. The probabilities of the events $C \,\mathsf{U}^{\leqslant n}\, B$ can be computed by the following iterative scheme: $\mathbf{x}^{(0)} = \mathbf{0}$ and $\mathbf{x}^{(i+1)} = \mathbf{A}\mathbf{x}^{(i)} + \mathbf{b}$ for $0 \leqslant i < n$. The matrix $\mathbf{A}$ and the vector $\mathbf{b}$ are defined based on $S_{=0} = S \setminus (C \cup B)$, $S_{=1} = B$ and $S_? = C \setminus B$. That is:

$$\mathbf{A} \;=\; (\,\mathbf{P}(s,t)\,)_{s,t \in C \setminus B} \quad \text{and} \quad \mathbf{b} \;=\; (\,\mathbf{P}(s,B)\,)_{s \in C \setminus B}.$$

For $s \in C \setminus B$, entry $\mathbf{x}^{(n)}(s)$ equals $Pr(s \models C \,\mathsf{U}^{\leqslant n}\, B)$.

Example 10.21. Constrained Reachability in the Craps Game

Consider the Markov chain modeling the craps game, see Figure 10.3 (page 751). The event of interest is $C \,\mathsf{U}^{\leqslant n}\, B$ where $B = \{\, won \,\}$, $C = \{\, start, 4, 5, 6 \,\}$. The bounded constrained reachability probability $Pr(start \models C \,\mathsf{U}^{\leqslant n}\, B)$ is the likelihood of winning the game by either throwing 7 or 11 (the player wins directly), or one or more times only 4, 5, or 6. It follows that

$$S_{=0} = \{\, 8, 9, 10, lost \,\} \quad S_{=1} = \{\, won \,\} \quad S_? = \{\, start, 4, 5, 6 \,\}.$$

Using the state order $start < 4 < 5 < 6$, the matrix $\mathbf{A}$ and the vector $\mathbf{b}$ are given by

$$\mathbf{A} = \frac{1}{36}\begin{pmatrix} 0 & 3 & 4 & 5 \\ 0 & 27 & 0 & 0 \\ 0 & 0 & 26 & 0 \\ 0 & 0 & 0 & 25 \end{pmatrix} \qquad \mathbf{b} = \frac{1}{36}\begin{pmatrix} 8 \\ 3 \\ 4 \\ 5 \end{pmatrix}.$$

The least fixed point characterization suggests the following iterative scheme:

$$\mathbf{x}^{(0)} = \mathbf{0} \quad \text{and} \quad \mathbf{x}^{(i+1)} = \mathbf{A}\mathbf{x}^{(i)} + \mathbf{b} \text{ for } 0 \leqslant i < n.$$

where $\mathbf{x}^i$ records for any states in $S_?$ the probability of the event $C \,\mathsf{U}^{\leqslant n}\, B$. Applying this iterative scheme to the example yields $\mathbf{x}^{(1)} = \mathbf{b}$ and

$$\mathbf{x}^{(2)} = \frac{1}{36}\begin{pmatrix} 0 & 3 & 4 & 5 \\ 0 & 27 & 0 & 0 \\ 0 & 0 & 26 & 0 \\ 0 & 0 & 0 & 25 \end{pmatrix} \cdot \frac{1}{36}\begin{pmatrix} 8 \\ 3 \\ 4 \\ 5 \end{pmatrix} + \frac{1}{36}\begin{pmatrix} 8 \\ 3 \\ 4 \\ 5 \end{pmatrix} = \left(\frac{1}{36}\right)^2 \begin{pmatrix} 338 \\ 189 \\ 248 \\ 305 \end{pmatrix}.$$

For instance, we have $Pr(start \models C \,\mathsf{U}^{\leqslant 2}\, B) = \frac{338}{36^2}$. In a similar way, one obtains $\mathbf{x}^{(3)}$, $\mathbf{x}^{(4)}$, and so forth. ■

Remark 10.22. Transient State Probabilities

The nth power of $\mathbf{A}$, i.e., the matrix $\mathbf{A}^n$, contains the state probabilities after exactly n steps (i.e., transitions) inside $S_?$. More precisely, matrix entry $\mathbf{A}^n(s,t)$ equals the sum of the probabilities $\mathbf{P}(s_0\, s_1 \ldots s_n)$ of all path fragments $s_0\, s_1 \ldots s_n$ with $s_0 = s$, $s_n = t$ and $s_i \in S_?$ for $0 \leqslant i \leqslant n$. That is:

$$\mathbf{A}^n(s,t) = Pr(s \models S_? \,\mathsf{U}^{=n}\, t).$$

If $B = \varnothing$ and $C = S$, then $S_{=1} = S_{=0} = \varnothing$ and $S_? = S$, yielding $\mathbf{A} = \mathbf{P}$. The entry $\mathbf{P}^n(s,t)$ (of the nth power of $\mathbf{P}$) thus equals the probability of being in state t after n steps given that the computation starts in state s, i.e., $\mathbf{P}^n(s,t) = Pr(s \models S \,\mathsf{U}^{=n}\, t)$. The probability of $\mathcal{M}$ being in state t after exactly n transitions

$$\Theta_n^{\mathcal{M}}(t) = \sum_{s \in S} \mathbf{P}^n(s,t) \cdot \iota_{\text{init}}(s),$$

is called the *transient state probability* for state t. The function $\Theta_n^{\mathcal{M}}$ is the *transient state distribution*. When considering $\Theta_n^{\mathcal{M}}$ as the vector $(\,\Theta_n^{\mathcal{M}}(t)\,)_{t \in S}$, the above equation can be rewritten as

$$\Theta_n^{\mathcal{M}} = \underbrace{\mathbf{P} \cdot \mathbf{P} \cdot \ldots \cdot \mathbf{P}}_{n \text{ times}} \cdot \iota_{\text{init}} = \mathbf{P}^n \cdot \iota_{\text{init}},$$

where the initial distribution is viewed as a column vector. Due to the numerical instability of computing the nth power of a matrix, using, e.g., iterative squaring, it is recommended to compute $\Theta_n^{\mathcal{M}}$ by successive matrix-vector multiplication:

$$\Theta_0^{\mathcal{M}} \;=\; \iota_{\text{init}} \quad \text{and} \quad \Theta_{n+1}^{\mathcal{M}} \;=\; \mathbf{P} \cdot \Theta_n^{\mathcal{M}} \quad \text{for } n \geqslant 0.$$

Transient state probabilities are thus special instances of constrained reachability probabilities. In fact, constrained reachability probabilities in MC $\mathcal{M}$ coincide with transient state probabilities in a slightly modified Markov chain. Let us first illustrate this by means of simple, i.e., unconstrained, step-bounded reachability probabilities in $\mathcal{M}$, say the event $\Diamond^{\leqslant n} B$. The MC $\mathcal{M}$ is modified such that all states $s \in B$ are made *absorbing*. That is, all outgoing transitions of $s \in B$ are replaced by a self-loop. This yields the MC $\mathcal{M}_B$. The intuition of this transformation is that once a path reaches a state in B, its subsequent states are of no importance to $Pr^{\mathcal{M}}(s \models \Diamond^{\leqslant n} B)$. Formally, $\mathcal{M}_B = (S, \mathbf{P}_B, L)$ with S and L as for $\mathcal{M}$, and $\mathbf{P}_B$ is defined by $\mathbf{P}_B(s,t) = \mathbf{P}(s,t)$ if $s \notin B$, $\mathbf{P}_B(s,s) = 1$, and $\mathbf{P}_B(s,t) = 0$ for $s \in B$ and $s \neq t$. It then follows for any $s \in S$:

$$Pr^{\mathcal{M}}(s \models \Diamond^{\leqslant n} B) \;=\; Pr^{\mathcal{M}_B}(s \models \Diamond^{=n} B).$$

The probability of reaching a B-state within n steps in $\mathcal{M}$ is now given by:

$$Pr^{\mathcal{M}}(\Diamond^{\leqslant n} B) \;=\; \sum_{t \in B} \Theta_n^{\mathcal{M}_B}(t).$$

The reachability probability $Pr^{\mathcal{M}}(\Diamond^{\leqslant n} B)$ in the MC $\mathcal{M}$ thus agrees with the cumulative transient state probability for the B-states in the MC $\mathcal{M}_B$.

Using similar arguments, one can show that the computation of step-bounded constrained reachability probabilities $Pr^{\mathcal{M}}(C \,\mathsf{U}^{\leqslant n} B)$ can be reduced to the problem of computing transient state probabilities in a slightly modified Markov chain. As for the simple reachability probabilities, all states in B are made absorbing. In addition, all states in $S \setminus (C \cup B)$ are made absorbing. This is justified by the fact that paths that visit some state in $S \setminus (C \cup B)$ contribute probability zero to $Pr^{\mathcal{M}}(s \models C \,\mathsf{U}^{\leqslant n} B)$. We thus consider the MC $\mathcal{M}' = \mathcal{M}_{B \cup (S \setminus (C \cup B))}$ that results from $\mathcal{M}$ by making all states in $B \cup (S \setminus (C \cup B))$ absorbing. It then follows that for any state $s \in S$:

$$Pr^{\mathcal{M}}(s \models C \,\mathsf{U}^{\leqslant n}\, B) \;=\; Pr^{\mathcal{M}'}(s \models \Diamond^{=n} B).$$

The probability of reaching a B-state within n steps in $\mathcal{M}$ via C-states only is now given by

$$Pr^{\mathcal{M}}(C \,\mathsf{U}^{\leqslant n}\, B) \;=\; \sum_{t \in B} \Theta_n^{\mathcal{M}'}(t).$$

■

Example 10.23. Reachability by Transient Probabilities

Consider again the Markov chain modeling the craps game (see Figure 10.3, page 751). Let $C \,\mathsf{U}^{\leqslant n}\, B$ be the event of interest where $B = \{\, won \,\}$, $C = \{\, start, 4, 5, 6 \,\}$. According to the described procedure just above, all states in B and all states that are neither in B nor in C are made absorbing. This yields the MC depicted in Figure 10.5. The constrained

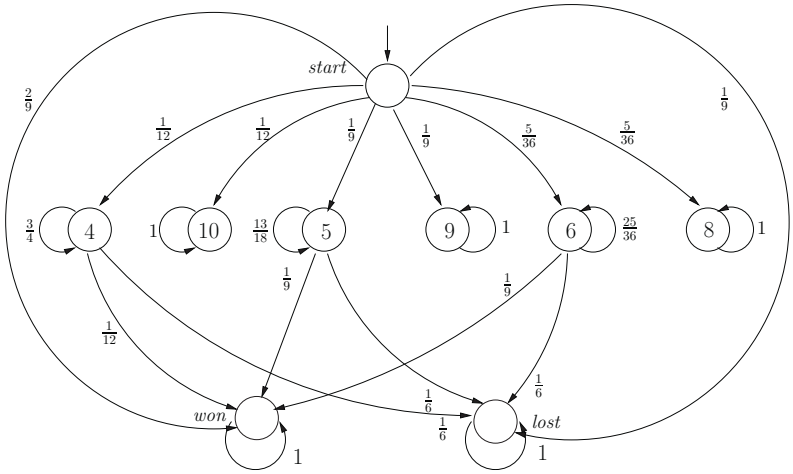


Figure 10.5: Markov chain for the craps game for $C \,\mathsf{U}^{\leqslant n} B$.

reachability probabilities for $C \,\mathsf{U}^{\leqslant n}\, B$ are obtained as follows. As before, let $\iota_{\text{init}}(start) = 1$ and $\iota_{\text{init}}(s) = 0$ for any other state. For n=0, we have

$$Pr(start \models C \,\mathsf{U}^{\leqslant 0} B) = \Theta_0(won) = \iota_{init}(won) = 0.$$

For n=1, one obtains

$$Pr(start \models C \,\mathsf{U}^{\leqslant 1} B) = \iota_{init}(start) \cdot \mathbf{P}(start, won) \;=\; \frac{2}{9}.$$

For n=2, one obtains $Pr(start \models C \,\mathsf{U}^{\leqslant 2} B) = \iota_{init}(start) \cdot \mathbf{P}^2(start, won)$ which equals $\frac{338}{36^2}$. For other n, the probabilities are obtained in a similar way. ■

10.1.2 Qualitative Properties

In the previous section, we have shown different ways to compute reachability probabilities in Markov chains. This section is focused on *qualitative properties* of Markov chains. Such properties typically require certain events to happen with probability 1, or more generally, to compute all states where a certain event holds almost surely. Dually, the problem

to check whether an event occurs with zero probability is a qualitative property. This section will show that qualitative properties for events such as reachability, constrained reachability, repeated reachability—can a certain set of states be visited repeatedly?—and persistence—can only a certain set of states be visited from some moment on?—can all be verified using graph analysis, i.e., by just considering the underlying digraph of the Markov chain and ignoring the transition probabilities. This is due to a fundamental result for *finite* Markov chains stating that almost surely the computations will enter a bottom strongly connected component (an SCC that once entered cannot be left anymore) and visit each of its states infinitely often. This result will be proven in detail. Finally, it will be shown that this result does not apply to infinite Markov chains.

The first thing to be settled is that the events to be considered, such as repeated reachability and persistence, are measurable.

Remark 10.24. Measurability of Repeated Reachability and Persistence
Let $B \subseteq S$ be a set of states in a Markov chain $\mathcal{M}$. The set of all paths that visit B infinitely often is measurable. This holds in arbitrary (possibly infinite) Markov chains and follows from the fact that the event $\Box\Diamond B$ can be written as the countable intersection of countable unions of cylinder sets:

$$\Box\Diamond B \;=\; \bigcap_{n\geqslant 0} \bigcup_{m\geqslant n} \mathit{Cyl}(\text{“}(m{+}1)\text{st state is in } B\text{”})$$

where $\mathit{Cyl}(\text{“}(m{+}1)\text{st state is in } B\text{”})$ denotes the union of all cylinder sets $\mathit{Cyl}(t_0\, t_1 \ldots t_m)$ spanned by a finite path fragment of length m that ends in a B-state. That is, $t_0\, t_1 \ldots t_m \in \mathit{Paths}_{\mathit{fin}}(\mathcal{M})$ and $t_m \in B$. Let us prove the equality in the above equation.

1. $\subseteq$: Let $\pi = s_0\, s_1\, s_2 \ldots$ be a path in $\mathcal{M}$ such that $\pi \models \Box\Diamond B$. Thus, for all indices n there exists some $m \geqslant n$ such that $s_m \in B$. But then $s_0\, s_1 \ldots\ldots s_m$ is a finite path fragment which ends in some state in T and whose cylinder set contains π. Thus, for all $n \geqslant 0$ there exists $m \geqslant n$ such that

   $$\pi \in \mathit{Cyl}(s_0 \ldots s_m) \subseteq \mathit{Cyl}(\text{“}(m+1)\text{-st state is in } B\text{”}).$$

   This shows that π belongs to the set on the right.

2. $\supseteq$: Let $\pi = s_0\, s_1\, s_2 \ldots$ be a path in $\bigcap_{n\geqslant 0} \bigcup_{m\geqslant n} \mathit{Cyl}(\text{“}(m{+}1)\text{st state} \in B\text{”})$. Thus, for each n there exists some $m \geqslant n$ and a finite path fragment $t_0\, t_1 \ldots t_m$ such that $t_m \in B$ and $\pi \in \mathit{Cyl}(t_0\, t_1 \ldots t_m)$. But then $t_0\, t_1 \ldots t_m$ is a prefix of π and $s_m = t_m \in B$. As this holds for any n, it follows that $\pi \models \Box\Diamond B$.

These arguments can be generalized to establish the measurability of events stating that finite path fragments, like $\widehat{\pi}$, appear infinitely often. This can be seen as follows. Let

$\widehat{\pi} = t_0 \ldots t_k$ be a finite path fragment in $\mathcal{M}$ and $Cyl(\text{“}\widehat{\pi}$ is taken from the mth state”) the union of all cylinder sets $Cyl(s_0 \ldots s_{m-1}\ t_0 \ldots t_k)$. Then, the set of paths satisfying $\Box\Diamond\widehat{\pi}$, i.e., the set of paths that contain infinitely many occurrences of $\widehat{\pi}$ is measurable, since it is given by:

$$\bigcap_{n \geqslant 0} \bigcup_{m \geqslant n} Cyl(\text{“}\widehat{\pi} \text{ is taken from the } m\text{th state”}).$$

Now let Π be a set of finite path fragments. The above yields that the event $\bigwedge_{\widehat{\pi} \in \Pi} \Box\Diamond\widehat{\pi}$, stating that each finite path fragment $\widehat{\pi} \in \Pi$ is taken infinitely often, is measurable too. Note that since the state space S of a Markov chain is countable (see Definition 10.1, page 747), $Paths_{fin}(\mathcal{M})$ is countable.

Let us now consider *persistence* properties, i.e., events of the form $\Diamond\Box B$. Such events consist of all paths $\pi = s_0\, s_1\, s_2 \ldots$ such that, for some $n \geqslant 0$, the suffix $s_n\, s_{n+1} \ldots$ only contains states in B. The event $\Diamond\Box B$ is measurable as it is the complement of the measurable event $\Box\Diamond(S \setminus B)$. ■

While the infinite repetition of a nondeterministic choice in a transition system does not impose any restrictions on the sequence of selected alternatives, randomization (as in Markov chains) somehow implies strong fairness for all transitions. This follows from the following theorem which states that under the assumption that a certain state t, say, of a Markov chain is visited infinitely often, then almost surely all finite path fragments $t_0\, t_1 \ldots t_n$ that start in t (i.e., $t_0 = t$) will be taken infinitely often too. Here, the notion “almost surely” refers to the *conditional probabilities*, under the condition that t is visited infinitely often. Formally, an event E is said to hold almost surely under the condition of another event D if $Pr(D) = Pr(E \cap D)$.

Theorem 10.25. Probabilistic Choice as Strong Fairness

For (possibly infinite) Markov chain $\mathcal{M}$ and s, t states in $\mathcal{M}$:

$$Pr^{\mathcal{M}}(s \models \Box\Diamond t) = Pr_s^{\mathcal{M}}\Big(\bigwedge_{\widehat{\pi} \,\in\, Paths_{fin}(t)} \Box\Diamond\widehat{\pi} \Big)$$

where $\bigwedge_{\widehat{\pi} \,\in\, Paths_{fin}(t)} \Box\Diamond\widehat{\pi}$ denotes the set of all paths π such that any path fragment $\widehat{\pi} \in Paths_{fin}(t)$ occurs infinitely often in π.

In particular, for each state $t \in S$ and $u \in Post(t)$ the event “transition $t \to u$ is taken infinitely often” holds almost surely given that t is visited infinitely often. In this sense, executions of Markov chains are strongly fair with respect to all probabilistic choices.

Proof: The proof is provided in three steps.

1. We first prove that for any $\widehat{\pi} \in \mathit{Paths}_{\mathit{fin}}(t)$ it holds that:

$$\mathit{Pr}(s \models \Box\Diamond t) \;=\; \mathit{Pr}(s \models \Diamond\widehat{\pi}).$$

Let $p = \mathbf{P}(\widehat{\pi})$. As $\widehat{\pi} \in \mathit{Paths}_{\mathit{fin}}(t)$, it follows that $0 < p \leqslant 1$. Let $E_n(\widehat{\pi})$ be the event "visit t at least n times, but never take the path fragment $\widehat{\pi}$". Note that $E_1(\widehat{\pi}) \supseteq E_2(\widehat{\pi}) \supseteq \ldots$. Furthermore:

$$\mathit{Pr}_s(E_n(\widehat{\pi})) \;\leqslant\; (1-p)^n.$$

Let the event $E(\widehat{\pi}) = \bigcap_{n \geqslant 1} E_n(\widehat{\pi})$. That is, $E(\widehat{\pi})$ is the event that t is visited infinitely often, but the path fragment $\widehat{\pi}$ is never taken. As $E_1(\widehat{\pi}) \supseteq E_2(\widehat{\pi}) \supseteq \ldots$ and

$$\mathit{Pr}_s(E(\widehat{\pi})) \;=\; \lim_{n\to\infty} \mathit{Pr}_s(E_n(\widehat{\pi})) \;\leqslant\; \lim_{n\to\infty} (1-p)^n \;=\; 0,$$

we have that:

$$\mathit{Pr}(s \models \Box\Diamond t \wedge \text{"path fragment } \widehat{\pi} \text{ is never taken"}) \;=\; 0$$

2. Now consider the event $F_n(\widehat{\pi})$, for $n \geqslant 0$, that represents the fact that state t is visited infinitely often, while the path fragment $\widehat{\pi}$ will not be taken anymore from the nth state on. The probability for the event $F_n(\widehat{\pi})$ is given by

$$\mathit{Pr}_s(F_n(\widehat{\pi})) \;=\; \sum_{s' \in S} \mathit{Pr}\Big(\underbrace{s \models \bigcirc^n s'}_{\text{the } n\text{-th state is } s'} \Big) \cdot \underbrace{\mathit{Pr}_{s'}(E(\widehat{\pi}))}_{=0} \;=\; 0.$$

Now consider the event

$$F(\widehat{\pi}) \;=\; \Box\Diamond t \wedge \text{"from some moment on, the path fragment } \widehat{\pi} \text{ is never taken"}.$$

It follows that $F(\widehat{\pi}) = \bigcup_{n \geqslant 1} F_n(\widehat{\pi})$. Since $F_1 \subseteq F_2 \subseteq \ldots$ it follows that

$$\mathit{Pr}_s(F(\widehat{\pi})) \;=\; \lim_{n\to\infty} \mathit{Pr}_s(F_n(\widehat{\pi})) = 0.$$

Hence:

$$\begin{aligned} & \mathit{Pr}(s \models \Box\Diamond t \;\wedge\; \Box\Diamond \text{"path fragment } \widehat{\pi} \text{ is taken"}) \\ = \;& \mathit{Pr}(s \models \Box\Diamond t) \;-\; \mathit{Pr}_s(F(\widehat{\pi})) \\ = \;& \mathit{Pr}(s \models \Box\Diamond t). \end{aligned}$$

3. We now generalize this result to *all* finite path fragments starting in state t. Let the event

$$F \;=\; \bigcup_{\widehat{\pi} \in \mathit{Paths}_{\mathit{fin}}(t)} F(\widehat{\pi}).$$

As the set of finite path fragments is countable we have

$$\mathit{Pr}_s(F) \;\leqslant\; \sum_{\widehat{\pi}} \mathit{Pr}_s(F(\widehat{\pi})) \;=\; 0$$

where $\widehat{\pi}$ ranges over all finite path fragments starting in t. Hence, $\mathit{Pr}_s(F) = 0$.

Thus, under the condition that state t is visited infinitely often, almost surely any finite path fragment starting in t will be taken infinitely often. ■

As a direct consequence of Theorem 10.25, it follows that any successor of t is visited infinitely often:

$$Pr^{\mathcal{M}}(s \models \Box\Diamond t) \;=\; Pr^{\mathcal{M}}(s \models \bigwedge_{u \,\in\, Post^*(t)} \Box\Diamond u).$$

Hence, for any state s it holds that

$$Pr(s \models \bigwedge_{t\in S} \; \bigwedge_{u\in Post^*(t)} (\Box\Diamond t \;\rightarrow\; \Box\Diamond u)) \;=\; 1.$$

Notation 10.26. Graph Notations for Markov Chains

Let $\mathcal{M} = (S, \mathbf{P}, \iota_{\text{init}}, AP, L)$ be a finite MC. In the sequel, we often use graph-theoretical notations for MCs which refer to the underlying digraph of $\mathcal{M}$. For example, a subset T of S is called *strongly connected* if for each pair (s,t) of states in T there exists a path fragment $s_0\, s_1 \ldots s_n$ such that $s_i \in T$ for $0 \leqslant i \leqslant n$, $s_0 = s$ and $s_n = t$. A *strongly connected component* (SCC, for short) of $\mathcal{M}$ denotes a strongly connected set of states such that no proper superset of T is strongly connected. A *bottom* SCC (BSCC, for short) of $\mathcal{M}$ is an SCC T from which no state outside T is reachable, i.e., for each state $t \in T$ it holds that $\mathbf{P}(t, T) = 1$. (Recall that $\mathbf{P}(s,T) = \sum_{t\in T} \mathbf{P}(s,t)$.) Let $BSCC(\mathcal{M})$ denote the set of all BSCCs of the underlying digraph of $\mathcal{M}$. ■

Let us now apply Theorem 10.25 to finite MCs. They enjoy the property that at least one state is visited infinitely often on each infinite path. The following result asserts that the set of states that is visited infinitely often on a path almost surely forms a BSCC. Before presenting this result, we have to ensure that the event "being in a given BSCC" is measurable. This can be seen as follows. Recall that $\inf(\pi)$ denotes the set of states that are visited infinitely often along π. The event $\inf(\pi) = T$ for some BSCC T is measurable as it can be written as a finite intersection of measurable sets (cf. Remark 10.24 on page 771):

$$\bigwedge_{t\in T} \Box\Diamond t \;\wedge\; \Diamond\Box T.$$

The set of paths for which $\inf(\pi)$ is a BSCC agrees with the measurable event

$$\bigvee_{T\in BSCC(\mathcal{M})} (\inf(\pi) = T).$$

Theorem 10.27. Limit Behavior of Markov Chains

For each state s of a finite Markov chain $\mathcal{M}$:

$$Pr_s^{\mathcal{M}}\{\, \pi \in Paths(s) \mid \inf(\pi) \in BSCC(\mathcal{M}) \,\} = 1.$$

Proof: For each path π, the set $\inf(\pi)$ is strongly connected, and thus contained in some SCC of $\mathcal{M}$. Hence, for each state s:

$$\sum_{T} Pr_s\{\, \pi \in Paths(s) \mid \inf(\pi) = T \,\} = 1 \qquad (*)$$

where T ranges over all nonempty SCCs of $\mathcal{M}$. Assume that $Pr_s\{\, \pi \in Paths(s) \mid \inf(\pi) = T \,\}$ is positive. By Theorem 10.25, almost all paths π with $\inf(\pi) = T$ fulfill

$$Post^*(T) = Post^*(\inf(\pi)) \subseteq \inf(\pi) = T.$$

Hence, $T = Post^*(T)$, i.e., T is a BSCC. The claim now follows from $(*)$. ■

Stated in words, Theorem 10.27 states that almost surely any finite Markov chain eventually reaches a BSCC and visits all its states infinitely often.

Example 10.28. Zeroconf Protocol (Revisited)

Consider the Markov chain for the zeroconf protocol; see Example 10.5 (page 751). This MC has two BSCCs, namely $\{\, s_8 \,\}$ and $\{\, s_6 \,\}$. According to the previous theorem, any infinite path will almost surely lead to one of these BSCCs. This means in particular that the probability of infinitely often attempting to acquire a new address (after receiving no response on a probe) equals zero. ■

The previous theorem is a central result for the analysis of many types of properties of finite MCs. In the following, we consider three important almost sure properties: reachability, repeated reachability, and persistence properties. The problem of almost sure reachability amounts to determining the set of states that reach a certain given set of states B, say, almost surely. The following result provides the foundations for such a computation.

Theorem 10.29. Almost Sure Reachability

Let $\mathcal{M}$ be a finite Markov chain with state space S, $s \in S$ and $B \subseteq S$ a set of absorbing states. Then, the following statements are equivalent:

(a) $Pr(s \models \Diamond B) = 1$.

(b) $Post^*(t) \cap B \neq \varnothing$ *for each state* $t \in Post^*(s)$.

(c) $s \in S \setminus Pre^*(\, S \setminus Pre^*(B)\,)$.

In particular, $\{\, s \in S \mid Pr(s \models \Diamond B) = 1 \,\} = S \setminus Pre^*(\, S \setminus Pre^*(B)\,)$.

Proof:

1. (a) $\Longrightarrow$ (b): Let $t \in Post^*(s)$ such that $Post^*(t) \cap B = \varnothing$, i.e., t is a successor of s from which B cannot be reached. Then: $Pr(s \models \Diamond B) \leqslant 1 - Pr(s \models \Diamond t) < 1$. This shows that if (b) is violated, (a) is so too.

2. (b) $\Longleftrightarrow$ (c): By definition of *Post* and *Pre* it follows that for each $u \in S$ and $C \subseteq S$, $Post^*(u) \cap C \neq \varnothing$ if and only if $u \in Pre^*(C)$. Thus:

$$\begin{aligned} & Post^*(t) \cap B \neq \varnothing \text{ for any state } t \in Post^*(s) \\ \text{iff} \quad & Post^*(s) \subseteq Pre^*(B) \\ \text{iff} \quad & Post^*(s) \cap (S \setminus Pre^*(B)) = \varnothing \\ \text{iff} \quad & s \notin Pre^*(\, S \setminus Pre^*(B)\,) \\ \text{iff} \quad & s \in S \setminus Pre^*(\, S \setminus Pre^*(B)\,). \end{aligned}$$

3. (b) $\Longrightarrow$ (a): Assume $Post^*(t) \cap B \neq \varnothing$ for any successor t of s. By Theorem 10.27, almost all paths $\pi \in Paths(s)$ reach a BSCC. Since each state in B is absorbing, each BSCC T of $\mathcal{M}$ is either of the form $T = \{\, t \,\}$ with $t \in B$ or it satisfies $T \cap B = \varnothing$. Let us show that the latter case cannot occur. Consider $T \cap B = \varnothing$ for BSCC T. As T is a BSCC, $Post^*(u) \cap B = \varnothing$ for each $u \in T$. However, as B is reachable from any successor of s, there is no BSCC T with $T \cap B = \varnothing$ that is reachable from s. Hence, from state s almost surely a state in B will be reached.

■

Thus, if all states in a finite Markov chain can reach a set B of absorbing states, then B will be reached almost surely from each state. This, however, also implies that almost surely B will be visited infinitely often, as stated by the following corollary:

Corollary 10.30. Global Almost Sure Reachability

Let $\mathcal{M}$ be a finite Markov chain with state space S, and $B \subseteq S$. Then:

$$s \models \exists\Diamond B \text{ for any } s \in S \quad \text{implies} \quad Pr(s \models \Box\Diamond B) = 1 \text{ for all } s \in S.$$

Theorem 10.29 suggests the following algorithm to determine the set of states that reach a certain set B of states almost surely in a finite MC $\mathcal{M}$:

1. Make all states in B absorbing. This yields MC $\mathcal{M}_B$.
2. Determine the set $S \setminus Pre^*(S \setminus Pre^*(B))$ by a graph analysis in $\mathcal{M}_B$. This can be done by a backward search from B to compute $Pre^*(B)$ followed by a backward search from $S \setminus Pre^*(B)$ to determine $Pre^*(S \setminus Pre^*(B))$. Both searches are carried out on $\mathcal{M}_B$.

Thus, the time complexity of determining the set of states s for which $Pr(s \models \Diamond B) = 1$ is linear in the size of $\mathcal{M}$.

Now consider a constrained reachability condition $C \,\mathsf{U}\, B$ where C and B are subsets of the state space of a finite MC $\mathcal{M}$.

Corollary 10.31. Qualitative Constrained Reachability

For finite Markov chain $\mathcal{M}$ with state space S and $B, C \subseteq S$, the sets

$$S_{=0} = \{\, s \in S \mid Pr(s \models C \,\mathsf{U}\, B) = 0 \,\} \quad \text{and} \quad S_{=1} = \{\, s \in S \mid Pr(s \models C \,\mathsf{U}\, B) = 1 \,\}$$

can be computed in time $\mathcal{O}(size(\mathcal{M}))$.

Proof: Let $\mathcal{M}$ be a finite MC and $B, C \subseteq S$.

1. The set $S_{=0} = \{\, s \in S \mid Pr(s \models C \,\mathsf{U}\, B) = 0 \,\}$ agrees with the complement of the satisfaction set of the CTL formula $\exists(C \,\mathsf{U}\, B)$. This set can be computed by means of a backward analysis starting from the B-states in time linear in the size of $\mathcal{M}$.
2. A linear-time algorithm for the computation of $S_{=1} = \{\, s \in S \mid Pr(s \models C \,\mathsf{U}\, B) = 1 \,\}$ is obtained as follows. In fact, the problem of calculating $S_{=1}$ can be solved by a reduction to the problem of computing the set of states that almost surely eventually

reach B in a slightly modified Markov chain. The idea is to make all B-states absorbing and do the same for all states in $S \setminus (C \cup B)$. To that end, $\mathcal{M}$ is changed into the Markov chain $\mathcal{M}'$ with state space S and transition probability function $\mathbf{P}'$ given by

$$\mathbf{P}'(s,t) \;=\; \begin{cases} 1 & : \text{ if } s \in B \cup S \setminus (C \cup B) \\ \mathbf{P}(s,t) & : \text{ otherwise.} \end{cases}$$

The justification for the transformation of $\mathcal{M}$ into $\mathcal{M}'$ is given by

- $Pr^{\mathcal{M}}(s \models C \,\mathsf{U}\, B) \;=\; Pr^{\mathcal{M}'}(s \models \Diamond B)$ for all states $s \in C \setminus B$,
- $Pr^{\mathcal{M}}(s \models C \,\mathsf{U}\, B) \;=\; Pr^{\mathcal{M}'}(s \models \Diamond B) = 1$ for all states $s \in B$,
- $Pr^{\mathcal{M}}(s \models C \,\mathsf{U}\, B) \;=\; Pr^{\mathcal{M}'}(s \models \Diamond B) = 0$ for all states $s \in S \setminus (C \cup B)$.

Thus, the probabilities for the constrained reachability property $C \,\mathsf{U}\, B$ in $\mathcal{M}$ can be determined by computing the reachability probabilities $\Diamond B$ in $\mathcal{M}'$. The latter can be done in time linear in the size of $\mathcal{M}'$ (see above), which is bounded from above by that of $\mathcal{M}$. (Recall that $size(\mathcal{M})$ is the number of states and transitions in $\mathcal{M}$.) ■

Example 10.32. Qualitative Constrained Reachability

Consider again the craps game, see Example 10.4 (page 750). As before, let $B = \{\, won \,\}$ and $C = \{\, start, 4, 5, 6 \,\}$ and consider the event $C \,\mathsf{U}\, B$. We have

$$S_{=0} = S \setminus Sat(\exists(C \,\mathsf{U}\, B)) = \{\, lost, 8, 9, 10 \,\}.$$

To determine the set $S_{=1}$ all states in B and in $S \setminus (B \cup C)$ are made absorbing. This yields the MC in Figure 10.5 (page 770). The states that almost surely reach the state *won* in this MC are $\{\, 4, 5, 6, won \,\}$. These are the states in $S_{=1}$. ■

Using the above result, the qualitative model-checking problem to determine whether $Pr^{\mathcal{M}}(C \,\mathsf{U}\, B) = 1$ can be solved by a graph analysis in time linear in the size of $\mathcal{M}$. In fact, the same holds for repeated reachability events:

Corollary 10.33. Qualitative Repeated Reachability

Let $\mathcal{M}$ be a finite Markov chain with state space S, $B \subseteq S$, and $s \in S$. Then, the following statements are equivalent:

(a) $Pr(s \models \Box\Diamond B) \;=\; 1$.

(b) $T \cap B \neq \varnothing$ *for each BSCC* T *that is reachable from* s.

(c) $s \models \forall\Box\exists\Diamond B$.

Proof: The equivalence of (a) and (b) follows immediately from the fact that from state s almost surely a BSCC T will be reached and all states of T will be visited infinitely often (see Theorem 10.27). The equivalence of (a)/(b) and (c) is left as an exercise to the reader (see Exercise 10.5 on page 901). ■

As a consequence, to check whether $Pr^{\mathcal{M}}(\Box\Diamond B) = 1$, it suffices to analyze the BSCCs in $\mathcal{M}$, which can be done in linear time. The following result asserts that repeated reachability probabilities can be computed in time polynomial in the size of $\mathcal{M}$. This is achieved by a reduction to the problem of computing the probabilities for eventually visiting certain BSCCs:

Corollary 10.34. Quantitative Repeated Reachability

Let $\mathcal{M}$ be a finite Markov chain with state space S, $B \subseteq S$, and $s \in S$, and U be the union of all BSCCs T (in $\mathcal{M}$) with $T \cap B \neq \varnothing$. Then:

$$Pr(s \models \Box\Diamond B) \;=\; Pr(s \models \Diamond U).$$

In a similar way, by an analysis of the BSCCs it can also be checked whether a strong or weak fairness constraint (or other ω-regular liveness property) holds almost surely. For persistence property $\Diamond\Box B$ with $B \subseteq S$, for instance, we have

- $Pr(s \models \Diamond\Box B) = 1$ iff $T \subseteq B$ for each BSCC T that is reachable from s.
- $Pr(s \models \Diamond\Box B) = Pr(s \models \Diamond V)$ where V is the union of all BSCCs T with $T \subseteq B$.

The conclusion of this section is that checking a qualitative property, such as reachability, repeated reachability, and persistence on a finite Markov chain can be achieved by a graph analysis. The transition probabilities thus are of no importance for this purpose! As the following example indicates, this does not hold for infinite Markov chains.

Remark 10.35. Qualitative Properties in Infinite Markov Chains

We state without proof that the above considerations on the limiting behavior of finite Markov chains do *not* hold for *infinite* Markov chains. In fact, even for strongly connected infinite Markov chains it is possible that the probability of visiting a certain state infinitely

often is zero. Thus, it is possible that $Pr(s \models \Diamond T) > 0$ for all states s, while $Pr(s \models \Box\Diamond T) = 0$. Typical examples are so-called one-dimensional *random walks*. A concrete example is the infinite Markov chain $\mathcal{M}_p$ where p is a rational number in $]0,1[$. The state space of $\mathcal{M}_p$ is $S = \mathbb{N}$ with $L(0) = \{\,0\,\}$ and $L(n) = \varnothing$ for $n > 0$. The transition probabilities in $\mathcal{M}_p$ are given by (see Figure 10.6):

$$\mathbf{P}(n, n{+}1) = p \qquad \text{and} \qquad \mathbf{P}(n, n{-}1) = 1 - p \quad \text{for all } n > 0$$
$$\mathbf{P}(0,0) = 1 - p \qquad \text{and} \qquad \mathbf{P}(0,1) = p.$$

It can be shown that for $p \leqslant \frac{1}{2}$, the leftmost state is almost surely eventually reached, and almost surely infinitely often visited, i.e.:

$$Pr(n \models \Diamond 0) = Pr(n \models \Box\Diamond 0) = 1 \quad \text{for all } n \geqslant 0.$$

However, for $p > \frac{1}{2}$, it is more likely to move to the right (i.e., to move from state n to state $n{+}1$) than to the left, and it can be shown that $Pr(n \models \Diamond 0) < 1$ and $Pr(n \models \Box\Diamond 0) = 0$ for all $n > 0$. (Further details on infinite Markov chains and random walks can be found in most textbook on Markov chains; see e.g., [63, 238, 248].) ■

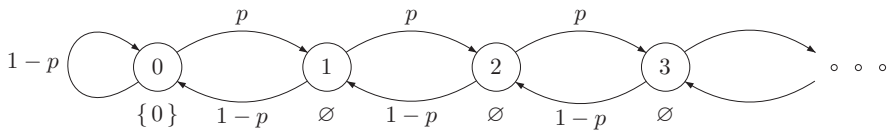


Figure 10.6: Infinite Markov chain for a one-dimensional random walk.

10.2 Probabilistic Computation Tree Logic

Probabilistic computation tree logic (PCTL, for short) is a branching-time temporal logic, based on the logic CTL (see Chapter 6). A PCTL formula formulates conditions on a state of a Markov chain. The interpretation is Boolean, i.e., a state either satisfies or violates a PCTL formula. The logic PCTL is defined like CTL with one major difference. Instead of universal and existential path quantification, PCTL incorporates, besides the standard propositional logic operators, the probabilistic operator $\mathbb{P}_J(\varphi)$ where φ is a path formula and J is an interval of $[0,1]$. The path formula φ imposes a condition on the set of paths, whereas J indicates a lower bound and/or upper bound on the probability. The intuitive meaning of the formula $\mathbb{P}_J(\varphi)$ in state s is: the probability for the set of paths satisfying φ and starting in s meets the bounds given by J. The probabilistic operator can be considered as the quantitative counterpart to the CTL path quantifiers $\exists$ and $\forall$. The CTL formulae $\exists\varphi$ and $\forall\varphi$ assert the existence of certain paths and the absence of paths where a certain condition does not hold respectively. They, however, do not impose any

constraints on the likelihood of the paths that satisfy the condition φ. Later on in this section, the relationship between the operator $\mathbb{P}_J(\varphi)$ and universal and existential path quantification is elaborated in detail.

The path formulae φ are defined as for CTL, except that a bounded until operator is additionally incorporated. The intuitive meaning of the path formula $\Phi \,\mathsf{U}^{\leqslant n}\, \Psi$ for a natural number n is that a Ψ state should be reached within n transitions, and that all states prior to reaching the Ψ-state satisfy Φ.

Definition 10.36. Syntax of PCTL

PCTL *state formulae* over the set AP of atomic propositions are formed according to the following grammar:

$$\Phi \;::=\; \text{true} \;\Big|\; a \;\Big|\; \Phi_1 \wedge \Phi_2 \;\Big|\; \neg\Phi \;\Big|\; \mathbb{P}_J(\varphi)$$

where $a \in AP$, φ is a path formula and $J \subseteq [0,1]$ is an interval with rational bounds. PCTL *path formulae* are formed according to the following grammar:

$$\varphi \;::=\; \bigcirc \Phi \;\Big|\; \Phi_1 \,\mathsf{U}\, \Phi_2 \;\Big|\; \Phi_1 \,\mathsf{U}^{\leqslant n}\, \Phi_2$$

where Φ, Φ_1, and Φ_2 are state formulae and $n \in \mathbb{N}$. ■

As in CTL, the linear temporal operators $\bigcirc$ and U (and its bounded variant) are required to be immediately preceded by $\mathbb{P}$. Rather than writing the intervals explicitly, often abbreviations are used; e.g., $\mathbb{P}_{\leqslant 0.5}(\varphi)$ denotes $\mathbb{P}_{[0,0.5]}(\varphi)$, $\mathbb{P}_{=1}(\varphi)$ stands for $\mathbb{P}_{[1,1]}(\varphi)$, and $\mathbb{P}_{>0}(\varphi)$ denotes $\mathbb{P}_{]0,1]}(\varphi)$.

The propositional logic fragment of PCTL, as well as the path formulae $\bigcirc \Phi$ and $\Phi_1 \,\mathsf{U}\, \Phi_2$ has the same meaning as in CTL. Path formula $\Phi_1 \,\mathsf{U}^{\leqslant n} \Phi_2$ is the *step-bounded* variant of $\Phi_1 \,\mathsf{U}\, \Phi_2$. It asserts that the event specified by Φ_2 will hold within at most n steps, while Φ_1 holds in all states that are visited before a Φ_2-state has been reached. Other Boolean connectives are derived in the usual way, e.g., $\Phi_1 \vee \Phi_2$ is obtained by $\neg(\neg\Phi_1 \wedge \neg\Phi_2)$. The eventually operator ($\Diamond$) can be derived as usual: $\Diamond\Phi \;=\; \text{true} \,\mathsf{U}\, \Phi$. Similarly, for step-bounded eventually we have:

$$\Diamond^{\leqslant n}\Phi \;=\; \text{true} \,\mathsf{U}^{\leqslant n}\, \Phi.$$

A path satisfies $\Diamond^{\leqslant n}\Phi$ if it reaches a Φ-state within n steps.

The always operator can be derived using the duality of eventually and always (as in CTL and LTL) and the duality of lower and upper bounds. The latter means that an event E

holds with probability at most p if and only if the dual event $\bar{E} = \mathit{Paths}(\mathcal{M}) \setminus E$ holds with probability at least $1-p$. Thus, it is possible to define, e.g.,

$$\mathbb{P}_{\leqslant p}(\Box\Phi) \;=\; \mathbb{P}_{\geqslant 1-p}(\Diamond\neg\Phi) \quad \text{and} \quad \mathbb{P}_{]p,q]}(\Box^{\leqslant n}\Phi) \;=\; \mathbb{P}_{[1-q,1-p[}(\Diamond^{\leqslant n}\neg\Phi).$$

Other temporal operators, like weak until W or release R (see Section 5.1.5, page 252) can be derived in an analogous way. This is left as an exercise to the reader; see Exercise 10.9 (page 902).

Example 10.37. Specifying Properties in PCTL

Consider the simulation of a six-sided die by a fair coin as described in Example 10.3 (page 750). The PCTL formula

$$\bigwedge_{1\leqslant i\leqslant 6} \mathbb{P}_{=\frac{1}{6}}(\Diamond i)$$

expresses that each of the six possible outcomes of the die should occur with equal probability.

Consider a communication protocol that uses an imperfect channel that might lose messages. The PCTL formula

$$\mathbb{P}_{=1}(\Diamond \mathit{delivered}) \;\wedge\; \mathbb{P}_{=1}\Big(\,\Box\,(\,\mathit{try\_to\_send} \;\rightarrow\; \mathbb{P}_{\geqslant 0.99}(\Diamond^{\leqslant 3}\mathit{delivered})\,)\,\Big)$$

asserts that almost surely some message will be delivered (first conjunct) and that almost surely for any attempt to send a message, with probability at least 0.99, the message will be delivered within three steps.

Consider the craps game, see Example 10.4. The property "the probability of winning without ever rolling 8, 9, or 10 is at least 0.32" is expressed by the PCTL formula:

$$\mathbb{P}_{\geqslant 0.32}(\neg(8 \;\vee\; 9 \;\vee\; 10)\,\mathsf{U}\,\mathit{won}).$$

Impatient players might be interested in this property but with a bounded number of rolls, say, five. This is expressed by

$$\mathbb{P}_{\geqslant 0.32}(\neg(8 \;\vee\; 9 \;\vee\; 10)\,\mathsf{U}^{\leqslant 5}\,\mathit{won}).$$

Finally, the PCTL formula

$$\mathbb{P}_{\geqslant 0.32}(\neg(8 \;\vee\; 9 \;\vee\; 10)\,\mathsf{U}^{\leqslant 5}\,\mathbb{P}_{=1}(\Box\mathit{won}))$$

expresses in addition that almost surely the player will always win. ■

PCTL formulae are interpreted over the states and paths of a Markov chain $\mathcal{M}$. For the state formulae, the satisfaction relation $\models$ is a relation between states in $\mathcal{M}$ and state

formulae. As usual, we write $s \models \Phi$ rather than $(s, \Phi) \in \models$. The interpretation is as usual, e.g., $s \models \Phi$ if and only if Φ holds in s. For the path formulae, $\models$ is a relation between infinite path fragments in $\mathcal{M}$ and path formulae.

Definition 10.38. Satisfaction Relation for PCTL

Let $a \in AP$ be an atomic proposition, $\mathcal{M} = (S, \mathbf{P}, \iota_{\text{init}}, AP, L)$ be a Markov chain, state $s \in S$, Φ, Ψ be PCTL state formulae, and φ be a PCTL path formula. The satisfaction relation $\models$ is defined for state formulae by

$$\begin{aligned}
s &\models a && \text{iff} && a \in L(s),\\
s &\models \neg\Phi && \text{iff} && s \not\models \Phi,\\
s &\models \Phi \wedge \Psi && \text{iff} && s \models \Phi \text{ and } s \models \Psi,\\
s &\models \mathbb{P}_J(\varphi) && \text{iff} && Pr(s \models \varphi) \in J.
\end{aligned}$$

Here, $Pr(s \models \varphi) = Pr_s\{\, \pi \in Paths(s) \mid \pi \models \varphi \,\}$.

Given a path π in $\mathcal{M}$, the satisfaction relation is defined (as for CTL):

$$\begin{aligned}
\pi &\models \bigcirc \Phi && \text{iff} && \pi[1] \models \Phi,\\
\pi &\models \Phi \,\mathsf{U}\, \Psi && \text{iff} && \exists\, j \geqslant 0.\ (\pi[j] \models \Psi \ \wedge\ (\forall\, 0 \leqslant k < j.\, \pi[k] \models \Phi)),\\
\pi &\models \Phi \,\mathsf{U}^{\leqslant n}\, \Psi && \text{iff} && \exists\, 0 \leqslant j \leqslant n.\ (\pi[j] \models \Psi \ \wedge\ (\forall\, 0 \leqslant k < j.\, \pi[k] \models \Phi))
\end{aligned}$$

where for path $\pi = s_0\, s_1\, s_2 \ldots$ and integer $i \geqslant 0$, $\pi[i]$ denotes the $(i{+}1)$-st state of π, i.e., $\pi[i] = s_i$. ■

Let $Sat_{\mathcal{M}}(\Phi)$, or briefly $Sat(\Phi)$, denote $\{\, s \in S \mid s \models \Phi \,\}$.

The semantics of the probability operator $\mathbb{P}$ refers to the probability for the sets of paths for which a path formula holds. To ensure that this is well-defined, we need to establish that the events specified by PCTL path formulae are measurable, i.e., elements of the σ-algebra $\mathfrak{E}^{\mathcal{M}}$ (see Definition 10.10, page 758). As the set $\{\, \pi \in Paths(s) \mid \pi \models \varphi \,\}$ for PCTL path formula φ can be considered as a countable union of cylinder sets, its measurability is ensured. This follows from the following lemma.

Lemma 10.39. Measurability of PCTL Events

For each PCTL path formula φ and state s of a Markov chain $\mathcal{M}$, the set $\{\, \pi \in Paths(s) \mid \pi \models \varphi \,\}$ is measurable.

Proof: For any PCTL path formula φ, it will be proven that the set $Paths(s, \varphi) = \{\, \pi \in$

$Paths(s) \mid \pi \models \varphi\}$ is measurable, i.e., belongs to the σ-algebra $\mathfrak{E}^{\mathcal{M}}$. This is done by showing that this set can be considered as the countable union of cylinder sets in $\mathfrak{E}^{\mathcal{M}}$.

There are three possibilities for φ. If $\varphi = \bigcirc \Phi$, then $Paths(s, \varphi)$ agrees with the union of the cylinder sets $Cyl(s\,t)$ where $t \models \Phi$. For $\varphi = \Phi \,\mathsf{U}^{\leqslant n} \Psi$, the set $Paths(s, \varphi)$ is obtained by the union of all cylinder sets $Cyl(s_0\, s_1\, \ldots s_k)$ where $k \leqslant n$, $s_k \models \Psi$, $s_0 = s$, and $s_i \models \Phi$ for $0 \leqslant i < k$. For unbounded until, i.e., $\varphi = \Phi \,\mathsf{U}\, \Psi$, we have:

$$Paths(s, \varphi) \;=\; \bigcup_{n \geqslant 0} \{\, \pi \in Paths(s) \mid \pi \models \Phi \,\mathsf{U}^{\leqslant n} \Psi \,\}.$$

■

The *equivalence* of PCTL formulae is defined as for the other logics encountered so far: two state formulae are equivalent whenever their semantics are equal. Formally, for PCTL state formulae Φ and Ψ:

$$\begin{array}{lll} \Phi \equiv \Psi & \text{iff} & \text{for all Markov chains } \mathcal{M} \text{ and states } s \text{ of } \mathcal{M} \text{ it holds that:} \\ & & \qquad s \models \Phi \iff s \models \Psi \\ & \text{iff} & Sat_{\mathcal{M}}(\Phi) = Sat_{\mathcal{M}}(\Psi) \text{ for all Markov chains } \mathcal{M}. \end{array}$$

In addition to the equivalence rules for propositional logic, we have, e.g.,

$$\mathbb{P}_{<p}(\varphi) \;\equiv\; \neg \mathbb{P}_{\geqslant p}(\varphi)$$

where $p \in \,]0, 1]$ is a rational number and φ an arbitrary PCTL path formula. This equivalence follows directly from the PCTL semantics. Thus, the expressiveness of PCTL does not change when only permitting the upper bounds on the path probabilities ($< p$ and $\leqslant p$) and one of the (qualitative) bounds $= 0$ or $= 1$. Note that, e.g.,

$$\mathbb{P}_{]0.3,0.7]}(\varphi) \;\equiv\; \neg \mathbb{P}_{\leqslant 0.3}(\varphi) \;\wedge\; \mathbb{P}_{\leqslant 0.7}(\varphi).$$

Another example is the following duality law:

$$\mathbb{P}_{>0}(\,\bigcirc\, \mathbb{P}_{>0}(\Diamond \Phi)\,) \;\equiv\; \mathbb{P}_{>0}(\,\Diamond\, \mathbb{P}_{>0}(\bigcirc\, \Phi)\,).$$

This is proven as follows. First, consider $\Rightarrow$. Let s be a state of Markov chain $\mathcal{M}$ with $s \models \mathbb{P}_{>0}(\,\bigcirc\, \mathbb{P}_{>0}(\Diamond \Phi)\,)$. Then, for some direct successor t of s, it holds that $t \models \mathbb{P}_{>0}(\Diamond \Phi)\,)$. This implies that there exists a finite path fragment $t_0\, t_1 \ldots t_n$ which starts in $t_0 = t$ and ends in state $t_n \in Sat(\Phi)$. So, $t_{n-1} \models \mathbb{P}_{>0}(\bigcirc\, \Phi)$. Since $s\, t_0\, t_1 \ldots t_{n-1}$ is a path fragment starting in s, it follows that $s \models \mathbb{P}_{>0}(\Diamond \mathbb{P}_{>0}(\bigcirc\, \Phi))$.

Consider $\Leftarrow$. Let $s \models \mathbb{P}_{>0}(\Diamond \mathbb{P}_{>0}(\bigcirc\, \Phi)\,)$. From the PCTL semantics it follows that

$$Pr(s \models \Diamond Sat(\mathbb{P}_{>0}(\bigcirc\, \Phi))) > 0.$$

Thus, there exists a path fragment $s_0\, s_1 \ldots s_n$ starting in $s_0 = s$ and ending in a state $s_n \in \mathit{Sat}(\mathbb{P}_{>0}(\bigcirc \Phi))$. Hence, s_n has a successor t with $t \models \Phi$. But then $s_1 \ldots s_n\, t$ is a witness for $s_1 \models \mathbb{P}_{>0}(\Diamond \Phi)$. Since $s_1 \in \mathit{Post}(s)$, it follows that $s \models \mathbb{P}_{>0}(\bigcirc\, \mathbb{P}_{>0}(\Diamond \Phi))$.

10.2.1 PCTL Model Checking

The PCTL model-checking problem is the following decision problem. Given a finite Markov chain $\mathcal{M}$, state s in $\mathcal{M}$, and PCTL state formula Φ, determine whether $s \models \Phi$. As for CTL model checking, the basic procedure is to compute the satisfaction set $\mathit{Sat}(\Phi)$. This is done recursively using a bottom-up traversal of the parse tree of Φ; see Algorithm 13 (page 342). The nodes of the parse tree represent the subformulae of Φ. For each node of the parse tree, i.e., for each subformula Ψ of Φ, the set $\mathit{Sat}(\Psi)$ is determined. For the propositional logic fragment of PCTL this is performed in exactly the same way as for CTL. The most interesting part is the treatment of subformulae of the form $\Psi = \mathbb{P}_J(\varphi)$. In order to determine whether $s \in \mathit{Sat}(\Psi)$, the probability $\mathit{Pr}(s \models \varphi)$ for the event specified by φ needs to be established. Then

$$\mathit{Sat}(\mathbb{P}_J(\varphi)) \;=\; \{s \in S \mid \mathit{Pr}(s \models \varphi) \in J\}.$$

For several path formulae φ, the computation of the probability $\mathit{Pr}(s \models \varphi)$ has, in fact, already been treated in previous sections.

Let us first consider the next-step operator. For $\varphi = \bigcirc \Psi$, the following equality holds:

$$\mathit{Pr}(s \models \bigcirc \Psi) \;=\; \sum_{s' \in \mathit{Sat}(\Psi)} \mathbf{P}(s, s')$$

where $\mathbf{P}$ is the transition probability function of $\mathcal{M}$. In matrix-vector notation we thus have that the vector $(\,\mathit{Pr}(s \models \bigcirc \Psi)\,)_{s \in S}$ can be computed by multiplying $\mathbf{P}$ with the characteristic vector for $\mathit{Sat}(\Psi)$, i.e., bit vector $(b_s)_{s \in S}$ where $b_s = 1$ if and only if $s \in \mathit{Sat}(\Psi)$. Checking the next-step operator thus reduces to a single matrix-vector multiplication.

The probability $\mathit{Pr}(s \models \varphi)$ for until formulae $\varphi = \Phi \,\mathsf{U}^{\leqslant n}\, \Psi$ or $\varphi = \Phi \,\mathsf{U}\, \Psi$ can be obtained using the techniques explained in Section 10.1.1 (see page 759ff). The events $C \,\mathsf{U}^{\leqslant n} B$ or $C \,\mathsf{U}\, B$, respectively, should be taken with $C = \mathit{Sat}(\Phi)$ and $B = \mathit{Sat}(\Psi)$. For the bounded until operator $\mathsf{U}^{\leqslant n}$, the vector $(\,\mathit{Pr}(s \models \varphi)\,)_{s \in S}$ can be obtained by $\mathcal{O}(n)$ vector-matrix multiplications. For the until operator, a linear equation system needs to be solved. In both cases, the dimension of the involved matrix is bounded by $N \times N$ where $N = |S|$ is the number of states in $\mathcal{M}$. This yields

Theorem 10.40. Time Complexity of PCTL Model Checking for MCs

For finite Markov chain $\mathcal{M}$ and PCTL formula Φ, the PCTL model-checking problem $\mathcal{M} \models \Phi$ can be solved in time

$$\mathcal{O}(\,\mathrm{poly}(\mathit{size}(\mathcal{M}))\,\cdot\, n_{\max} \cdot |\Phi|\,)$$

where $n_{\max}$ is the maximal step bound that appears in a subpath formula $\Psi_1 \,\mathsf{U}^{\leqslant n}\, \Psi_2$ of Φ (and $n_{\max} = 1$ if Φ does not contain a step-bounded until operator).

For efficiency reasons, in order to check the qualitative PCTL properties, such as $\mathbb{P}_{=1}(a \,\mathsf{U}\, b)$ or $\mathbb{P}_{>0}(a \,\mathsf{U}\, b)$, graph-based techniques are employed as described in the next subsection. This avoids solving a system of linear equations.

For CTL, counterexamples and witnesses can be provided as an additional diagnostic feedback when checking a transition system against a CTL formula $\exists\varphi$ or $\forall\varphi$. Alternatively, a CTL model checker may provide an output file containing the information on which states fulfill the subformulae. Recall that the satisfaction sets of all subformulae are computed by the CTL model checker anyway. The probabilistic analogue is to provide the information on the probabilities $Pr(s \models \varphi)$ for all subformulae $\mathbb{P}_J(\varphi)$. Note that these values $Pr(s \models \varphi)$ have been computed by the PCTL model checker. Although this information can be very helpful, for large state spaces it might be hard to extract any relevant information. To provide more comprehensive diagnostic information, the PCTL model-checking procedure can be extended to return *sets* of finite path fragments that constitute a witness for the satisfaction or refutation of subformulae $\mathbb{P}_J(\varphi)$. Let us explain this by means of (unconstrained) reachability properties. If $s \not\models \mathbb{P}_{\leqslant p}(\Diamond\Psi)$, then $Pr(s \models \Diamond\Psi) > p$. A proof for the latter is given by a finite set Π of finite path fragments $s_0\, s_1 \ldots s_n$ such that $s_0 = s$, $s_i \not\models \Psi$ for $0 \leqslant i < n$ and $s_n \models \Psi$ and $\sum_{\widehat{\pi}\in\Pi} \mathbf{P}(\widehat{\pi}) > p$. A witness of $s \not\models \mathbb{P}_{\leqslant p}(\Diamond\Psi)$ thus is a finite set of paths, all satisfying $\Diamond\Psi$, such that their total probability mass exceeds p.

Example 10.41. Counterexample

Consider the (abstract) Markov chain depicted in Figure 10.7 and assume the PCTL formula of interest is $\mathbb{P}_{\leqslant \frac{1}{2}}(\Diamond b)$. The fact that $s_0 \not\models \mathbb{P}_{\leqslant \frac{1}{2}}(\Diamond b)$ can be witnessed by, e.g., the set of finite paths:

$$\{\quad \underbrace{s_0\, s_1\, t_1}_{\text{probability } 0.2} \quad , \quad \underbrace{s_0\, s_1\, s_2\, t_1}_{\text{probability } 0.2} \quad , \quad \underbrace{s_0\, s_2\, t_1}_{\text{probability } 0.15} \quad \}$$

whose total probability exceeds the probability bound $\frac{1}{2}$. Note that counterexamples are not unique; e.g., replacing the finite path $s_0\, s_2\, t_1$ by $s_0\, s_1\, s_2\, t_1$ in the above set of paths also yields a counterexample. ■

An evidence of $s \not\models \mathbb{P}_{\geqslant p}(\Diamond\Psi)$—note that the only difference with the previous case is that

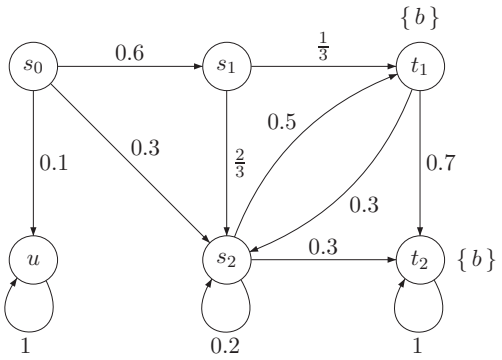


Figure 10.7: An example of a Markov chain.

p is an upper bound rather than a lower bound—is obtained by considering the set of finite paths that refute $\Diamond\Psi$ and showing that these paths occur with a probability that exceeds $1-p$. More precisely, consider the finite set Π of finite paths $s_0\, s_1 \ldots s_n$ such that $s_0 = s$, $s_i \not\models \Psi$ for $0 \leqslant i < n$ and s_n belongs to some BSCC C of $\mathcal{M}$ such that $C \cap \mathit{Sat}(\Psi) = \varnothing$. Moreover, it is required that $\sum_{\widehat{\pi}\in\Pi} \mathbf{P}(\widehat{\pi}) > 1-p$. Note that all paths that start with a prefix $s_0 s_1 \ldots s_n \in \Pi$ never visit a state for which Ψ holds. The cylinder sets $\mathit{Cyl}(\widehat{\pi})$ of all path fragments $\widehat{\pi} \in \Pi$ are contained in $\{\, \pi \in \mathit{Paths}(s) \mid \pi \models \Box\neg\Psi \,\}$. Hence:

$$\mathit{Pr}(s \models \Diamond\Psi) \;=\; 1 - \mathit{Pr}(s \models \Box\neg\Psi) \;\leqslant\; 1 - \sum_{\widehat{\pi}\in\Pi} \mathbf{P}(\widehat{\pi}) \;<\; 1-(1-p) \;=\; p.$$

Vice versa, $\mathit{Pr}(s \models \Box\neg\Psi)$ equals the probability of reaching a BSCC T via a path fragment through $\neg\Psi$-states such that $T \cap \mathit{Sat}(\Psi) = \varnothing$. Hence, $\mathit{Pr}(s \models \Diamond\Psi) < p$ if and only if there exists such a finite set Π. Such a set Π can be obtained by successively increasing n and collecting all finite path fragments $s_0\, s_1 \ldots s_k$ up to length $k \leqslant n$ that fulfill the above conditions (i.e., $s_0, \ldots, s_{k-1} \not\models \Psi$ and $s_k \in T$ for some BSCC T with $T \cap \mathit{Sat}(\Psi) = \varnothing$), until the probabilities $\mathbf{P}(s_0\, s_1 \ldots s_k)$ sum up to a value $> 1-p$.

10.2.2 The Qualitative Fragment of PCTL

The logic PCTL has been introduced as a variant of CTL where the path quantifiers $\exists$ and $\forall$ are replaced by the probabilistic operator $\mathbb{P}_J$. It is the purpose of this section to compare the expressiveness of CTL and PCTL in more detail. As PCTL provides the possibility to specify lower (or upper) bounds in the likelihood that they differ from zero and 1, as in $\mathbb{P}_{\geqslant\frac{1}{2}}(\varphi)$, it is evident that there exist properties that can be defined in PCTL but not in CTL. It remains to investigate how the *qualitative fragment* of PCTL—only allowing bounds with p=0 or p=1—relates to CTL. As we will see, these logics are not equally expressive, but are incomparable (though their common fragment is large).

Definition 10.42. Qualitative Fragment of PCTL

State formulae in the *qualitative fragment* of PCTL (over AP) are formed according to the following grammar:

$$\Phi ::= \text{true} \;\Big|\; a \;\Big|\; \Phi_1 \wedge \Phi_2 \;\Big|\; \neg\Phi \;\Big|\; \mathbb{P}_{>0}(\varphi) \;\Big|\; \mathbb{P}_{=1}(\varphi)$$

where $a \in AP$, φ is a path formula formed according to the following grammar:

$$\varphi ::= \bigcirc \Phi \;\Big|\; \Phi_1 \,\mathsf{U}\, \Phi_2$$

where Φ, Φ_1, and Φ_2 are state formulae. ■

The formulae of the qualitative fragment of PCTL are referred to as *qualitative* PCTL formulae. Although the only allowed probability bounds are > 0 and $=1$, the bounds $= 0$ and < 1 can be derived, as

$$\mathbb{P}_{=0}(\varphi) \;\equiv\; \neg\mathbb{P}_{>0}(\varphi) \quad \text{and} \quad \mathbb{P}_{<1}(\varphi) \;\equiv\; \neg\mathbb{P}_{=1}(\varphi).$$

Thus, e.g., $\mathbb{P}_{=1}(\Diamond\mathbb{P}_{>0}(\bigcirc a))$ and $\mathbb{P}_{<1}(\mathbb{P}_{>0}(\Diamond a)\,\mathsf{U}\, b)$ are qualitative PCTL formulae, while $\mathbb{P}_{<0.5}(\Diamond a)$ and $\mathbb{P}_{=1}(a\,\mathsf{U}^{\leqslant 5} b)$ are not. Note that the bounded until operator is not part of the qualitative fragment of PCTL.

In the sequel, let $\mathcal{M}$ be a Markov chain. PCTL formulae will be interpreted over $\mathcal{M}$. CTL formulae are interpreted over the induced transition system of $\mathcal{M}$, i.e., $TS(\mathcal{M})$ (see page 753). Recall that $s \xrightarrow{\tau} s'$ in this transition system if and only if $\mathbf{P}(s, s') > 0$; the exact transition probabilities are thus abstracted away in $TS(\mathcal{M})$.

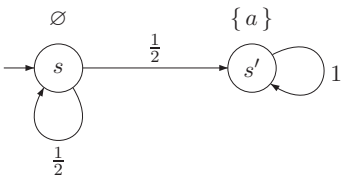
Definition 10.43. Equivalence of PCTL and CTL Formulae

The PCTL formula Φ is *equivalent* to the CTL formula Ψ, denoted $\Phi \equiv \Psi$, if $Sat_{\mathcal{M}}(\Phi) = Sat_{TS(\mathcal{M})}(\Psi)$ for each Markov chain $\mathcal{M}$. ■

Consider first the PCTL formula $\mathbb{P}_{=1}(\varphi)$ and the CTL formula $\forall\varphi$. The former asserts φ to hold almost surely. This allows for certain exceptional executions where φ is violated. The formula $\forall\varphi$, however, requires φ to hold for all paths. No exceptions are allowed. In fact, we have:

$$s \models \forall\Diamond a \quad \text{implies} \quad s \models \mathbb{P}_{=1}(\Diamond a).$$

The reverse direction, however, does not hold. For example, consider the following Markov chain:



Formula $\mathbb{P}_{=1}(\Diamond a)$ holds in state s as the probability of visiting state s infinitely often is zero. On the other hand, the path s^ω is possible and clearly violates $\Diamond a$. Thus:

$$s \models \mathbb{P}_{=1}(\Diamond a) \quad \text{but} \quad s \not\models \forall \Diamond a.$$

However, for certain path formulae φ, the probability bounds > 0 and $= 1$ correspond to existential and universal path quantification in CTL. The simplest such cases are path formulae involving the next-step operator:

$$\begin{aligned} s \models \mathbb{P}_{=1}(\bigcirc a) \quad &\text{iff} \quad s \models \forall \bigcirc a \\ s \models \mathbb{P}_{>0}(\bigcirc a) \quad &\text{iff} \quad s \models \exists \bigcirc a \end{aligned}$$

where s is a state in an arbitrary (possibly infinite) Markov chain. The same applies to reachability conditions $\Diamond a$ that hold with positive probability if and only if $\Diamond a$ holds for some path, and to invariants $\Box a$ that hold on all paths if and only if they hold almost surely:

$$\begin{aligned} s \models \mathbb{P}_{>0}(\Diamond a) \quad &\text{iff} \quad s \models \exists \Diamond a \\ s \models \mathbb{P}_{=1}(\Box a) \quad &\text{iff} \quad s \models \forall \Box a. \end{aligned}$$

Let us illustrate how to conduct the formal proof of these statements. Consider the first statement. Assume $s \models \mathbb{P}_{>0}(\Diamond a)$. By the PCTL semantics, it follows that $Pr(s \models \Diamond a) > 0$. Thus, $\{ \pi \in \mathit{Paths}(s) \mid \pi \models \Diamond a \} \neq \varnothing$, and hence, $s \models \exists \Diamond a$. Vice versa, assume $s \models \exists \Diamond a$, i.e., there exists a finite path fragment $s_0\, s_1 \ldots s_n$ with $s_0 = s$ and $s_n \models a$. It follows that all paths in the cylinder set $\mathit{Cyl}(s_0 s_1 \ldots s_n)$ fulfill $\Diamond a$. Thus:

$$Pr(s \models \Diamond a) \;\geqslant\; Pr_s(\mathit{Cyl}(s_0 s_1 \ldots s_n)) \;=\; \mathbf{P}(s_0 s_1 \ldots s_n) \;> 0.$$

Therefore, $s \models \mathbb{P}_{>0}(\Diamond a)$.

The second statement follows by duality. First, observe that

$$s \models \mathbb{P}_{=1}(\Box a) \;=\; \mathbb{P}_{=0}(\Diamond \neg a) \;\equiv\; \neg \mathbb{P}_{>0}(\Diamond \neg a).$$

We can now apply the equivalence of $\mathbb{P}_{>0}(\Diamond a)$ and $\exists \Diamond\, a$ as follows:

$$s \models \neg \mathbb{P}_{>0}(\Diamond \neg a) \text{ iff } s \not\models \mathbb{P}_{>0}(\Diamond \neg a) \text{ iff } s \not\models \exists \Diamond \neg a \text{ iff } s \models \underbrace{\neg \exists \Diamond \neg a}_{= \forall \Box a}.$$

Thus, the equivalence of $\mathbb{P}_{>0}(\bigcirc\, \mathbb{P}_{>0}(\Diamond \Phi))$ and $\mathbb{P}_{>0}(\Diamond\, \mathbb{P}_{>0}(\bigcirc\, \Phi))$ (stated on page 784) is the probabilistic analogue of the CTL duality law $\exists \bigcirc \exists \Diamond\, \Phi \;\equiv\; \exists \Diamond\, \exists \bigcirc\, \Phi$.

The arguments for the equivalence of $\mathbb{P}_{>0}(\Diamond a)$ and $\exists \Diamond a$ can be generalized toward constrained reachability:

$$s \models \mathbb{P}_{>0}(a \,\mathsf{U}\, b) \quad \text{iff} \quad s \models \exists (a \,\mathsf{U}\, b).$$

A generalization that, however, does not work is swapping the probability bound > 0 and $=1$ in the above two equivalences, as $\mathbb{P}_{=1}(\Diamond a) \not\equiv \forall\Diamond a$ and $\mathbb{P}_{>0}(\Box a) \not\equiv \exists\Box a$. In fact, the PCTL formula $\mathbb{P}_{=1}(\Diamond a)$ cannot be expressed in CTL.

Lemma 10.44.

1. *There is no CTL formula that is equivalent to* $\mathbb{P}_{=1}(\Diamond a)$.
2. *There is no CTL formula that is equivalent to* $\mathbb{P}_{>0}(\Box a)$.

Proof: We provide the proof of the first statement; the second statement follows by duality, i.e., $\mathbb{P}_{=1}(\Diamond a) = \neg\mathbb{P}_{>0}(\Box\neg a)$. The proof is by contraposition. Assume there exists a CTL formula Φ such that $\Phi \equiv \mathbb{P}_{=1}(\Diamond a)$. Consider the infinite Markov chain $\mathcal{M}_p$ modeling the random walk; see Figure 10.6 (page 780). The state space of $\mathcal{M}_p$ is $S = \mathbb{N}$ and there are transitions from state n to state $n-1$ with probability $1-p$ and to state $n+1$ with probability p for $n \geqslant 1$, while $\mathbf{P}(0,0) = 1-p$ and $\mathbf{P}(0,1) = p$. Let the labeling function L be such that a only holds in state 0. That is, $0 \models a$ and $n \not\models a$ for all $n > 0$. Recall that for $p < \frac{1}{2}$, it holds that for any state n almost surely state 0 will be visited, while for $p > \frac{1}{2}$, the Markov chain $\mathcal{M}_p$ drifts to the right and the probability of reaching state 0 from any other state $n > 0$ is strictly smaller than 1. Thus, e.g., in $\mathcal{M}_{\frac{1}{4}}$ the formula $\mathbb{P}_{=1}(\Diamond a)$ holds for all states, while in $\mathcal{M}_{\frac{3}{4}}$, the formula $\mathbb{P}_{=1}(\Diamond a)$ does not hold in, e.g., state $n = 1$. Hence:

$$1 \in \mathit{Sat}_{\mathcal{M}_{\frac{1}{4}}}(\,\mathbb{P}_{=1}(\Diamond a)\,) \quad \text{but} \quad 1 \notin \mathit{Sat}_{\mathcal{M}_{\frac{3}{4}}}(\,\mathbb{P}_{=1}(\Diamond a)\,).$$

Since $\mathit{TS}(\mathcal{M}_{\frac{1}{4}}) = \mathit{TS}(\mathcal{M}_{\frac{3}{4}})$, it follows that

$$\mathit{Sat}_{\mathcal{M}_{\frac{1}{4}}}(\Phi) \;=\; \mathit{Sat}_{\mathcal{M}_{\frac{3}{4}}}(\Phi).$$

Hence, state 1 either fulfills the CTL formula Φ in both structures or in none of them. This, however, contradicts the assumption that $\mathbb{P}_{=1}(\Diamond a)$ and Φ are equivalent. ■

The proof relies on the fact that the satisfaction of $\mathbb{P}_{=1}(\Diamond a)$ for infinite Markov chains may depend on the precise value of the transition probabilities, while CTL just refers to the underlying graph of a Markov chain. In case one is restricted to finite Markov chains, the statement in Lemma 10.44 no longer holds. For each finite Markov chain $\mathcal{M}$ and state s of $\mathcal{M}$ it holds that

$$s \models \mathbb{P}_{=1}(\Diamond a) \quad \text{iff} \quad s \models \forall(\,(\exists\Diamond a)\,\mathsf{W}\,a\,)$$

where W is the weak until operator (see Remark 6.9 on page 327) defined by $\Phi \,\mathsf{W}\, \Psi = \Phi \,\mathsf{U}\, \Psi \vee \Box\, \Phi$. The proof of (a slight generalization of) this statement is the subject of Exercise 10.11 (page 903). For finite Markov chains, the qualitative fragment of PCTL can be embedded into CTL.

Whereas the previous lemma stated that some qualitative PCTL cannot be expressed in CTL, the following result states that for some CTL formulae no equivalent qualitative PCTL exists. As a result, CTL and the qualitative fragment of PCTL are incomparable.

Lemma 10.45.

1. *There is no qualitative PCTL formula that is equivalent to* $\forall \Diamond a$.
2. *There is no qualitative PCTL formula that is equivalent to* $\exists \Box a$.

Proof: We provide the proof of the first claim; the second claim follows by duality, since $\forall \Diamond a = \neg \exists \Box \neg a$. For $n \geqslant 1$, let the MCs $\mathcal{M}_n$ and $\mathcal{M}'_n$ be defined as follows. The state spaces of $\mathcal{M}_n$ and $\mathcal{M}'_n$ are $S_n = \{ t_0, t_1, \ldots, t_{n-1} \} \cup \{ s_n \}$ and $S'_n = \{ t_0, t_1, \ldots, t_n \}$, respectively. $\mathcal{M}_n$ and $\mathcal{M}'_n$ agree on the fragment consisting of their common states, i.e., state t_0 through t_{n-1}. State s_n in $\mathcal{M}_n$ has a self-loop and a transition to t_{n-1}, both with probability $\frac{1}{2}$. In contrast, state t_n in $\mathcal{M}'_n$ only has a transition $t_n \to t_{n-1}$ with probability 1. Both Markov chains are depicted in Figure 10.8. The transition probabilities in $\mathcal{M}'_n$ are given by:

$$\mathbf{P}'_n(t_i, t_{i-1}) = 1 \text{ for } 1 \leqslant i \leqslant n \quad \text{and} \quad \mathbf{P}'_n(t_0, t_0) = 1$$

and $\mathbf{P}'_n(\cdot) = 0$ otherwise. The transition probabilities in $\mathcal{M}_n$ are given by

$$\begin{aligned} &\mathbf{P}_n(s_n, t_{n-1}) = \mathbf{P}_n(s_n, s_n) = \tfrac{1}{2}, \\ &\mathbf{P}_n(t_i, t_{i-1}) = 1 \quad \text{for } 1 \leqslant i < n, \\ &\mathbf{P}_n(t_0, t_0) = 1 \end{aligned}$$

and $\mathbf{P}_n(\cdot) = 0$ in the remaining cases. Let $AP = \{ a \}$ and labeling functions L_n for $\mathcal{M}_n$ and L'_n for $\mathcal{M}'_n$ given by $L_n(s_n) = L_n(t_i) = L'_n(t_j) = \varnothing$ for $1 \leqslant i < n$ and $1 \leqslant j \leqslant n$ and $L_n(t_0) = L'_n(t_0) = \{ a \}$.

It can now be shown that for any qualitative PCTL formula Φ of *nesting depth* smaller than n:

$$s_n \models \Phi \quad \text{iff} \quad t_n \models \Phi.$$

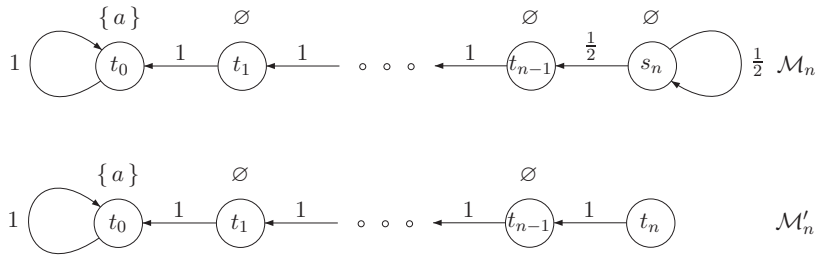


Figure 10.8: The Markov chains $\mathcal{M}_n$ and $\mathcal{M}'_n$.

This can be proven by induction on the nesting depth of Φ. (The proof is omitted here and left as an exercise.) The nesting depth of qualitative PCTL formulae is defined by

$$\begin{aligned}
nd(\text{true}) &= 0 \\
nd(a) &= 0 \\
nd(\Phi \wedge \Psi) &= \max\{nd(\Phi), nd(\Psi)\} \\
nd(\neg\Phi) &= nd(\Phi) \\
nd(\mathbb{P}_J(\varphi)) &= nd(\varphi) + 1 \\
nd(\bigcirc \Phi) &= nd(\Phi) \\
nd(\Phi \,\mathsf{U}\, \Psi) &= \max\{nd(\Phi), nd(\Psi)\}
\end{aligned}$$

Suppose now that there exists a qualitative PCTL formula Φ such that $\Phi \equiv \forall\Diamond a$. Let $n = nd(\Phi) + 1$. Since $s_n \models \Phi$ iff $t_n \models \Phi$, states s_n and t_n either both fulfill Φ or none of them. On the other hand, $s_n \not\models \forall\Diamond a$ (because $s_n\, s_n\, s_n \ldots \not\models \Diamond a$), while $t_n \models \forall\Diamond a$. Contradiction. ∎

These results indicate that extending the syntax of PCTL by adding universal and existential path quantification increases its expressiveness. An essential aspect in the proof of Lemma 10.45 is the existence of a state s, say, that has a self-loop and (at least) one outgoing transition to another state. (In the proof, the state of interest is s_n.) As the probability of taking this loop infinitely often is zero, the validity of a qualitative PCTL formula in s in unaffected by the self-loop. However, the (zero probable) path s^ω may be decisive for the validity of CTL formulae. The path s^ω, however, can be considered as unfair, as there are infinitely many opportunities to take the other outgoing transition. The existence of such unfair computations turns out to be vital for the validity of Lemma 10.45. In fact, under appropriate fairness constraints, the equivalence of $\mathbb{P}_{=1}(\Diamond a)$ and $\forall\Diamond a$ can be established. This can be seen as follows. Assume $\mathcal{M}$ is a finite Markov chain and that any state s in $\mathcal{M}$ is uniquely characterized by an atomic proposition, say s. We

regard the strong fairness constraint *sfair* defined by

$$\mathit{sfair} \;=\; \bigwedge_{s\in S} \; \bigwedge_{t \,\in\, \mathit{Post}(s)} (\Box\Diamond s \rightarrow \Box\Diamond t).$$

It asserts that when a state s is visited infinitely often, then any of its direct successors is visited infinitely often too. Using Theorem 10.25 (page 772), we obtain (see Exercise 10.8 on page 902)

$$\begin{aligned} s \models \mathbb{P}_{=1}(a \,\mathsf{U}\, b) \quad &\text{iff} \quad s \models_{\mathit{sfair}} \forall(a \,\mathsf{U}\, b) \;\text{ and} \\ s \models \mathbb{P}_{>0}(\Box a) \quad &\text{iff} \quad s \models_{\mathit{sfair}} \exists\Box a. \end{aligned}$$

As *sfair* is a realizable fairness constraint—from every reachable state at least one fair path starts—we obtain

$$\begin{array}{lllll} s \models_{\mathit{sfair}} \exists(a \,\mathsf{U}\, b) & \text{iff} & s \models \exists(a \,\mathsf{U}\, b) & \text{iff} & s \models \mathbb{P}_{>0}(a \,\mathsf{U}\, b) \\ s \models_{\mathit{sfair}} \forall\bigcirc\, a & \text{iff} & s \models \forall\bigcirc\, a & \text{iff} & s \models \mathbb{P}_{=1}(\bigcirc\, a) \\ s \models_{\mathit{sfair}} \exists\bigcirc\, a & \text{iff} & s \models \exists\bigcirc\, a & \text{iff} & s \models \mathbb{P}_{>0}(\bigcirc\, a). \end{array}$$

Thus, for finite Markov chains the qualitative fragment of PCTL can be viewed as a variant of CTL with some kind of strong fairness. A similar result holds for infinite Markov chains, but then the outermost conjunction in *sfair* has an infinite range. The same applies to the innermost conjunction in case infinite branching occurs.

Repeated Reachability and Persistence Properties. We now show that two properties that cannot be expressed in CTL (but which can be in CTL*) can be expressed in the qualitative fragment of PCTL.

For CTL, universal repeated reachability properties can be formalized by the combination of the modalities $\forall\Box$ and $\forall\Diamond$:

$$s \models \forall\Box\forall\Diamond a \quad \text{iff} \quad \pi \models \Box\Diamond a \text{ for all } \pi \in \mathit{Paths}(s).$$

More details can be found in Remark 6.8 on page 326. For finite Markov chains, a similar result holds for (the qualitative fragment of) PCTL.

Lemma 10.46. Almost Sure Repeated Reachability is PCTL-definable

Let $\mathcal{M}$ be a finite Markov chain and s a state of $\mathcal{M}$. Then:

$$s \models \mathbb{P}_{=1}(\,\Box\,\mathbb{P}_{=1}(\Diamond a)\,) \quad \textit{iff} \quad \mathit{Pr}_s\{\,\pi \in \mathit{Paths}(s) \mid \pi \models \Box\Diamond a\,\} \;=\; 1.$$

Proof: "$\Longrightarrow$": Assume $s \models \mathbb{P}_{=1}(\,\Box\,\mathbb{P}_{=1}(\Diamond a)\,)$. As $\mathbb{P}_{=1}(\Box a)$ is equivalent to $\forall\Box a$, we have $s \models \forall\Box\, Sat(\mathbb{P}_{=1}(\Diamond a))$. Hence, all states t that are reachable from s satisfy $\mathbb{P}_{=1}(\Diamond a))$. Consider a BSCC $T \subseteq S$ of $\mathcal{M}$ such that $Pr(s \models \Diamond T) > 0$. Then, each state t in T is reachable from s and $t \models \mathbb{P}_{=1}(\Diamond a)$. As $Post^*(t) = T$, $T \cap Sat(a) \neq \varnothing$. Hence, almost all paths $\pi \in Paths(s)$ that fulfill $\Diamond T$ satisfy $\Box\Diamond a$. Theorem 10.27 (page 775) then yields the claim.

"$\Longleftarrow$": Assume $Pr_s\{\,\pi \in Paths(s) \mid \pi \models \Box\Diamond a\,\} = 1$. Theorem 10.27 (page 775) yields

$$\sum_{\substack{T \,\in\, BSCC(\mathcal{M}) \\ T \,\cap\, Sat(a) \,\neq\, \varnothing}} Pr_s\{\pi \in Paths(s) \mid \pi \models \Diamond T\} \;=\; 1.$$

On the other hand, $Pr(t \models \Diamond a) = 1$ for each state t of a BSCC T in $\mathcal{M}$ with $T \cap Sat(a) \neq \varnothing$. Therefore, the union of all BSCCs T with $T \cap Sat(a) \neq \varnothing$ is a subset of $Sat(\mathbb{P}_{=1}(\Diamond a))$. This yields

$$Pr(\,s \models \Diamond Sat(\mathbb{P}_{=1}(\Diamond a))\,) \;=\; 1$$

and $s \models \mathbb{P}_{=1}(\,\Box\,\mathbb{P}_{=1}(\Diamond a)\,)$. ∎

According to this result, the PCTL formula $\mathbb{P}_{=1}(\,\Box\,\mathbb{P}_{=1}(\Diamond a)\,)$ denotes that almost surely a state for which a holds is repeatedly reachable. In the sequel, let $\mathbb{P}_{=1}(\Box\Diamond a)$ abbreviate this formula.

There is no CTL formula that is equivalent to the CTL* formula $\exists\Box\Diamond a$. This is different for PCTL, as the requirement that the probability for the event $\Box\Diamond a$ is positive can be described in PCTL. For finite Markov chains, even arbitrary rational lower or upper bounds for the likelihood of $\Box\Diamond a$ are PCTL-definable. In fact, the argument given in the proof of Lemma 10.46 can be generalized for arbitrary probability bounds. The crucial point is that almost surely a BSCC $T \in \mathcal{M}$ will be reached and each of its states will be visited infinitely often. Thus, the probabilities for $\Box\Diamond a$ agree with the probability of reaching a BSCC T where a holds for some state in T. Hence, for finite Markov chain $\mathcal{M}$ and probability interval J, the PCTL formula $\mathbb{P}_J(\Box\Diamond a)$ defined by

$$\mathbb{P}_J(\Box\Diamond a) \;=\; \mathbb{P}_J(\,\Diamond\, \underbrace{\mathbb{P}_{=1}(\Box\Diamond a)}_{=\mathbb{P}_{=1}(\Box\mathbb{P}_{=1}(\Diamond a))}\,)$$

characterizes all states s in $\mathcal{M}$ for which $Pr(s \models \Box\Diamond a) \in J$. This is expressed by the following theorem:

Theorem 10.47. Repeated Reachability Probabilities Are PCTL-definable

Let $\mathcal{M}$ be a finite Markov chain, s a state of $\mathcal{M}$ and $J \subseteq [0,1]$ an interval. Then:

$$s \models \underbrace{\mathbb{P}_J(\Diamond \mathbb{P}_{=1}(\Box \mathbb{P}_{=1}(\Diamond a)))}_{=\mathbb{P}_J(\Box\Diamond a)} \quad \textit{iff} \quad Pr(s \models \Box\Diamond a) \in J.$$

Proof: As $\mathcal{M}$ is finite, Theorem 10.27 (page 775) permits computing the probability for visiting an a-state infinitely often, as follows:

$$\begin{aligned}
& Pr(s \models \Box\Diamond a) \\
= \; & Pr_s\{\, \pi \in Paths(s) \mid \pi \models \Box\Diamond a \,\} \\
= \; & Pr_s\{\, \pi \in Paths(s) \mid \inf(\pi) \in BSCC(\mathcal{M}) \wedge \inf(\pi) \cap Sat(a) \neq \varnothing \,\} \\
= \; & Pr_s\{\, \pi \in Paths(s) \mid \pi \models \Diamond T \text{ for some } T \in BSCC(\mathcal{M}) \text{ with } T \cap Sat(a) \neq \varnothing \,\} \\
= \; & Pr(\, s \models \Diamond Sat(\mathbb{P}_{=1}(\Box\Diamond a)) \,).
\end{aligned}$$

Hence, $Pr(s \models \Box\Diamond a) \in J$ if and only if $Pr(s \models \Diamond Sat(\mathbb{P}_{=1}(\Box\Diamond a))) \in J$ if and only if $s \models \mathbb{P}_J(\Diamond \mathbb{P}_{=1}(\Box\Diamond a)) = \mathbb{P}_J(\Box\Diamond a)$. ■

Recall that universal persistence properties cannot be expressed in CTL; see Lemma 6.19 (page 335). For finite Markov chains, PCTL allows specifying almost sure persistence and, moreover, persistence properties with arbitrary lower or upper bounds on the probability. This is stated by the following theorem.

Theorem 10.48. Persistence Probabilities are PCTL-Definable

For finite Markov chain $\mathcal{M}$, state s of $\mathcal{M}$ and interval $J \subseteq [0,1]$:

$$s \models \mathbb{P}_J(\Diamond \mathbb{P}_{=1}(\Box a)) \quad \textit{iff} \quad Pr(s \models \Diamond\Box a) \in J.$$

Proof: This result follows by Theorem 10.27 (page 775) and the observation that for each BSCC T of $\mathcal{M}$: (i) if $T \subseteq Sat(a)$, then $Pr(t \models \Box a) = 1$ for all states $t \in T$, and (ii) if $T \setminus Sat(a) \neq \varnothing$, then $Pr(t \models \Diamond\Box a) = 0$ for all states $t \in T$. ■

In particular, the requirement that a persistence property holds almost surely for a certain state s (i.e., $Pr(s \models \Diamond\Box a) = 1$) is given by the PCTL formula $\mathbb{P}_{=1}(\Diamond \mathbb{P}_{=1}(\Box a))$.

Let $\mathbb{P}_J(\Diamond\Box\Phi)$ denote the PCTL formula to express a probability bound on persistence properties.

10.3 Linear-Time Properties

The previous section has introduced the branching-time temporal logic PCTL, and has presented a model-checking algorithm for this logic for finite Markov chains. This section deals with linear-time properties; see Chapter 3. Recall that an LT property is a set of infinite traces. The quantitative model-checking problem that we are confronted with is: given a finite Markov chain $\mathcal{M}$ and an ω-regular property P, compute the probability for the set of paths in $\mathcal{M}$ for which P holds. Some special cases, like properties of the form $C \,\mathsf{U}\, B$ or $\Box\Diamond B$, where B and C are sets of states in $\mathcal{M}$, have been discussed before. The purpose of this section is to generalize these results toward arbitrary ω-regular properties.

Definition 10.49. Probability of LT Properties

Let $\mathcal{M}$ be a Markov chain and P an ω-regular property (both over AP). The probability for $\mathcal{M}$ to exhibit a trace in P, denoted $Pr^{\mathcal{M}}(P)$, is defined by

$$Pr^{\mathcal{M}}(P) \;=\; Pr^{\mathcal{M}}\{\, \pi \in \mathit{Paths}(\mathcal{M}) \mid \mathit{trace}(\pi) \in P \,\}.$$

■

This definition, of course, requires the measurability of the set of paths π with $\mathit{trace}(\pi) \in P$; see Remark 10.57 on page 804. For state s in $\mathcal{M}$ let $Pr^{\mathcal{M}}(s \models P)$, or briefly, $Pr(s \models P)$, for $Pr^{\mathcal{M}_s}(P)$, i.e.,

$$Pr(s \models P) \;=\; Pr_s\{\, \pi \in \mathit{Paths}(s) \mid \mathit{trace}(\pi) \in P \,\}.$$

Similarly, for the LTL formula φ we write $Pr^{\mathcal{M}}(\varphi)$ for $Pr^{\mathcal{M}}(\mathit{Words}(\varphi)) \;=\; Pr^{\mathcal{M}}\{\pi \in \mathit{Paths}(\mathcal{M}) \mid \pi \models \varphi\}$, where $\models$ is the standard LTL satisfaction relation, i.e., $\pi \models \varphi$ iff $\mathit{trace}(\pi) \in \mathit{Words}(\varphi)$. For state s of $\mathcal{M}$, $Pr^{\mathcal{M}}(s \models \varphi)$ denotes $Pr^{\mathcal{M}_s}(\varphi)$, i.e.,

$$Pr(s \models \varphi) \;=\; Pr_s\{\, \pi \in \mathit{Paths}(s) \mid \pi \models \varphi \,\}.$$

Given an ω-regular property P over AP and finite Markov chain $\mathcal{M} = (S, \mathbf{P}, \iota_{\mathrm{init}}, AP, L)$, the goal is to compute $Pr^{\mathcal{M}}(P)$. As for verifying ω-regular properties, we adopt an automata-based approach. The main steps of this approach are as follows. The (complement of the) LT property P is represented by means of an automaton $\mathcal{A}$, say. For

regular safety properties this is an automaton for the bad prefixes, whereas for ω-regular properties this is a Büchi automaton for the complement of P. It then suffices to check a reachability and persistence property, respectively, on the product $\mathcal{M} \otimes \mathcal{A}$.

In order to guarantee that $\mathcal{M} \otimes \mathcal{A}$ is a Markov chain, however, the automaton $\mathcal{A}$ needs to be deterministic. This is a main difference with the traditional setting of transition systems where nondeterministic (finite-state or Büchi) automata do suffice. For regular safety properties we therefore assume $\mathcal{A}$ to be a DFA for the bad prefixes. For the ω-regular properties, $\mathcal{A}$ is assumed to be a deterministic Rabin automaton (DRA, for short). DRAs are equally expressive as ω-regular languages. (Recall that deterministic Büchi automata are strictly less expressive than NBAs and therefore do not suffice for our purpose).

Rather than checking a reachability or persistence property, determining $Pr^{\mathcal{M}}(P)$ is reduced to computing the probability of accepting runs in the product Markov chain $\mathcal{M} \otimes \mathcal{A}$. This is first discussed in detail for regular safety properties, and subsequently for ω-regular properties.

Regular Safety Properties We first consider regular safety properties. Recall that a safety property is regular whenever all bad prefixes constitute a regular language. Let $\mathcal{A} = (Q, 2^{AP}, \delta, q_0, F)$ be a deterministic finite automaton (DFA) for the bad prefixes of a regular safety property P_{safe}. That is,

$$P_{safe} \;=\; \{\, A_0\, A_1\, A_2 \ldots \in (2^{AP})^{\omega} \mid \forall n \geqslant 0.\, A_0\, A_1 \ldots A_n \notin \mathcal{L}(\mathcal{A}) \,\}.$$

Without loss of generality, the transition function δ is assumed to be total, i.e., $\delta(q, A)$ is defined for each $A \subseteq AP$ and each state $q \in Q$. Furthermore, let $\mathcal{M} = (S, \mathbf{P}, \iota_{\text{init}}, AP, L)$ be a finite Markov chain. Our interest is to compute the probability

$$Pr^{\mathcal{M}}(P_{safe}) \;=\; 1 \;-\; \sum_{s \in S} \iota_{\text{init}}(s) \cdot Pr(s \models \mathcal{A})$$

for $\mathcal{M}$ to generate a trace in P_{safe}, i.e., a trace such that none of its prefixes is accepted by $\mathcal{A}$. The probability $Pr(s \models \mathcal{A})$ is given by

$$\begin{aligned} Pr(s \models \mathcal{A}) \;&=\; Pr_s^{\mathcal{M}}\{\, \pi \in Paths(s) \mid \mathit{pref}(\mathit{trace}(\pi)) \cap \mathcal{L}(\mathcal{A}) \neq \varnothing \,\} \\ &=\; Pr_s^{\mathcal{M}}\{\, \pi \in Paths(s) \mid \mathit{trace}(\pi) \notin P_{safe} \,\} \end{aligned}$$

where $\mathit{pref}(A_0\, A_1\, \ldots)$ denotes the set of all finite prefixes of the infinite word $A_0\, A_1\, \ldots \in (2^{AP})^{\omega}$. The value $Pr(s \models \mathcal{A})$ can be written as the (possibly infinite) sum:

$$Pr(s \models \mathcal{A}) \;=\; \sum_{\widehat{\pi}} \mathbf{P}(\widehat{\pi})$$

where $\widehat{\pi}$ ranges over all finite path fragments $s_0\, s_1 \ldots s_n$ starting in $s_0 = s$ such that (1) $\mathit{trace}(s_0\, s_1 \ldots s_n) = L(s_0)\, L(s_1) \ldots L(s_n) \in \mathcal{L}(\mathcal{A})$, and (2) the length n of $\widehat{\pi}$ is minimal according to (1), i.e., $\mathit{trace}(s_0\, s_1 \ldots s_i) \notin \mathcal{L}(\mathcal{A})$ for all $0 \leqslant i < n$. Condition (2) is equivalent to the requirement that $\mathit{trace}(\widehat{\pi})$ is a minimal bad prefix of P_{safe}.

Computing the values $Pr(s \models \mathcal{A})$ by using these sums may be difficult. Instead, we adapt the techniques for checking regular safety properties of transition systems to the probabilistic case. This involves the product of $\mathcal{M}$ and $\mathcal{A}$ which is defined as follows.

Definition 10.50. Product Markov Chain

Let $\mathcal{M} = (S, \mathbf{P}, \iota_{\mathrm{init}}, AP, L)$ be a Markov chain and $\mathcal{A} = (Q, 2^{AP}, \delta, q_0, F)$ be a DFA. The product $\mathcal{M} \otimes \mathcal{A}$ is the Markov chain:

$$\mathcal{M} \otimes \mathcal{A} \;=\; (S \times Q, \mathbf{P}', \iota'_{\mathrm{init}}, \{\, \mathit{accept} \,\}, L')$$

where $L'(\langle s, q\rangle) = \{\, \mathit{accept} \,\}$ if $q \in F$ and $L'(\langle s, q\rangle) = \varnothing$ otherwise, and

$$\iota'_{\mathrm{init}}(\langle s, q\rangle) \;=\; \begin{cases} \iota_{\mathrm{init}}(s) & \text{if } q = \delta(q_0, L(s)) \\ 0 & \text{otherwise.} \end{cases}$$

The transition probabilities in $\mathcal{M} \otimes \mathcal{A}$ are given by

$$\mathbf{P}'(\langle s, q\rangle, \langle s', q'\rangle) \;=\; \begin{cases} \mathbf{P}(s, s') & \text{if } q' = \delta(q, L(s')) \\ 0 & \text{otherwise.} \end{cases}$$

■

Since $\mathcal{A}$ is deterministic, $\mathcal{M} \otimes \mathcal{A}$ can be viewed as the unfolding of $\mathcal{M}$ where the automaton component q of the states $\langle s, q\rangle$ in $\mathcal{M} \otimes \mathcal{A}$ records the current state in $\mathcal{A}$ for the path fragment taken so far. More precisely, for each (finite or infinite) path fragment $\pi = s_0\, s_1\, s_2 \ldots$ in $\mathcal{M}$ there exists a unique run $q_0\, q_1\, q_2 \ldots$ in $\mathcal{A}$ for $\mathit{trace}(\pi) = L(s_0)\, L(s_1)\, L(s_2) \ldots$ and

$$\pi^+ \;=\; \langle s_0, q_1\rangle\, \langle s_1, q_2\rangle\, \langle s_2, q_3\rangle \ldots$$

is a path fragment in $\mathcal{M} \otimes \mathcal{A}$. Vice versa, every path fragment in $\mathcal{M} \otimes \mathcal{A}$ which starts in state $\langle s, \delta(q_0, L(s))\rangle$ arises from the combination of a path fragment in $\mathcal{M}$ and a corresponding run in $\mathcal{A}$. Note that the DFA $\mathcal{A}$ does not affect the probabilities. That is, for each measurable set Π of paths in $\mathcal{M}$ and state s,

$$Pr_s^{\mathcal{M}}(\Pi) \;=\; Pr^{\mathcal{M} \otimes \mathcal{A}}_{\langle s, \delta(q_0, L(s))\rangle} \underbrace{\{\, \pi^+ \mid \pi \in \Pi \,\}}_{\Pi^+}$$

where the superscripts $\mathcal{M}$ and $\mathcal{M} \otimes \mathcal{A}$ are used to indicate the underlying Markov chain. In particular, if Π is the set of paths that start in s and refute P_{safe}, i.e.,

$$\Pi \;=\; \{\, \pi \in \mathit{Paths}^{\mathcal{M}}(s) \;\mid\; \mathit{pref}(\mathit{trace}(\pi)) \cap \mathcal{L}(\mathcal{A}) \neq \varnothing \,\},$$

the set Π^+ is the set of paths in $\mathcal{M} \otimes \mathcal{A}$ that start in $\langle s, \delta(q_0, L(s))\rangle$ and eventually reach an accept state of $\mathcal{A}$:

$$\Pi^+ \;=\; \{\, \pi^+ \in \mathit{Paths}^{\mathcal{M}\otimes\mathcal{A}}(\langle s, \delta(q_0, L(s))\rangle) \;\mid\; \pi^+ \models \Diamond \mathit{accept} \,\}.$$

Recall that the atomic proposition *accept* characterizes the set of states $\langle s, q\rangle$ where q is an accept state of $\mathcal{A}$. Thus, the paths satisfying $\Diamond \mathit{accept}$ agree with the event $\Diamond B$ where $B = S \times F$. This shows that $Pr^{\mathcal{M}}(s \models P_{safe})$ can be derived from the probability for the event $\Diamond \mathit{accept}$ in $\mathcal{M} \otimes \mathcal{A}$. This is formally stated in the following theorem.

Theorem 10.51. Quantitative Analysis for Safety Properties

Let P_{safe} be a regular safety property, $\mathcal{A}$ a DFA for the set of bad prefixes of P_{safe}, $\mathcal{M}$ a Markov chain, and s a state in $\mathcal{M}$. Then:

$$\begin{aligned} Pr^{\mathcal{M}}(s \models P_{safe}) &= Pr^{\mathcal{M}\otimes\mathcal{A}}(\langle s, q_s\rangle \not\models \Diamond \mathit{accept}) \\ &= 1 - Pr^{\mathcal{M}\otimes\mathcal{A}}(\langle s, q_s\rangle \models \Diamond \mathit{accept}) \end{aligned}$$

where $q_s = \delta(q_0, L(s))$.

Thus, the computation of the probability for a regular safety property in a Markov chain is reducible to computing the reachability probability in a product Markov chain. For finite Markov chains, the latter problem can be solved by means of the techniques discussed in Section 10.1.1 (see page 759ff).

For the special case of qualitative regular safety properties, i.e., whether P_{safe} almost surely holds in (finite) $\mathcal{M}$, a graph analysis suffices. By means of a DFS- or BFS-based search algorithm we can check whether a state $\langle s, q\rangle$ with $q \in F$ is reachable in $\mathcal{M} \otimes \mathcal{A}$. That is, the CTL formula $\exists\Diamond \mathit{accept}$ is checked. For the dual qualitative constraint, i.e., whether P_{safe} holds for $\mathcal{M}$ with probability zero, the graph-based techniques suggested in Corollary 10.29 (page 775) can be exploited to check whether $\Diamond \mathit{accept}$ holds almost surely in $\mathcal{M} \otimes \mathcal{A}$.

ω-Regular Properties Let us now consider the wider class of LT properties, i.e., ω-regular properties. Recall that P is ω-regular whenever P defines an ω-regular language.

Let P be an ω-regular property. In case the (complement of) P can be described by a *deterministic* Büchi automaton $\mathcal{A}$, say, the technique for regular safety properties can

be roughly adopted. Consider the product Markov chain $\mathcal{M} \otimes \mathcal{A}$ (see Definition 10.50 on page 798). It can now be shown, using similar arguments as for regular safety properties, that the probability of the event $\Box\Diamond accept$ in the product MC $\mathcal{M} \otimes \mathcal{A}$ coincides with the probability of refuting P by $\mathcal{M}$, i.e.,

$$Pr^{\mathcal{M}}(s \models \mathcal{A}) \;=\; Pr_s^{\mathcal{M}}\{\, \pi \in Paths(s) \mid trace(\pi) \in \mathcal{L}_\omega(\mathcal{A}) \,\}.$$

The probability of $\Box\Diamond accept$ can be obtained in polynomial time in the following way. First, determine the BSCCs of $\mathcal{M} \otimes \mathcal{A}$ (by a standard graph analysis). For each BSCC B that contains a state $\langle s, q \rangle$ with $q \in F$, determine the probability of eventually reaching B. This goes as indicated in Corollary 10.34 on page 779. Thus:

Theorem 10.52. Quantitative Analysis for DBA-Definable Properties

Let $\mathcal{A}$ be a DBA and $\mathcal{M}$ a Markov chain. Then, for all states s in $\mathcal{M}$:

$$Pr^{\mathcal{M}}(s \models \mathcal{A}) \;=\; Pr^{\mathcal{M} \otimes \mathcal{A}}(\langle s, q_s \rangle \models \Box\Diamond accept)$$

where $q_s = \delta(q_0, L(s))$.

Proof: The argument is similar as for regular safety properties. The connection between a path π in $\mathcal{M}$ and the corresponding path π^+ in $\mathcal{M} \otimes \mathcal{A}$ (which arises by augmenting the states s_i in $\pi = s_0\, s_1\, s_2 \ldots$ with the automaton states of the unique run for $trace(\pi)$ in $\mathcal{A}$) is as follows:

$$trace(\pi) \in \mathcal{L}_\omega(\mathcal{A}) \quad \text{iff} \quad \pi^+ \models \Box\Diamond accept.$$

Since $\mathcal{A}$ is deterministic, $\mathcal{A}$ does not affect the probabilities in $\mathcal{M} \otimes \mathcal{A}$. Thus, the probability for path π in $\mathcal{M}$ with $trace(\pi) \in \mathcal{L}_\omega(\mathcal{A})$ agrees with the probability of generating a path π^+ in $\mathcal{M} \otimes \mathcal{A}$ which arises by the lifting of path π in $\mathcal{M}$ where $trace(\pi) \in \mathcal{L}_\omega(\mathcal{A})$. The latter agrees with the probability for the paths π^+ in $\mathcal{M} \otimes \mathcal{A}$ with $\pi^+ \models \Box\Diamond accept$. ■

Since DBAs do not have the full power of ω-regular languages (see Section 4.3.3 on page 188ff), this approach is not capable of handling arbitrary ω-regular properties. To overcome this deficiency, Büchi automata will be replaced by an alternative automaton model for which their deterministic counterparts are as expressive as ω-regular languages. Such automata have the same components as NBAs (finite set of states, and so on) except for the acceptance condition. We consider *Rabin automata*. The acceptance condition of a Rabin automaton is given by a set of pairs of states:

$$\{\, (L_i, K_i) \mid 0 < i \leqslant k \,\} \quad \text{with} \quad L_i, K_i \subseteq Q.$$

A run of a Rabin automaton is accepting if for some pair (L_i, K_i) the states in L_i are visited finitely often and the states in K_i infinitely often. That is, an accepted run should

satisfy the following LTL formula:

$$\bigvee_{1 \leqslant i \leqslant k} (\Diamond\Box\neg L_i \wedge \Box\Diamond K_i).$$

A deterministic Rabin automaton is deterministic in the usual sense: it has a single initial state, and for each state and each input symbol, there is at most one successor state.

Definition 10.53. Deterministic Rabin Automaton (DRA)

A *deterministic Rabin automaton* (DRA) is a tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, Acc)$ where Q is a finite set of states, Σ an alphabet, $\delta : Q \times \Sigma \to Q$ the transition function, $q_0 \in Q$ the starting state, and

$$Acc \subseteq 2^Q \times 2^Q.$$

A run for $\sigma = A_0 A_1 A_2 \ldots \in \Sigma^\omega$ denotes an infinite sequence $q_0\, q_1\, q_2 \ldots$ of states in $\mathcal{A}$ such that $q_i \xrightarrow{A_i} q_{i+1}$ for $i \geqslant 0$. The run $q_0\, q_1\, q_2 \ldots$ is *accepting* if there exists a pair $(L, K) \in Acc$ such that

$$(\exists n \geqslant 0.\, \forall m \geqslant n.\, q_m \notin L) \;\wedge\; \left(\overset{\infty}{\exists}\, n \geqslant 0.\, q_n \in K\right).$$

The *accepted language* of $\mathcal{A}$ is

$$\mathcal{L}_\omega(\mathcal{A}) \;=\; \{\, \sigma \in \Sigma^\omega \mid \text{the run for } \sigma \text{ in } \mathcal{A} \text{ is accepting} \,\}.$$

■

Any DBA can be considered as an DRA in the following way. Assume a DBA is given with accept set F, i.e., an accepting run should visit some state in F infinitely often. The DRA with the same states and transitions and with the singleton acceptance condition $Acc = \{\, (\varnothing, F) \,\}$ is evidently equivalent to this DBA. DRAs are thus at least as expressive as DBAs.

Example 10.54. DRA for $\Box\Diamond a$

Consider the DBA for the LTL formula $\Box\Diamond a$; see Figure 10.9. The alphabet of this DBA is $2^{\{a\}} = \{\{a\}, \varnothing\}$. Each accepting run has to visit q_1 infinitely often. The DRA with $Acc = \{\, (\varnothing, \{q_1\}) \,\}$ is equivalent to this DBA: a run is accepting if and only if state q_1 is visited infinitely often. As the first component of the accept condition is $\varnothing$, the requirement that the first set is only visited finitely many times is vacuously true. ■

Recall that some ω-regular properties cannot be expressed by a DBA. This applies, e.g., to persistence properties such as the LTL formula $\Diamond\Box a$. DRAs are more expressive than

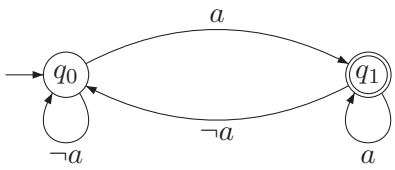


Figure 10.9: A DBA for $\Box\Diamond a$.

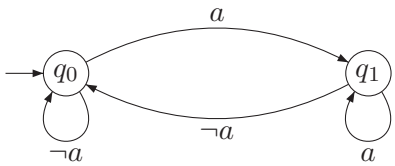


Figure 10.10: A deterministic Rabin automaton for $\Diamond\Box a$.

DBAs. For instance, consider the DRA in Figure 10.10 with the acceptance condition $Acc = \{(\{q_0\}, \{q_1\})\}$. The accepted language of this DRA is the set of infinite words whose runs end with a suffix that never visits q_0, and, thus stays forever in state q_1. These are exactly the words $A_0 A_1 \ldots$ in $Words(\Diamond\Box a)$. As $\Diamond\Box a$ cannot be described by a DBA, the class of languages accepted by a DRA is strictly larger than the class of languages that are recognizable by a DBA. In fact, the following fundamental result asserts that DRAs are as expressive as ω-regular properties.

Theorem 10.55. DRAs and ω-Regular Languages

The class of languages accepted by DRAs agrees with the class of ω-regular languages.

Proof: The proof of this result is outside the scope of this monograph. The interested reader is referred to [174] for more details. ■

Recall that NBAs are also as expressive as ω-regular properties, cf. Theorem 4.32, and thus DRAs and NBAs are equally expressive. In fact, there exists an algorithm that takes as input an NBA $\mathcal{A}$ and generates an equivalent DRA of size $2^{\mathcal{O}(n \log n)}$ where n is the size of $\mathcal{A}$.

Let us now return to our original problem: the verification of quantitative ω-regular properties over Markov chains. Let $\mathcal{M} = (S, \mathbf{P}, \iota_{\text{init}}, AP, L)$ be a finite Markov chain and let $\mathcal{A} = (Q, 2^{AP}, \delta, q_0, Acc)$ be a DRA which accepts the complement of the ω-regular LT property P. The probability of $\mathcal{M}$ to generate traces in $\mathcal{L}_\omega(\mathcal{A})$ can be computed—as for regular safety properties—on the basis of a product construction. The product of $\mathcal{M}$ and $\mathcal{A}$ is defined as the product of a Markov chain and a DFA; see Definition 10.50 (page 798).

If the acceptance condition of $\mathcal{A}$ is

$$Acc \;=\; \{\,(L_1, K_1), \ldots, (L_k, K_k)\,\},$$

then the sets L_i, K_i serve as atomic propositions in $\mathcal{M} \otimes \mathcal{A}$. The labeling function is the obvious one, i.e., if $H \in \{L_1, \ldots, L_k, K_1, \ldots, K_k\}$, then H holds in state $\langle s, q\rangle$ of $\mathcal{M} \otimes \mathcal{A}$ iff $q \in H$.

According to the following result, the computation of probabilities for satisfying ω-regular properties boils down to computing the reachability probabilities for certain bottom strongly connected components (BSCCs) in $\mathcal{M} \otimes \mathcal{A}$. These BSCCs are called accepting. A BSCC T in $\mathcal{M} \otimes \mathcal{A}$ is *accepting* if it fulfills the acceptance condition Acc. More precisely, T is accepting if and only if there exists some index $i \in \{\,1, \ldots, k\,\}$ such that

$$T \cap (S \times L_i) \;=\; \varnothing \quad \text{and} \quad T \cap (S \times K_i) \;\neq\; \varnothing.$$

Stated in words, there is no state $\langle s, q\rangle \in T$ such that $q \in L_i$ and for some state $\langle t, q'\rangle \in T$ it holds that $q \in K_i$. Thus, once such an accepting BSCC T is reached in $\mathcal{M} \otimes \mathcal{A}$, the acceptance criterion for the DRA $\mathcal{A}$ is fulfilled almost surely.

Theorem 10.56. DRA-Based Analysis of Markov Chains

Let $\mathcal{M}$ be a finite Markov chain, s a state in $\mathcal{M}$, $\mathcal{A}$ a DRA, and let U be the union of all accepting BSCCs in $\mathcal{M} \otimes \mathcal{A}$. Then:

$$Pr^{\mathcal{M}}(s \models \mathcal{A}) \;=\; Pr^{\mathcal{M}\otimes\mathcal{A}}(\,\langle s, q_s\rangle \;\models\; \Diamond U\,)$$

where $q_s = \delta(q_0, L(s))$.

Proof: Let Π be the set of paths $\pi \in Paths(s)$ in the Markov chain $\mathcal{M}$ such that $trace(\pi) \in \mathcal{L}_\omega(\mathcal{A})$, and Π^+ be the set of paths in $\mathcal{M} \otimes \mathcal{A}$ that are obtained by augmenting the paths $\pi \in \Pi$ with their corresponding runs in $\mathcal{A}$. Note that the automaton $\mathcal{A}$ is deterministic. Hence, for any path π in $\mathcal{M}$ there is a unique run in $\mathcal{A}$ for $trace(\pi)$. As the transition probabilities in $\mathcal{M} \otimes \mathcal{A}$ are not affected by the DRA $\mathcal{A}$, it follows that

$$Pr_s^{\mathcal{M}}(\Pi) \;=\; Pr_{\langle s,q_s\rangle}^{\mathcal{M}\otimes\mathcal{A}}(\Pi^+)$$

with $q_s = \delta(q_0, L(s))$.

Since the traces of the paths $\pi \in \Pi$ belong to $\mathcal{L}_\omega(\mathcal{A})$, their runs are accepting, i.e., for some acceptance pair (L_i, K_i) of $\mathcal{A}$, K_i is visited infinitely often, and from some moment on, L_i is not visited anymore. Vice versa, $trace(\pi) \in L_\omega(\mathcal{A})$ for any path π in $\mathcal{M}$ where

the extended path π^+ in $\mathcal{M} \otimes \mathcal{A}$ fulfills $\Diamond\Box\neg L_i \wedge \Box\Diamond K_i$ for some acceptance pair (L_i, K_i) of $\mathcal{A}$. Hence:

$$Pr^{\mathcal{M}}(s \models \mathcal{A}) \;=\; Pr^{\mathcal{M}\otimes\mathcal{A}}\{\, \langle s, q_s\rangle \models \bigvee_{1\leqslant i\leqslant k} (\Diamond\Box\neg L_i \wedge \Box\Diamond K_i)\,\}.$$

Whether a run $\langle s_0, q_1\rangle\,\langle s_1, q_2\rangle \ldots$ satisfies the acceptance condition of a DRA only depends on the states that are visited infinitely often. By Theorem 10.27 (page 775), such states almost surely form a BSCC in $\mathcal{M}\otimes\mathcal{A}$. Hence, the probability for $\bigvee_{1\leqslant i\leqslant k}(\Diamond\Box\neg L_i \wedge \Box\Diamond K_i)$ in $\mathcal{M} \otimes \mathcal{A}$ coincides with the probability of reaching an accepting BSCC in $\mathcal{M} \otimes \mathcal{A}$. ■

This theorem yields that the probability of $\mathcal{M}$ to generate a trace in $\mathcal{L}_\omega(\mathcal{A})$ is given by

$$Pr^{\mathcal{M}}(\mathcal{A}) \;=\; \sum_{s\in S} \iota_{\text{init}}(s) \cdot Pr^{\mathcal{M}\otimes\mathcal{A}}(\langle s, \delta(q_0, L(s))\rangle \models \Diamond U) \quad .$$

This result suggests determining the BSCCs in the product Markov chain $\mathcal{M}\otimes\mathcal{A}$ to check which BSCC is accepting (i.e., determine U), and then to compute for each of the accepting BSCCs the reachability probability. The first stage of this algorithm can be performed by a standard graph analysis. To check whether a BSCC is accepting amounts to checking all pairs $(L_i, K_i) \in \mathit{Acc}$. Finally, reachability probabilities can be determined by solving a set of linear equations, as indicated earlier in this chapter (see Section 10.1.1). The size of this linear equation system is linear in the size of the Markov chain $\mathcal{M}$ and the DRA $\mathcal{A}$. The overall time complexity of this procedure is

$$\mathcal{O}(\text{poly}(\mathit{size}(\mathcal{M}), \mathit{size}(\mathcal{A}))).$$

To check whether almost all traces of $\mathcal{M}$ are accepted by the DRA $\mathcal{A}$, it suffices to check whether all BSCCs of $\mathcal{M}\otimes\mathcal{A}$ that are reachable from some initial state $\langle s, \delta(q_0, L(s))\rangle$ in $\mathcal{M}\otimes\mathcal{A}$ for which $\iota_{\text{init}}(s) > 0$ are accepting.

Remark 10.57. Measurability of ω-Regular Properties

So far, we have implicitly assumed the measurability of ω-regular properties. The fact that ω-regular properties are indeed measurable follows by the following argument.

Let $\mathcal{M} = (S, \mathbf{P}, \iota_{\text{init}}, AP, L)$ be a Markov chain and P an ω-regular property which is represented by the DRA $\mathcal{A} = (Q, 2^{AP}, \delta, q_0, \mathit{Acc})$ with $\mathit{Acc} = \{\,(L_1, K_1), \ldots, (L_k, K_k)\,\}$. We need to prove that:

$$\Pi \;=\; \{\, \pi \in \mathit{Paths}(\mathcal{M}) \mid \mathit{trace}(\pi) \in \underbrace{\mathcal{L}_\omega(\mathcal{A})}_{=P} \,\}$$

is measurable. Each path $\pi = s_0\, s_1 \ldots \in \Pi$ is lifted to a path π^+ in the product Markov chain $\mathcal{M}\otimes\mathcal{A}$ such that $\pi^+ = \langle s_0, q_1\rangle\,\langle s_1, q_2\rangle \ldots$. The path π^+ results from π by "adding" the states in $\mathcal{A}$ of the (unique) run $q_0\, q_1\, q_2 \ldots$ in $\mathcal{A}$ for $\mathit{trace}(\pi)$. Then:

$$
\begin{aligned}
& \pi \in \Pi, \text{ i.e., } \mathit{trace}(\pi) \in \mathcal{L}_\omega(\mathcal{A}) \\
\text{iff} \quad & \text{the (unique) run } q_0\, q_1\, q_2 \ldots \text{ for } \mathit{trace}(\pi) \text{ in } \mathcal{A} \text{ is accepting} \\
\text{iff} \quad & \text{the path } \pi^+ = \langle s_0, q_1\rangle \langle s_1, q_2\rangle \ldots \text{ in } \mathcal{M} \otimes \mathcal{A} \text{ satisfies} \\
& \qquad \bigvee_{1 \leqslant i \leqslant k} (\Diamond\Box\neg L_i \wedge \Box\Diamond K_i).
\end{aligned}
$$

Let $\varphi_i = \Diamond\Box\neg L_i \wedge \Box\Diamond K_i$, for $0 < i \leqslant k$, and Π_i the set of all paths π in $\mathcal{M}$ such that $\pi^+ \models \varphi_i$. Clearly, $\Pi = \Pi_1 \cup \ldots \cup \Pi_k$. To show the measurability of Π it thus suffices to prove the measurability of Π_i. Let us consider Π_i in somewhat more detail. It follows that $\Pi_i = \Pi_i^{\Diamond\Box} \cap \Pi_i^{\Box\Diamond}$ where $\Pi_i^{\Diamond\Box}$ is the set of paths π in $\mathcal{M}$ such that $\pi^+ \models \Diamond\Box\neg L_i$, and $\Pi_i^{\Box\Diamond}$ is the set of paths π in $\mathcal{M}$ such that $\pi^+ \models \Box\Diamond K_i$. It remains to show that $\Pi_i^{\Diamond\Box}$ and $\Pi_i^{\Box\Diamond}$ are measurable. The set $\Pi_i^{\Diamond\Box}$ can be written as

$$
\bigcup_{n \geqslant 0} \bigcap_{m \geqslant n} \bigcup_{s_0 \ldots s_n \ldots s_m} \mathit{Cyl}(s_0 \ldots s_n \ldots s_m)
$$

where $s_0 \ldots s_n \ldots s_m$ ranges over all finite path fragments in $\mathcal{M}$ such that $q_j \notin L_i$ for $n < j \leqslant m{+}1$ for the induced run $q_0\, q_1 \ldots q_{n+1} \ldots q_{m+1}$ in DRA $\mathcal{A}$. Thus, $\Pi_i^{\Diamond\Box}$ is measurable.

Similarly, the set $\Pi_i^{\Box\Diamond}$ can be written as

$$
\bigcap_{n \geqslant 0} \bigcup_{m \geqslant n} \bigcup_{s_0 \ldots s_n \ldots s_m} \mathit{Cyl}(s_0 \ldots s_n \ldots s_m)
$$

where $s_0 \ldots s_n \ldots s_m$ ranges over all finite path fragments in $\mathcal{M}$ such that $q_{m+1} \in K_i$ for the induced run $q_0\, q_1 \ldots q_{n+1} \ldots q_{m+1}$ in $\mathcal{A}$. Hence, both sets $\Pi_i^{\Diamond\Box}$ and $\Pi_i^{\Box\Diamond}$ arise through countable unions and intersections of cylinder sets and are therefore measurable. ■

Although the approach with DRA is conceptually simple, it has the drawback of a double exponential blowup when starting with an LTL formula for the LT property. We mention without proof that there exist LTL formulae φ_n of size $\mathcal{O}(\mathrm{poly}(n))$ for which the smallest DRA representation for φ_n has 2^{2^n} states. Using alternative techniques, the double exponential blowup can be reduced to a single exponential blowup. The details of these advanced techniques fall outside the scope of this monograph. Algorithms with a better asymptotic worst-case complexity cannot be expected, as the qualitative model-checking problem for finite Markov chains "given a finite Markov chain $\mathcal{M}$ and an LTL formula φ, does $Pr(\mathcal{M} \models \varphi) = 1$ hold?" is PSPACE-complete. This result is due to Vardi [407] and stated here without proof.

Theorem 10.58.

The qualitative model-checking problem for finite Markov chains is PSPACE*-complete.*

10.4 PCTL* and Probabilistic Bisimulation

This section introduces probabilistic bisimulation for Markov chains, and shows that this notion of bisimulation coincides with PCTL equivalence. That is to say, PCTL equivalence serves as a logical characterization of probabilistic bisimulation. Vice versa, probabilistic bisimulation serves as an operational characterization of PCTL equivalence. It is shown that the same applies to PCTL*, a logic that results from the state formulae in PCTL and allowing LTL formulae as path formulae. These results may thus be considered as the quantitative analogue to the result that bisimulation on transition systems coincides with CTL and CTL* equivalence; see Theorem 7.20 (page 469).

10.4.1 PCTL*

The logic PCTL* extends PCTL by dropping the requirement that any temporal operator must be proceeded by a state formula. In addition, it allows for boolean combinations of path formulae. Thus, the logic PCTL* permits Boolean combinations of formulae of the form $\mathbb{P}_J(\varphi)$ where the interval J specifies a probability bound, whereas φ is an LTL formula whose substate formulae are PCTL* state formulae. The logic PCTL* is strictly more expressive than LTL (with probability bounds) and PCTL.

Definition 10.59. Syntax of PCTL*

PCTL* *state formulae* over the set AP of atomic propositions are formed according to the following grammar:

$$\Phi ::= \text{true} \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \mathbb{P}_J(\varphi)$$

where $a \in AP$, φ is a path formula and $J \subseteq [0,1]$ is an interval with rational bounds. PCTL* *path formulae* are formed according to the following grammar:

$$\varphi ::= \Phi \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc\, \varphi \mid \varphi_1 \,\mathsf{U}\, \varphi_2$$

where Φ is a PCTL* state formula. ■

Other Boolean operators and the temporal modalities $\Diamond$, $\Box$, W, and R can be derived as in CTL*. The step-bounded until operator has been omitted from the syntax of PCTL* path formulae for the sake of simplicity. It can be defined in terms of the other operators in the following way:

$$\varphi_1 \,\mathsf{U}^{\leqslant n} \varphi_2 \;=\; \bigvee_{0 \leqslant i \leqslant n} \psi_i \quad \text{where } \psi_0 = \varphi_2 \text{ and } \psi_{i+1} = \varphi_1 \wedge \bigcirc \psi_i \text{ for } i \geqslant 0.$$

The logic PCTL can thus be regarded as a sublogic of PCTL*.

For a given Markov chain $\mathcal{M} = (S, \mathbf{P}, \iota_{\text{init}}, AP, L)$, the satisfaction relation for PCTL* state- and path formulae is defined as for PCTL and CTL*, respectively. For example,

$$s \models \mathbb{P}_J(\varphi) \quad \text{iff} \quad Pr(s \models \varphi) \in J$$

where $Pr(s \models \varphi) = Pr_s\{\, \pi \in Paths(s) \mid \pi \models \varphi \,\}$. The satisfaction relation for the path formulae is exactly as in CTL*. Let $Sat_{\mathcal{M}}(\Phi)$, or briefly $Sat(\Phi)$, denote the set $\{\, s \in S \mid s \models \Phi \,\}$. A PCTL* state formula Φ is said to hold for $\mathcal{M}$, denoted $\mathcal{M} \models \Phi$, if $s \models \Phi$ for all states $s \in S$ with $\iota_{\text{init}}(s) > 0$.

The equivalence of PCTL* state formulae is defined as for PCTL, i.e.,

$$\Phi \equiv \Psi \quad \text{iff} \quad Sat_{\mathcal{M}}(\Phi) = Sat_{\mathcal{M}}(\Psi) \text{ for all Markov chains } \mathcal{M}.$$

(It is assumed that the set AP of atomic propositions is fixed.) In case we are restricted to finite Markov chains, we write $\equiv_f$, i.e.,

$$\Phi \equiv_f \Psi \quad \text{iff} \quad Sat_{\mathcal{M}}(\Phi) = Sat_{\mathcal{M}}(\Psi) \text{ for all finite Markov chains } \mathcal{M}.$$

The results established for repeated reachability and persistence properties (see Theorem 10.47, page 795 and Theorem 10.48, page 795, respectively, can now be rephrased by

$$\begin{aligned} \mathbb{P}_J(\Box\Diamond\Phi) &\equiv_f \mathbb{P}_J(\Diamond\mathbb{P}_{=1}(\Box\mathbb{P}_{=1}(\Diamond\Phi))), \\ \mathbb{P}_J(\Diamond\Box\Phi) &\equiv_f \mathbb{P}_J(\Diamond\mathbb{P}_{=1}(\Box\Phi)). \end{aligned}$$

Let us now consider the model-checking problem for PCTL*, i.e., for a given finite Markov chain $\mathcal{M}$ and PCTL formula Φ, does $\mathcal{M} \models \Phi$ hold? This problem can be tackled by combining the procedure for PCTL model checking and the techniques discussed before for the quantitative analysis of LTL formulae. The main procedure goes along the lines of the model-checking algorithm for CTL* and relies on a bottom-up treatment of the parse tree of Φ. For each inner node (corresponding to state subformula Ψ of Φ), the satisfaction set $Sat(\Psi) = \{\, s \in S \mid s \models \Psi \,\}$ is computed. For the propositional logic fragment this computation is obvious and is as for CTL. The interesting case is a state formula of the form $\Psi = \mathbb{P}_J(\varphi)$. In order to treat such formulae, first all maximal state subformulae of φ are replaced by new atomic propositions. Intuitively, these atomic propositions represent the satisfaction sets for these subformulae. Due to the bottom-up nature of the model-checking algorithm, these satisfaction sets have been determined already, and this replacement amounts to labeling the states in these sets. Applying this replacement procedure to φ yields an LTL formula φ'. Using the techniques explained in Section 10.3, one computes for each state s the probability $Pr(s \models \varphi')$ of satisfying φ' (and thus φ). The returned result is then:

$$Sat(\Psi) = \{\, s \in S \mid Pr(s \models \varphi) \in J \,\} \quad .$$

Due to the double-exponential transformation of an LTL formula φ' into a deterministic Rabin automaton, the time complexity of model-checking PCTL* is double exponential in $|\varphi|$ and polynomial in the size of the Markov chain $\mathcal{M}$. Using alternative techniques may yield a single exponential time complexity in $|\varphi|$. A further improvement cannot be expected due to the PSPACE-completeness of the qualitative model-checking problem for LTL; see Theorem 10.58, page 805.

10.4.2 Probabilistic Bisimulation

To compare the behavior of transition systems, bisimulation and simulation relations have been extensively treated in Chapter 7. Bisimulation relations are equivalences requiring two bisimilar states to be equally labeled and exhibit identical stepwise behavior. It can be lifted to transition systems by comparing their initial states. Bisimilar states thus need to be able to mimic each individual step of each other. This section considers a notion of bisimulation on Markov chains that takes the transition probabilities into account. They can be viewed as a quantitative variant of bisimulation for transition systems. The crucial idea is to require bisimulation-equivalent states to have the same transition probability for each equivalence class (under bisimulation). As for transition systems, bisimulation-equivalent states must be equally labeled.

Definition 10.60. Bisimulation for Markov Chains

Let $\mathcal{M} = (S, \mathbf{P}, \iota_{\text{init}}, AP, L)$ be a Markov chain. A *probabilistic bisimulation* on $\mathcal{M}$ is an equivalence relation $\mathcal{R}$ on S such that for all states $(s_1, s_2) \in \mathcal{R}$:

1. $L(s_1) = L(s_2)$.
2. $\mathbf{P}(s_1, T) = \mathbf{P}(s_2, T)$ for each equivalence class $T \in S/\mathcal{R}$.

States s_1 and s_2 are *bisimulation-equivalent* (or bisimilar), denoted $s_1 \sim_{\mathcal{M}} s_2$, if there exists a bisimulation $\mathcal{R}$ on $\mathcal{M}$ such that $(s_1, s_2) \in \mathcal{R}$. ■

A probabilistic bisimulation is often referred to as bisimulation in the sequel. The first condition states that the states are equally labeled. The last condition requires that for bisimilar states the probability of moving by a single transition to some equivalence class is equal. Recall that $\mathbf{P}(s, T) = \sum_{t \in T} \mathbf{P}(s, t)$ denotes the probability of moving from state s directly to some state in T. As opposed to bisimulation on states in transition systems (see Definition 7.7 on page 456), any probabilistic bisimulation is required to be

an equivalence—otherwise it would be senseless to refer to equivalence classes in the last constraint. This is in contrast to the nonprobabilistic case where bisimulation relations for a single transition system TS need not be symmetric (although the coarsest bisimulation $\sim_{TS}$ is, in fact, an equivalence).

As for transition systems, the above notion of (probabilistic) bisimulation equivalence for the states of a single Markov chain can be extended to compare two Markov chains $\mathcal{M}_1$, $\mathcal{M}_2$. Let $\mathcal{M}_1$, $\mathcal{M}_2$ be Markov chains over the same set of atomic propositions with initial distributions ι^1_{init} and ι^2_{init}, respectively. Consider the Markov chain $\mathcal{M} = \mathcal{M}_1 \uplus \mathcal{M}_2$ that results from the disjoint union of $\mathcal{M}_1$ and $\mathcal{M}_2$. Then $\mathcal{M}_1$ and $\mathcal{M}_2$ are bisimilar if

$$\iota^1_{\text{init}}(T) \;=\; \iota^2_{\text{init}}(T)$$

for each bisimulation equivalence class T of $\mathcal{M} = \mathcal{M}_1 \uplus \mathcal{M}_2$. Here, $\iota_{\text{init}}(T)$ denotes $\sum\limits_{t \in T} \iota_{\text{init}}(t)$.

Example 10.61. Bisimulation for Markov Chains

Consider the Markov chain in Figure 10.11. (Note that it consists of two mutually unreachable parts.) The reflexive, symmetric, and transitive closure of the relation

$$\mathcal{R} \;=\; \{\, (s_1, s_2), (u_1, u_3), (u_2, u_3), (v_1, v_3), (v_2, v_3) \,\}$$

is a probabilistic bisimulation. This can be seen as follows. The equivalence classes are $T_1 = [s_1]_\sim = \{\, s_1, s_2 \,\}$, $T_2 = [u_1]_\sim = \{\, u_1, u_2, u_3 \,\}$ and $T_3 = [v_1]_\sim = \{\, v_1, v_2, v_3 \,\}$. It follows that

$$\mathbf{P}(s_1, T_1) = \mathbf{P}(s_2, T_1) = 0, \mathbf{P}(s_1, T_2) = \mathbf{P}(s_2, T_2) = \frac{2}{3}, \mathbf{P}(s_1, T_3) = \mathbf{P}(s_2, T_3) = \frac{1}{3}.$$

In a similar way, it can be established that for all states in T_2 and T_3 the probabilities of transition to each of the equivalence classes are the same. ■

Definition 10.62. Bisimulation Quotient

Let $\mathcal{M} = (S, \mathbf{P}, \iota_{\text{init}}, AP, L)$ be a Markov chain. The quotient Markov chain $\mathcal{M}/\sim_{\mathcal{M}}$ is defined by

$$\mathcal{M}/\sim_{\mathcal{M}} = (S/\sim_{\mathcal{M}}, \mathbf{P}', \iota'_{\text{init}}, AP, L')$$

where $\mathbf{P}'([s]_\sim, [t]_\sim) = \mathbf{P}(s, [t]_\sim)$, $\iota'_{\text{init}}([s]_\sim) = \sum_{s' \in [s]} \iota_{\text{init}}(s)$ and $L'([s]_\sim) = L(s)$. ■

The state space of the quotient Markov chain $\mathcal{M}/\sim_{\mathcal{M}}$ is the set of equivalence classes under $\sim_{\mathcal{M}}$. The transition probability from equivalence class $[s]_\sim$ to $[t]_\sim$ equals $\mathbf{P}(s, [t]_\sim)$.

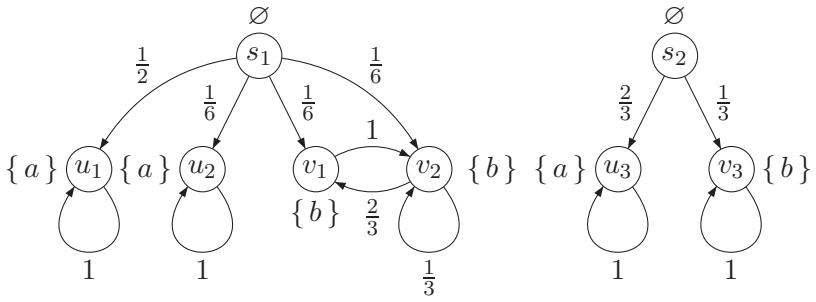


Figure 10.11: Two examples of Markov chains for which $s_1 \sim_{\mathcal{M}} s_2$.

Note that this is well-defined as $\mathbf{P}(s, T) = \mathbf{P}(s', T)$ for all $s \sim s'$ and all bisimulation equivalence classes T.

Example 10.63. The Craps Game

Consider the Markov chain modeling the craps game; see Figure 10.3 (page 751). Assume all states are labeled with the empty set, except for $L(won) = \{ won \}$. The equivalence relation that induces the following partitioning of the state space

$$\{ start \}, \{ won \}, \{ lost \}, \{ 4, 10 \}, \{ 5, 9 \}, \{ 6, 8 \}$$

is a probabilistic bisimulation. This can be seen as follows. The fact that state *won* is not bisimilar to any other state is clear as it is labeled differently. State *lost* is not bisimilar to any other state, as it is the only absorbing state labeled with $\varnothing$. As state *start* is the only state that can move to $\{ won \}$ with probability $\frac{2}{9}$, it is not bisimilar to any other state. Consider now, e.g., states 5 and 9. Both states can move to *won* with the same probability. The same applies to any equivalence class T. States 5 and 9 are thus bisimilar. A similar reasoning applies to the other states. The quotient Markov chain is depicted in Figure 10.12. ■

The remainder of this section is focused on establishing that bisimulation-equivalent states satisfy the same PCTL* formulae. This means in particular that s and any state in $[s]_\sim$ satisfy the same PCTL* formulae. Checking whether $\mathcal{M} \models \Phi$ for PCTL* formula Φ may thus be established by checking $\mathcal{M}/\sim \models \Phi$. In fact, it will be shown that PCTL equivalence, PCTL* equivalence, and probabilistic bisimulation *coincide*. This means that in order to show that two states are not bisimilar it suffices to indicate a PCTL (or PCTL*) formula that distinguishes them.

The key to establishing that bisimulation-equivalent states satisfy the same PCTL* formula is the following observation. Let $s \sim_{\mathcal{M}} s'$ where s and s' are states in the Markov

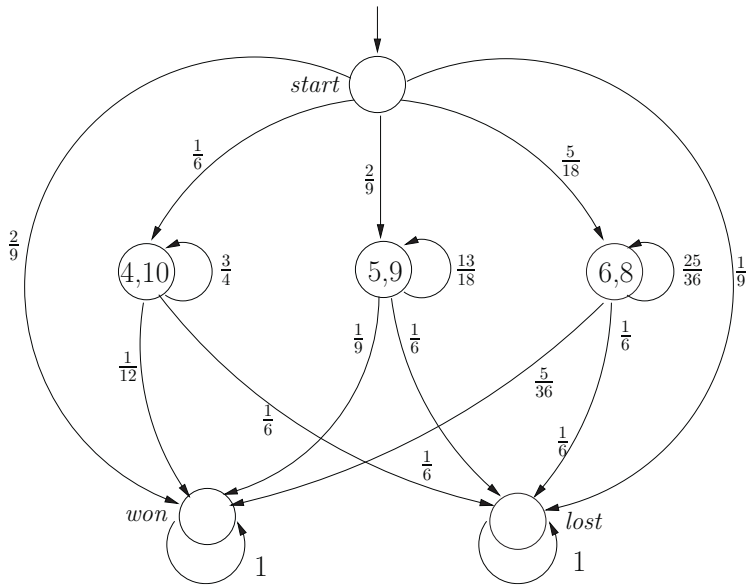


Figure 10.12: Bisimulation quotient for the craps game.

chain $\mathcal{M}$. Then, the probability measures $Pr_s^{\mathcal{M}}$ and $Pr_{s'}^{\mathcal{M}}$ agree for all measurable sets of paths that are closed under statewise bisimulation equivalence. In order to formally prove this property, we need some additional notations and some standard concepts from measure theory. Let us first lift the notion of bisimulation to paths.

Definition 10.64. Bisimulation-Equivalent Paths

The infinite paths $\pi_1 = s_{0,1}\, s_{1,1}\, s_{2,1} \ldots$ and $\pi_2 = s_{0,2}\, s_{1,2}\, s_{2,2} \ldots$ in a Markov chain $\mathcal{M}$ are *bisimulation equivalent*, denoted $\pi_1 \sim_{\mathcal{M}} \pi_2$, if they are statewise bisimilar:

$$\pi_1 \sim_{\mathcal{M}} \pi_2 \quad \text{if and only if} \quad s_{i,1} \sim_{\mathcal{M}} s_{i,2} \text{ for all } i \geqslant 0 \quad .$$

■

Definition 10.65. Bisimulation-Closed σ-Algebra

Let $\mathcal{M}$ be a Markov chain with state space S and $T_0, T_1, \ldots, T_n \in S/\sim_{\mathcal{M}}$ be equivalence classes of $\sim_{\mathcal{M}}$. The *bisimulation-closed σ-algebra* $\mathfrak{E}_{\sim_{\mathcal{M}}}^{\mathcal{M}}$ denotes the σ-algebra generated by the sets $Cyl(T_0\, T_1 \ldots T_n)$ where $Cyl(T_0\, T_1 \ldots T_n)$ is the set of all paths $t_0\, t_1 \ldots t_n\, t_{n+1}\, t_{n+2} \ldots$ with $t_i \in T_i$ for $0 \leqslant i \leqslant n$. ■

All sets in the bisimulation-closed σ-algebra $\mathfrak{E}_{\sim}^{\mathcal{M}}$ are measurable with respect to the stan-

dard σ-algebra $\mathfrak{E}^{\mathcal{M}}$ associated with $\mathcal{M}$ (see Definition 10.10). That is to say:

$$\mathfrak{E}^{\mathcal{M}}_{\sim} \subseteq \mathfrak{E}^{\mathcal{M}} \quad .$$

This inclusion is due to the fact that the basic events $Cyl(T_0\, T_1 \ldots T_n)$ of the σ-algebra $\mathfrak{E}^{\mathcal{M}}_{\sim}$ are countable unions of basic elements—cylinder sets spanned by the finite path fragments in $\mathcal{M}$—of the σ-algebra $\mathfrak{E}^{\mathcal{M}}$. Formally:

$$Cyl(T_0\, T_1 \ldots T_n) \;=\; \bigcup_{\substack{t_0\, t_1 \ldots t_n \,\in\, Paths_{fin}(\mathcal{M}) \\ t_i \in T_i, 0 \leqslant i \leqslant n}} Cyl(t_0\, t_1 \ldots t_n)$$

where $T_0, T_1, \ldots, T_n \in S/\!\sim_{\mathcal{M}}$ are bisimulation-equivalence classes. Thus, the bisimulation-closed σ-algebra $\mathfrak{E}^{\mathcal{M}}_{\sim}$ is a so-called *sub-σ-algebra* of the σ-algebra $\mathfrak{E}^{\mathcal{M}}$. We refer to the elements of $\mathfrak{E}^{\mathcal{M}}_{\sim}$ as *bisimulation-closed* events.

In fact, the events in the bisimulation-closed σ-algebra $\mathfrak{E}^{\mathcal{M}}_{\sim}$ are bisimulation-closed measurable sets of paths in $\mathfrak{E}^{\mathcal{M}}$. Let $\Pi \in \mathfrak{E}^{\mathcal{M}}$ be a measurable set of paths. Note that measurability is understood with respect to the standard σ-algebra $\mathfrak{E}^{\mathcal{M}}$ on the paths of $\mathcal{M}$. The set Π is bisimulation-closed if for any $\pi_1 \in \Pi$ and π_2 such that $\pi_1 \sim_{\mathcal{M}} \pi_2$ it holds that $\pi_2 \in \Pi$. That is:

$$\mathfrak{E}^{\mathcal{M}}_{\sim} \;=\; \{\, \Pi \in \mathfrak{E}^{\mathcal{M}} \mid \Pi \text{ is bisimulation-closed} \,\}.$$

The following result asserts that the probability measure $Pr^{\mathcal{M}}_{s}$ for bisimulation-closed sets of paths agree for bisimilar states.

Lemma 10.66. Preservation of Probabilities of Bisimulation-Closed Events
Let $\mathcal{M}$ be a Markov chain. For all states s_1, s_2 in $\mathcal{M}$:

$$s_1 \sim_{\mathcal{M}} s_2 \quad \textit{implies} \quad Pr^{\mathcal{M}}_{s_1}(\Pi) \;=\; Pr^{\mathcal{M}}_{s_2}(\Pi)$$

for all bisimulation-closed events $\Pi \subseteq Paths(\mathcal{M})$.

Proof: Since the bisimulation-closed σ-algebra $\mathfrak{E}^{\mathcal{M}}_{\sim}$ is closed under intersection, by standard results of measure theory it holds that: for fixed function $f : (S/\!\sim_{\mathcal{M}})^{+} \to [0,1]$, there exists at most one probability measure μ on the bisimulation-closed σ-algebra $\mathfrak{E}^{\mathcal{M}}_{\sim}$ such that

$$\mu(Cyl(T_0\, T_1 \ldots T_n)) \;=\; f(T_0\, T_1 \ldots T_n)$$

for all bisimulation equivalence classes T_i, $0 \leqslant i \leqslant n$.

This result yields that it suffices to show that $s_1 \sim_{\mathcal{M}} s_2$ implies that $Pr^{\mathcal{M}}_{s_1}$ and $Pr^{\mathcal{M}}_{s_2}$ agree on the basic events $Cyl(T_0\ T_1\ \ldots T_n)$ of $\mathfrak{E}^{\mathcal{M}}_{\sim}$. For bisimulation equivalence classes T, U of $\mathcal{M}$, let $\mathbf{P}(T,U)$ denote the value $\mathbf{P}(t,U)$ for all (some) $t \in T$. Then we have

$$\begin{aligned} & Pr^{\mathcal{M}}_{s_1}(\, Cyl(T_0T_1\ldots T_n)\,) \\ = \;& \mathbf{P}(T_0,T_1)\cdot\mathbf{P}(T_1,T_2)\cdot\ldots\cdot\mathbf{P}(T_{n-1},T_n) \\ = \;& Pr^{\mathcal{M}}_{s_2}(\, Cyl(T_0T_1\ldots T_n)\,) \end{aligned}$$

provided that $s_1, s_2 \in T_0$. Otherwise, i.e., if $s_1, s_2 \notin T_0$, then:

$$Pr^{\mathcal{M}}_{s_1}(\, Cyl(T_0T_1\ldots T_n)\,) \;=\; 0 \;=\; Pr^{\mathcal{M}}_{s_2}(\, Cyl(T_0T_1\ldots T_n)\,).$$

■

For finitely-branching transition systems, CTL-equivalence, CTL*-equivalence, and bisimulation equivalence all coincide. An analogous result holds for PCTL, PCTL*, and probabilistic bisimulation equivalence. In fact, the restriction to finitely-branching systems is not necessary. That is, PCTL, PCTL*, and probabilistic bisimulation equivalence coincide for any arbitrary, possibly infinite, Markov chain. In the probabilistic setting, nonbisimilar states can even be distinguished by a formula of a (small) fragment of PCTL that just consists of atomic propositions, conjunction, and the operator $\mathbb{P}_{\leqslant p}(\bigcirc\,\cdot)$. As in the nonprobabilistic setting, the until operator is thus not necessary for logically characterizing bisimulation. As opposed to the nonprobabilistic case where full propositional logic (containing conjunction and negation) is necessary, negation is *not* needed in the probabilistic setting.

Let PCTL$^-$ denote the following fragment of PCTL. State formulae in PCTL$^-$ are formed according to

$$\Phi \;::=\; a \;\mid\; \Phi_1 \wedge \Phi_2 \;\mid\; \mathbb{P}_{\leqslant p}(\bigcirc\,\Phi)$$

where $a \in AP$ and p is a rational number in $[0,1]$. Negation is present neither as an operator in PCTL$^-$ nor can it be expressed. The results indicated above are summarized in the following theorem:

Theorem 10.67. PCTL/PCTL* and Bisimulation Equivalence

Let $\mathcal{M}$ be a Markov chain and s_1, s_2 states in $\mathcal{M}$. Then, the following statements are equivalent:

(a) $s_1 \sim_{\mathcal{M}} s_2$.

(b) s_1 *and* s_2 *are PCTL*-equivalent, i.e., fulfill the same PCTL* formulae.*

(c) s_1 *and* s_2 *are PCTL-equivalent, i.e., fulfill the same PCTL formulae.*

(d) s_1 *and* s_2 *are PCTL$^-$-equivalent, i.e., fulfill the same PCTL$^-$ formulae.*

Proof: (a) $\Longrightarrow$ (b): Using Lemma 10.66 on page 812 it can be shown by structural induction on the syntax of PCTL* state formula Φ and PCTL* path formula φ:

(1) For states s_1, s_2 in $\mathcal{M}$, if $s_1 \sim_{\mathcal{M}} s_2$, then:

(1.1) $s_1 \models \Phi$ if and only if $s_2 \models \Phi$.

(1.2) $Pr(s_1 \models \varphi) = Pr(s_2 \models \varphi)$.

(2) For paths π_1, π_2 in $\mathcal{M}$, if $\pi_1 \sim_{\mathcal{M}} \pi_2$, then $\pi_1 \models \varphi$ if and only if $\pi_2 \models \varphi$.

The proof of the first part is fairly similar to the nonprobabilistic case (see Theorem 7.20 on page 469). We only consider statement (1.2) and assume as an induction hypothesis that (2) holds for φ. Let $s_1 \sim_{\mathcal{M}} s_2$ and Π be the set of paths in $\mathcal{M}$ satisfying φ, i.e., $\Pi = \{\, \pi \in \mathit{Paths}(\mathcal{M}) \mid \pi \models \varphi \,\}$. The induction hypothesis applied to φ (see item (2)) yields that Π is bisimulation-closed. Hence, by Lemma 10.66 it follows that

$$Pr(s_1 \models \varphi) \;=\; Pr_{s_1}^{\mathcal{M}}(\Pi) \;=\; Pr_{s_2}^{\mathcal{M}}(\Pi) \;=\; Pr(s_2 \models \varphi).$$

(b) $\Longrightarrow$ (c) and (c) $\Longrightarrow$ (d): Obvious, since PCTL is a sublogic of PCTL* and PCTL$^-$ is a sublogic of PCTL.

(c) $\Longrightarrow$ (a): Prior to proving that (a) is a consequence of (d), we first treat a simpler case and prove that PCTL-equivalent states in a *finite* Markov chain are probabilistically bisimilar. The aim of this step is to show that roughly the same arguments as in the nonprobabilistic case can be applied. We have to show that the relation

$$\mathcal{R} \;=\; \{\, (s_1, s_2) \in S \times S \mid\; s_1 \equiv_{PCTL} s_2 \,\}$$

is a probabilistic bisimulation. (Here, $\equiv_{PCTL}$ denotes PCTL equivalence of states.) Let $(s_1, s_2) \in \mathcal{R}$. As s_1 and s_2 fulfill the same atomic propositions, it follows that $L(s_1) = L(s_2)$. It remains to prove that $\mathbf{P}(s_1, T) = \mathbf{P}(s_2, T)$ for each equivalence class T under $\mathcal{R}$. Let T, U be $\mathcal{R}$-equivalence classes with $T \neq U$, and PCTL formula $\Phi_{T,U}$ be such that $\mathit{Sat}(\Phi_{T,U}) \subseteq T$ and $\mathit{Sat}(\Phi_{T,U}) \cap U = \varnothing$. Define:

$$\Phi_T \;=\; \bigwedge_{\substack{U \in S/\sim_{\mathcal{M}} \\ U \neq T}} \Phi_{T,U}.$$

It is evident that $Sat(\Phi_T) = T$. That is to say, Φ_T is a PCTL *master formula* for the equivalence class T. Without loss of generality, assume $\mathbf{P}(s_1, T) \leqslant \mathbf{P}(s_2, T)$. Let $s_1 \models \mathbb{P}_{\leqslant p}(\Phi_T)$ where $p = \mathbf{P}(s_1, T)$. As $s_1 \equiv_{PCTL} s_2$, it follows that $s_2 \models \mathbb{P}_{\leqslant p}(\Phi_T)$. But then $\mathbf{P}(s_1, T) = p = \mathbf{P}(s_2, T)$. This yields condition (2) for $\mathcal{R}$.

(d) $\Longrightarrow$ (a): Assume $\mathcal{M}$ is an arbitrary (possibly infinite) Markov chain. The goal is to show that

$$\mathcal{R} \;=\; \{\, (s_1, s_2) \in S \times S \mid \; s_1 \equiv_{PCTL^-} s_2 \,\}$$

is a probabilistic bisimulation. Let $(s_1, s_2) \in \mathcal{R}$. The fact that s_1 and s_2 are equally labeled is established as in the previous part of the proof (i.e., (c) implies (a)). However, the above argument to prove that $\mathbf{P}(s_1, T) = \mathbf{P}(s_2, T)$ for any $\mathcal{R}$-equivalence class T is not applicable in case there are infinitely many bisimulation equivalence classes. (Note that then Φ_T would be defined by an infinite conjunction.) Let the satisfaction sets $Sat(\Phi)$ for PCTL$^-$ state formula Φ be basic events on the state space S of $\mathcal{M}$ and $\mathfrak{E}_S$ be the smallest σ-algebra on S that contains the sets $Sat(\Phi)$. Since the set of all PCTL$^-$ formulae is countable, any PCTL$^-$-equivalence class $T \in S/\mathcal{R}$ can be written as the countable intersection of the satisfaction sets $Sat(\Phi)$ where Φ is a PCTL$^-$ formula and $T \subseteq Sat(\Phi)$. Thus, all PCTL$^-$-equivalence classes $T \in S/\mathcal{R}$ belong to $\mathfrak{E}_S$.

As PCTL$^-$ permits conjunction, the set of all satisfaction sets $Sat(\Phi)$ is closed under finite intersections. By a standard result of measure theory, we have that for every probability measure μ_1, μ_2 on $\mathfrak{E}_S$:

$$\mu_1(Sat(\Phi)) = \mu_2(Sat(\Phi)) \text{ for any PCTL}^- \text{ formula } \Phi \quad \text{implies} \quad \mu_1 = \mu_2.$$

In the remainder of the proof, this result is exploited to show that $Pr(s_1, T) = Pr(s_2, T)$ for all states s_1, s_2 with $(s_1, s_2) \in \mathcal{R}$ and all $T \in S/\mathcal{R}$.

For state s in $\mathcal{M}$, let the probability measure μ_s on $\mathfrak{E}_S$ be defined by

$$\mu_s(T) \;=\; \mathbf{P}(s, T) \;=\; \sum_{t \in T} \mathbf{P}(s, t) \quad \text{for } T \in \mathfrak{E}_S.$$

Clearly, $\mu_s(Sat(\Phi)) \;=\; \mathbf{P}(s, Sat(\Phi))$ for any PCTL$^-$ formula Φ. The goal is now to show that $\mu_{s_1} = \mu_{s_2}$ for PCTL$^-$-equivalent states s_1 and s_2. This is established as follows. Let $(s_1, s_2) \in \mathcal{R}$ and Φ be a PCTL$^-$ formula such that $\mathbf{P}(s_1, Sat(\Phi))$ equals p, say, and $\mathbf{P}(s_2, Sat(\Phi))$ equals q. Without loss of generality, assume $p \leqslant q$. Then:

$$s_1 \models \mathbb{P}_{\leqslant p}(\bigcirc \Phi).$$

As $\mathbb{P}_{\leqslant p}(\bigcirc \Phi)$ is a PCTL$^-$ formula and $(s_1, s_2) \in \mathcal{R}$, it follows that $s_2 \models \mathbb{P}_{\leqslant p}(\bigcirc \Phi)$. But then $q = \mathbf{P}(s_2, Sat(\Phi)) \leqslant p \leqslant q$, and hence:

$$\mu_{s_1}(Sat(\Phi)) \;=\; p \;=\; q \;=\; \mu_{s_2}(Sat(\Phi)).$$

The probability measures μ_{s_1} and μ_{s_2} on $\mathfrak{E}_S$ thus coincide for all basic events $Sat(\Phi)$ of $\mathfrak{E}_S$. As the set of basic events $Sat(\Phi)$ is closed under intersection, any measure on $\mathfrak{E}_S$ is uniquely determined by its values on the basic events. Thus, $\mu_{s_1}(T) = \mu_{s_2}(T)$ for all $T \in \mathfrak{E}_S$. All PCTL$^-$-equivalence classes $T \in S/\mathcal{R}$ belong to $\mathfrak{E}_S$, so:

$$\mathbf{P}(s_1, T) \;=\; \mu_{s_1}(T) \;=\; \mu_{s_2}(T) \;=\; \mathbf{P}(s_2, T).$$

Thus, $\mathcal{R}$ is a probabilistic bisimulation. ■

The importance of Theorem 10.67 is manifold. First, it states that bisimulation equivalence preserves all quantitative PCTL*-definable properties. This justifies considering bisimulation-equivalent states to be "equal". Second, Theorem 10.67 asserts that bisimulation equivalence is the *coarsest* equivalence enjoying this property. That is to say, any strictly coarser equivalence identifies states with different probabilistic behavior in the sense of PCTL*-definable properties, and even some relatively simple properties that can be stated in PCTL$^-$. Moreover, Theorem 10.67 can be used to prove that two states are not bisimulation equivalent. In order to do so, it suffices to provide a PCTL* formula (or PCTL$^-$ or PCTL formula) that distinguishes the given states. Finally, observe that probabilistic bisimulation equivalence of states s_1 and s_2 in a Markov chain $\mathcal{M}$ implies bisimulation equivalence of s_1 and s_2 (as considered in Chapter 7) in the transition system associated with $\mathcal{M}$.

Example 10.68. Craps Game

Consider the probabilistic bisimulation quotient for the craps game, see Figure 10.12. It follows that $s_{4,10} \not\sim_{\mathcal{M}} s_{6,8}$, since there exists a PCTL formula, e.g., $\mathbb{P}_{<\frac{1}{6}}(\bigcirc \mathit{won})$, which holds in $s_{4,10}$ but not in $s_{6,8}$. ■

10.5 Markov Chains with Costs

In addition to the probability of certain events, it is natural to analyze the average behavior of executions in a Markov chain. For instance, for a communication system where a sender and a receiver can transfer messages via an unreliable channel, an interesting measure of interest is the expected number of attempts to send a message until correct delivery. Another example is a multiprocessor system where one might be interested in the average number of steps between two successive failures—the mean "time" between failures. For a battery-powered embedded system, a measure of interest is the expected power consumption during operation.

The aim of this section is to consider an extension of Markov chains, called *Markov reward* chains, and to consider expected measures. A Markov reward chain is a Markov chain in which states (or transitions) are augmented with rewards, natural numbers that can be interpreted as bonuses, or dually as costs. We consider equipping states with rewards. The idea is that whenever a state s is left, the reward associated with s is earned.

Definition 10.69. Markov Reward Model (MRM)

A *Markov reward model* (MRM) is a tuple $(\mathcal{M}, rew)$ with $\mathcal{M}$ a Markov chain with state space S and $rew : S \to \mathbb{N}$ a reward function that assigns to each state $s \in S$ a non-negative integer reward $rew(s)$. ■

Intuitively, the value $rew(s)$ stands for the reward earned on leaving state s. Formally, the *cumulative reward* for a finite path $\widehat{\pi} = s_0\, s_1 \ldots s_n$ is defined by

$$rew(\widehat{\pi}) \;=\; rew(s_0) + rew(s_1) + \ldots + rew(s_{n-1}).$$

Note that the reward of the last state s_n in the path $\widehat{\pi}$ is not considered.

Example 10.70. Zeroconf Protocol

Consider the Markov chain modeling the behavior of a single station in the zeroconf protocol 10.5 (page 751). For convenience, the Markov chain is depicted in Figure 10.13. We consider three reward functions for this model:

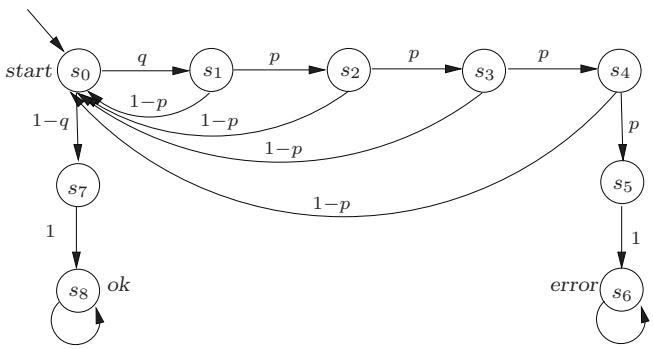


Figure 10.13: Markov chain of the IPv4 zeroconf protocol (for n=4 probes).

- The first reward assignment (denoted rew_1) represents waiting time—recall that on transmitting a probe, an acknowledgment is awaited for exactly r time units. It is defined by $rew_1(s_i) = r$ for $0 < i \leqslant n$, $rew_1(s_0) = 0$ assuming that the host randomly selects an address promptly, $rew_1(s_{n+3}) = n{\cdot}r$, $rew_1(s_{n+2}) = rew_1(s_{n+4}) = 0$, and

$rew_1(s_{n+1}) = E$, where E denotes some large number that represents the highly undesirable situation of an address collision.

- The second reward assignment (denoted rew_2) is used to keep track of the number of probes that are sent in total. It is defined by $rew_2(s_i) = 1$ for $0 < i \leqslant n$, $rew_2(s_{n+3}) = n$ and 0 otherwise.
- Finally, the third reward assignment (denoted rew_3) is used to keep track of the number of failed attempts to acquire an unused address. It is defined by $rew_3(s_1) = 1$ and 0 otherwise.

Consider the finite path $\widehat{\pi} = s_0\, s_1\, s_0\, s_1\, s_2\, s_0\, s_7\, s_8$. It follows that $rew_1(\widehat{\pi}) = 7r$, $rew_2(\widehat{\pi}) = 7$ and $rew_3(\widehat{\pi}) = 2$. ■

10.5.1 Cost-Bounded Reachability

Several quantitative properties of Markov reward models can be defined on the basis of cumulative rewards for finite paths. This section considers cost-bounded reachability, i.e., the expected reward before reaching a given set of states, and the probability of reaching these states within a given bound on the cumulative reward. Let $(\mathcal{M}, rew)$ be an MRM with state space S and $B \subseteq S$ the set of target states. For an infinite path $\pi = s_0\, s_1\, s_2 \ldots$ in $\mathcal{M}$ let

$$rew(\pi, \Diamond B) \;=\; \begin{cases} rew(s_0\, s_1 \ldots s_n) & \text{if } s_i \notin B \text{ for } 0 \leqslant i < n \text{ and } s_n \in B \\ \infty & \text{if } \pi \not\models \Diamond B. \end{cases}$$

Stated in words, $rew(\pi, \Diamond B)$ denotes the cumulative reward earned along an infinite path π until reaching a B-state for the first time. The expected reward until reaching B is now defined as the expectation of the function $rew(\pi, \Diamond B)$:

Definition 10.71. Expected Reward for Reachability Properties

For state s and $B \subseteq S$, the *expected reward* until reaching B from s is defined as follows. If $Pr(s \models \Diamond B) < 1$, then $ExpRew(s \models \Diamond B) = \infty$. Otherwise, i.e., if $Pr(s \models \Diamond B) = 1$, then:

$$ExpRew(s \models \Diamond B) \;=\; \sum_{r=0}^{\infty} r \cdot Pr_s\{\, \pi \in Paths(s) \mid \pi \models \Diamond B \wedge rew(\pi, \Diamond B) = r \,\}.$$

■

The infinite series for $Pr(s \models \Diamond B) = 1$ converges for any reward function *rew*. The intuitive argument for this is that the rewards along paths are added, while the probabilities of the transitions are multiplicated. The formal proof is left as an exercise to the reader; see Exercise 10.18 (page 904).

If $Pr(s \models \Diamond B) = 1$, then an equivalent characterization of the expected reward earned until B is reached can be provided by means of a weighted sum of the rewards earned along minimal path fragments from s to B:

$$ExpRew(s \models \Diamond B) \;=\; \sum_{\widehat{\pi}} \mathbf{P}(\widehat{\pi}) \cdot rew(\widehat{\pi})$$

where $\widehat{\pi}$ ranges over all finite paths $s_0 \ldots, s_n$ with $s_n \in B$, $s_0 = s$ and $s_0, \ldots, s_{n-1} \notin B$.

Example 10.72. Simulating a Die by a Coin (Revisited)

Consider again the Markov chain in Example 10.3 (page 750) which describes how a six-sided die can be simulated by a fair coin. For convenience, the Markov chain is illustrated again in Figure 10.14. In order to reason about the number of coin flips that are required

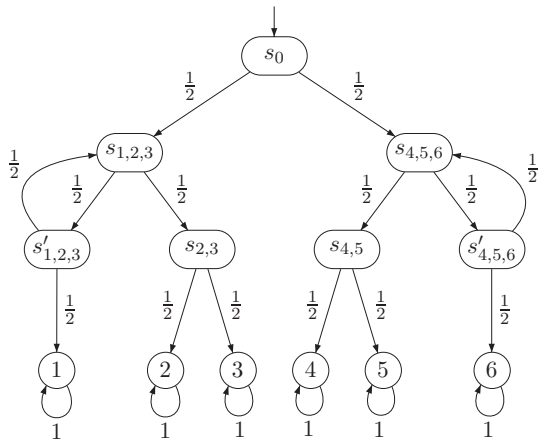


Figure 10.14: Markov chain for simulating a die by a fair coin.

to obtain a certain die outcome, the Markov chain is extended to an MRM. The set B of target states is thus $B = \{\, 1, 2, 3, 4, 5, 6 \,\}$. The reward function assigns reward 1 to all states except the states in B, and reward zero to the states in B. The cumulative reward of the infinite path $\pi = s_0\, s_{1,2,3}\, s'_{1,2,3}\, 1^\omega$ is $rew(\pi, \Diamond B) = 3$. The cumulative reward of $\pi' = s_0\, (s_{1,2,3}\, s'_{1,2,3})^\omega$ is ∞.

In order to determine the number of rounds that are performed until a die outcome is obtained, an alternative reward function is used. A round is a visit to either state $s_{1,2,3}$ or

$s_{4,5,6}$. To reason about the number of rounds, reward 1 is associated with the states $s_{1,2,3}$ and $s_{4,5,6}$. All other states are equipped with reward zero. Using this reward function, we obtain for the expected number of rounds before reaching a target state in B:

$$\mathit{ExpRew}(s_0 \models \Diamond B) \;=\; \sum_{r=1}^{\infty} r \cdot \frac{3}{4} \cdot \left(\frac{1}{4}\right)^{r-1} \;=\; \frac{4}{3} \quad .$$

Note that in every round a target state is reached (in this round) with probability $\frac{3}{4}$, while the probability for yet another round is $\frac{1}{4}$. Thus, the probability for exactly r rounds agrees with the probability of reaching B from the initial state s_0 while entering $\{ s_{1,2,3}, s_{3,4,5} \}$ exactly r times, i.e., $\frac{3}{4} \cdot (\frac{1}{4})^{r-1}$. The average number of rounds that is performed until an outcome is thus $\frac{4}{3}$.

Just a brief comment on the computation of $\sum_{r=1}^{\infty} r \cdot \frac{3}{4} \cdot \left(\frac{1}{4}\right)^{r-1}$. One possible way to compute the value of this infinite series is the observation that $\sum_{r=1}^{\infty} r \cdot (\frac{1}{4})^{r-1}$ can be viewed as the value of the first derivative of the function $f :]-1, 1[\to \mathbb{R}$:

$$f(x) \;=\; \sum_{r=0}^{\infty} x^r \;=\; \frac{1}{1-x}$$

for $x = \frac{1}{4}$. Then, $f'(x) \;=\; \sum\limits_{r=1}^{\infty} r \cdot x^{r-1} \;=\; \frac{1}{(1-x)^2}$. Hence,

$$\sum_{r=1}^{\infty} r \cdot \left(\tfrac{1}{4}\right)^{r-1} \;=\; f'(\tfrac{1}{4}) \;=\; \frac{1}{\left(1-\frac{1}{4}\right)^2} \;=\; \frac{1}{\left(\frac{3}{4}\right)^2}$$

Therefore: $\sum\limits_{r=1}^{\infty} r \cdot \frac{3}{4} \cdot \left(\frac{1}{4}\right)^{r-1} \;=\; \frac{3}{4} \cdot f'(\frac{1}{4}) \;=\; \frac{3}{4} \cdot \frac{1}{\left(\frac{3}{4}\right)^2} \;=\; \frac{4}{3}$. ■

Let us now discuss in detail how to compute the expected rewards $\mathit{ExpRew}(s \models \Diamond B)$, for finite MRMs. Using the graph-based techniques explained in Section 10.1.2 (page 770ff), we first determine the set $S_{=1}$ of states s that reach B almost surely, i.e.:

$$S_{=1} \;=\; \{\, s \in S \mid \mathit{Pr}(s \models \Diamond B) = 1 \,\}.$$

The task is to compute $x_s = \mathit{ExpRew}(s \models \Diamond B)$ for each state $s \in S_{=1}$. For all states $s \in S_{=1} \setminus B$, i.e., $s \in S \setminus B$ for which $\mathit{Pr}(s \models \Diamond B) = 1$, we have $\mathit{Pr}(u \models \Diamond B) = 1$ for all direct successors u of s. Hence, if $s \in S_{=1}$, then either $s \in B$ or $\mathit{Post}(s) \subseteq S_{=1}$. Moreover, the values $x_s = \mathit{ExpRew}(s \models \Diamond B)$ provide a solution of the following equation system:

$$x_s \;=\; \begin{cases} 0 & \text{if } s \in B \\ \mathit{rew}(s) + \sum\limits_{u \,\in\, \mathit{Post}(s)} \mathbf{P}(s,u) \cdot x_u & \text{if } s \in S_{=1} \setminus B. \end{cases}$$

In fact, the vector $(x_s)_{s\in S_{=1}}$ is the unique solution of the above linear equation system. This follows from the results established in the proof of Theorem 10.19 on page 766 for the subchain consisting of the states in $S_{=1}$. Notice that the proof of Theorem 10.19 yields that

$$\mathbf{A}\mathbf{x} = \mathbf{x} \quad \text{implies} \quad \mathbf{x} = \mathbf{0}$$

where $\mathbf{A} = (\,\mathbf{P}(s,u)\,)_{s,u\in S_{=1}\setminus B}$. Since $\mathbf{A}$ is a quadratic matrix, this implication is equivalent to the nonsingularity of $\mathbf{I} - \mathbf{A}$. That is, for any vector $\mathbf{b}$, the linear equation system $\mathbf{x} = \mathbf{A}\mathbf{x} + \mathbf{b}$, which can be rewritten as $(\mathbf{I} - \mathbf{A})\mathbf{x} = \mathbf{b}$, has a unique solution, namely $\mathbf{x} = (\mathbf{I} - \mathbf{A})^{-1}\mathbf{b}$. In fact, the above equation system for the expected rewards can be rewritten in the form $\mathbf{x} = \mathbf{A}\mathbf{x} + \mathbf{b}$ where $\mathbf{x}$ stands for the vector $(\,x_s\,)_{s\in S_{=1}\setminus B}$ and $\mathbf{b} = (\,b_s\,)_{s\in S_{=1}\setminus B}$ is the vector with the entries $b_s = \mathit{rew}(s)$.

Summarizing, computing the expected reward earned until a certain set of states will be reached in a finite MRM has a polynomial time complexity of the size of $\mathcal{M}$. The necessary techniques are a graph analysis to determine $S_{=1}$, and solving a linear equation system.

Example 10.73. Simulating a Die by a Coin (Revisited)

Consider again Knuth and Yao's example of simulating a die by a fair coin. As in Example 10.72, consider the MRM where rewards serve as counters for the number of rounds. That is, $\mathit{rew}(s_{1,2,3}) = \mathit{rew}(s_{4,5,6}) = 1$, and $\mathit{rew}(s) = 0$ for all other states. Let us compute the average number of rounds performed until an outcome is obtained, i.e., until a state in $B = \{\,1,2,3,4,5,6\,\}$ is reached, by the above linear equation system. Clearly, $\mathit{Pr}(s \models \Diamond B) = 1$ for each state s. Hence, all states are contained in $S_{=1}$. Let $x_{1,2,3}$ denote $x_{s_{1,2,3}}$ and $x'_{1,2,3}$ denote $x_{s'_{1,2,3}}$. The values $x_s = \mathit{ExpRew}(s \models \Diamond B)$ are a solution of the following equation system:

$$\begin{aligned}
x_0 &= \tfrac{1}{2}\cdot x_{1,2,3} + \tfrac{1}{2}\cdot x_{4,5,6}\\
x_{1,2,3} &= 1 + \tfrac{1}{2}\cdot x'_{1,2,3} + \tfrac{1}{2}\cdot x_{2,3}\\
x_{2,3} &= \tfrac{1}{2}\cdot x_2 + \tfrac{1}{2}\cdot x_3\\
x'_{1,2,3} &= \tfrac{1}{2}\cdot x_{1,2,3} + \tfrac{1}{2}\cdot x_1\\
x_{4,5,6} &= 1 + \tfrac{1}{2}\cdot x'_{4,5,6} + \tfrac{1}{2}\cdot x_{4,5}\\
x_{4,5} &= \tfrac{1}{2}\cdot x_4 + \tfrac{1}{2}\cdot x_5\\
x'_{4,5,6} &= \tfrac{1}{2}\cdot x_{4,5,6} + \tfrac{1}{2}\cdot x_6\\
x_1 &= x_2 = x_3 = x_4 = x_5 = x_6 = 0
\end{aligned}$$

Using the values $x_i = 0$ for $0 < i \leqslant 6$, we obtain $x_{2,3} = x_{4,5} = 0$. The equations for the

other states simplify to

$$\begin{aligned} x_0 &= \tfrac{1}{2} \cdot x_{1,2,3} + \tfrac{1}{2} \cdot x_{4,5,6} \\ x_{1,2,3} &= 1 + \tfrac{1}{2} \cdot x'_{1,2,3} \\ x'_{1,2,3} &= \tfrac{1}{2} \cdot x_{1,2,3} \\ x_{4,5,6} &= 1 + \tfrac{1}{2} \cdot x'_{4,5,6} \\ x'_{4,5,6} &= \tfrac{1}{2} \cdot x_{4,5,6} \end{aligned}$$

This yields the linear equation system:

$$\begin{pmatrix} 1 & -\frac{1}{2} & 0 & -\frac{1}{2} & 0 \\ 0 & 1 & -\frac{1}{2} & 0 & 0 \\ 0 & -\frac{1}{2} & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -\frac{1}{2} \\ 0 & 0 & 0 & -\frac{1}{2} & 1 \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_{1,2,3} \\ x'_{1,2,3} \\ x_{4,5,6} \\ x'_{4,5,6} \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

The unique solution of this linear equation system is $x_0 = x_{1,2,3} = x_{4,5,6} = \frac{4}{3}$ and $x'_{1,2,3} = x'_{4,5,6,} = \frac{2}{3}$. Thus, we obtain—as before—$\mathit{ExpRew}(x_0 \models \Diamond B) = x_0 = \frac{4}{3}$ for the average number of rounds. ■

Remark 10.74. Alternative Definitions of the Expected Reward

According to Definition 10.71 (page 818), $\mathit{ExpRew}(s \models \Diamond B) = \infty$ if B is not almost surely reached from s. This choice is consistent with the fact that $\mathit{rew}(\pi, \Diamond B) = \infty$ whenever $\pi \not\models \Diamond B$ for some path π. Let us discuss two variants of this definition.

For certain applications, it might be more reasonable to define the cumulative reward for an infinite path fragment which never visits B as zero (rather than ∞). In this case, the expected reward $\mathit{ExpRew}(s \models \Diamond B)$ is defined as

$$\sum_{r=0}^{\infty} r \cdot \mathit{Pr}_s\{\, \pi \in \mathit{Paths}(s) \mid \pi \models \Diamond B \wedge \mathit{rew}(\pi, \Diamond B) = r \,\}$$

for all cases. (That is, no exceptional case for $\mathit{Pr}(s \models \Diamond B) < 1$.) The proposed algorithm for computing $\mathit{ExpRew}(s \models \Diamond B)$ can easily be adapted to deal with this adapted definition by adding the constraint $x_s = 0$ for $\mathit{Pr}(s \models \Diamond B) = 0$ to the linear equation system. The equation $x_t = 0$ for $t \in B$ remains unchanged. For all other states s, i.e., all states s where $s \notin B$ and $\mathit{Pr}(s \models \Diamond B) > 0$, we deal with the equation:

$$x_s = \mathit{rew}(s) + \sum_{u \,\in\, \mathit{Post}(s)} \mathbf{P}(s,u) \cdot x_u.$$

Another way of treating the paths that never reach a B-state is to consider *conditional expectations*. That is, we consider the expected reward to reach B under the condition

that B is eventually reached. The corresponding definition of this conditional expectation, denoted $CExpRew(s \models \Diamond B)$, is as follows. If $Pr(s \models \Diamond B) > 0$, then

$$CExpRew(s \models \Diamond B) \;=\; \sum_{r=0}^{\infty} r \cdot \frac{Pr_s\{\, \pi \in Paths(s) \mid \pi \models \Diamond B \;\wedge\; rew(\pi, \Diamond B) = r \,\}}{Pr(s \models \Diamond B)}.$$

If $Pr(s \models \Diamond B) = 0$, i.e., B is not reachable from s, then $CExpRew(s \models \Diamond B)$ can be defined as undefined or some fixed value. The computation of the conditional expected rewards goes as follows. By means of a backward graph analysis starting from the B-states, the set

$$Pre^*(B) \;=\; \{\, s \in S \mid Pr(s \models \Diamond B) > 0 \,\}.$$

is determined. This is a standard backward reachability analysis. Let $\mathcal{M}'$ be a new Markov chain with state space $Pre^*(B)$. The transition probabilities of $\mathcal{M}'$ are defined by

$$\mathbf{P}'(s, s') \;=\; \frac{\mathbf{P}(s, s')}{\mathbf{P}(s, Pre^*(B))} \text{ if } s \in Pre^*(B) \setminus B \text{ and } s' \in Pre^*(B).$$

For $t \in B$ let $\mathbf{P}'(t, t) = 1$ and $\mathbf{P}'(t, s') = 0$ for all states $s' \in Pre^*(B) \setminus \{\, t \,\}$. Then $Pr^{\mathcal{M}'}(s \models \Diamond B) = 1$ for all s in $\mathcal{M}'$ and

$$CExpRew^{\mathcal{M}}(s \models \Diamond B) \;=\; ExpRew^{\mathcal{M}'}(s \models \Diamond B).$$

The conditional expected reward in $\mathcal{M}$ is thus equal to the expected reward in $\mathcal{M}'$. Thus, the problem of computing the conditional expected rewards $CExpRew(s \models \Diamond B)$ is reducible to the problem of computing (unconditional) expected rewards. ■

Other important quantitative measures for MRMs are *cost-bounded reachability probabilities*, i.e., the probability of reaching a certain given set of states within a certain bound on the cumulative reward. Paths that reach one of the target states whose cumulative reward exceeds the bound are considered too costly. Let $B \subseteq S$, $r \in \mathbb{N}$ and $\Diamond_{\leqslant r} B$ denote the event of reaching the set B with cumulative reward at most r. Then:

$$Pr(s \models \Diamond_{\leqslant r} B) \;=\; Pr_s\{\, \pi \in Paths(s) \mid \pi \models \Diamond B \;\wedge\; rew(\pi, \Diamond B) \leqslant r \,\}$$

denotes the probability of reaching B from s while the cumulative reward earned until B is reached is at most r. For example, when B stands for the set of good states, $Pr(s \models \Diamond_{\leqslant r} B)$ denotes the probability of reaching a good state with cost at most r. We have

$$Pr(s \models \Diamond_{\leqslant r} B) \;=\; \sum_{\widehat{\pi}} \mathbf{P}(\widehat{\pi})$$

where $\widehat{\pi}$ ranges over all finite paths $s_0 \ldots s_n$ with $s_n \in B$, $s_0 = s$ and $s_0, \ldots, s_{n-1} \notin B$ such that $rew(\widehat{\pi}) \leqslant r$.

For finite MRMs, the values $x_{s,r} = Pr(s \models \Diamond_{\leqslant r} B)$ can be computed via the following equation system:

- if $s \in B$, then $x_{s,r} = 1$;
- if $s \notin \mathit{Pre}^*(B)$ or $s \in \mathit{Pre}^*(B) \setminus B \wedge (\mathit{rew}(s) > r)$; then $x_{s,r} = 0$
- in all other cases, i.e., if $s \in \mathit{Pre}^*(B) \setminus B$ and $\mathit{rew}(s) \leqslant r$:

$$x_{s,r} \;=\; \sum_{u \in S} \mathbf{P}(s,u) \cdot x_{u,\, r \,-\, \mathit{rew}(s)}.$$

The above equation system can be regarded as a linear equation system with variables $x_{s,\rho}$ where $(s,\rho) \in S \times \{0,1,\ldots,r\}$. If all rewards are positive, i.e., $\mathit{rew}(s) > 0$ for any state s, then $x_{s,r}$ is completely determined by the values $x_{u,\rho}$ for $u \in S$ and $\rho < r$. Hence, in this case the above equation system can be solved by successively computing the vectors $(x_{s,\rho})_{s \in S}$ for $\rho = 0,1,\ldots,r$. In the presence of zero-reward states, the above equation for $x_{s,r}$ may contain some variables $x_{u,r}$ for the same reward bound r. However, $x_{s,r}$ only depends on the values $x_{u,\rho}$ for $\rho \leqslant r$. Thus, the vectors $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_r$ can be determined in this order, where the vector $\mathbf{x}_\rho = (x_{s,\rho})_{s \in S}$ is obtained from $\mathbf{x}_i$ for $0 \leqslant i < \rho$ as follows. If $s \in B$ or $s \notin \mathit{Pre}^*(B)$, the value of $x_{s,\rho}$ is 1 or 0 (see the first two items above). For $s \in \mathit{Pre}^*(B) \setminus B$ and $\mathit{rew}(s) > 0$, the value of $x_{s,\rho}$ is obtained from the previously computed values $x_{u,\, \rho \,-\, \mathit{rew}(s)}$ (see the sum in the third item). For the remaining states, i.e., states in $S_0 = \{\, s \in \mathit{Pre}^*(B) \setminus B \mid \mathit{rew}(s) = 0 \,\}$, the values $x_{s,\rho}$ are obtained as the unique solution of the following linear equation system:

$$\mathbf{x} \;=\; \mathbf{A}_0 \cdot \mathbf{x} \,+\, \mathbf{b}.$$

Vector $\mathbf{x}$ stands for $(\, x_{s,\rho} \,)_{s \in S_0}$. The matrix $\mathbf{A}_0$ contains the transition probabilities between the states in S_0, i.e., $\mathbf{A}_0 = (\, \mathbf{P}(s,u) \,)_{s,u \in S_0}$. The entries of the vector $\mathbf{b} = (\, b_{s,\rho} \,)_{s \in S_0}$ are given by

$$\begin{aligned}
b_{s,\rho} \;&=\; \sum_{u \in B} \mathbf{P}(s,u) \cdot \overbrace{x_{u,\rho}}^{=1} \;+\; \sum_{u \,\in\, S \,\setminus\, \mathit{Pre}^*(B)} \mathbf{P}(s,u) \cdot \overbrace{x_{u,\rho}}^{=0} \\
&\qquad +\; \sum_{\substack{u \,\in\, \mathit{Pre}^*(B) \,\setminus\, B \\ \mathit{rew}(u) \,>\, \rho}} \mathbf{P}(s,u) \cdot \overbrace{x_{u,\rho}}^{=0} \;+\; \sum_{\substack{u \,\in\, \mathit{Pre}^*(B) \,\setminus\, B \\ \rho \,\geqslant\, \mathit{rew}(u) \,>\, 0}} \mathbf{P}(s,u) \cdot x_{u,\rho} \\
&=\; \mathbf{P}(s,B) \;+\; \sum_{\substack{u \,\in\, \mathit{Pre}^*(B) \,\setminus\, B \\ \rho \,\geqslant\, \mathit{rew}(u) \,>\, 0}} \mathbf{P}(s,u) \cdot \underbrace{x_{u,\rho}}_{\substack{\text{already} \\ \text{computed}}}
\end{aligned}$$

The above equation system is linear and has a unique solution. Recall that $\mathbf{P}(s,B)$ stands for the probability of moving from state s within one transition to a state $t \in B$, i.e., $\mathbf{P}(s,B) = \sum_{t \in B} \mathbf{P}(s,t)$.

Summarizing, the probabilities $Pr(s \models \Diamond_{\leqslant r} B)$ can be computed with a time complexity which is polynomial in the size of $\mathcal{M}$ and linear in the reward bound r. To compute $Pr(s_0 \models \Diamond_{\leqslant r} B)$ for some designated state s_0, in fact not all variables $x_{s,\rho}$ might be relevant. The relevant variables are obtained by starting with the equation for $x_{s_0,r}$ and then adding only those equations $x_{s,\rho}$ that appeared in an already generated equation. This observation has no influence on the worst-case time complexity, but may yield drastic speedups.

Example 10.75. Simulation of a Die by a Coin (Revisited)

Consider again the MRM for the simulation of a die by a fair coin where the reward function serves to count the number of rounds; see Examples 10.72 (page 819) and 10.73 (page 821). Consider the probability of obtaining a die outcome within the first two rounds. This amounts to computing the values $x_{s,\rho} = Pr(s \models \Diamond_{\leqslant \rho} B)$ for $\rho = 0, 1, 2$ and all states s, where $B = \{\, 1, 2, 3, 4, 5, 6 \,\}$. We have

$$\begin{aligned} & x_{s_0,0} = x_{s_{1,2,3},0} = x_{s_{4,5,6},0} = 0, \\ & x_{s'_{1,2,3},0} = x_{s'_{4,5,6},0} = \tfrac{1}{2}, \\ & x_{s_{2,3},0} = x_{s_{4,5},0} = x_i = 1, \quad i = 1, \ldots, 6. \end{aligned}$$

The probability of reaching a B-state from the initial state in the first round is given by $x_{s_0,1}$ where

$$\begin{aligned} x_{s_0,1} &= \tfrac{1}{2} \cdot x_{s_{1,2,3},1} + \tfrac{1}{2} \cdot x_{s_{4,5,6},1} \\ x_{s_{1,2,3},1} &= \tfrac{1}{2} \cdot \underbrace{x_{s_{2,3},0}}_{=1} + \tfrac{1}{2} \cdot \underbrace{x_{s'_{1,2,3},0}}_{=\frac{1}{2}} = \tfrac{3}{4} \\ x_{s_{4,5,6},1} &= \tfrac{1}{2} \cdot \underbrace{x_{s_{4,5},0}}_{=1} + \tfrac{1}{2} \cdot \underbrace{x_{s'_{4,5,6},0}}_{=\frac{1}{2}} = \tfrac{3}{4}. \end{aligned}$$

This yields $x_{s_0,1} = \frac{3}{4}$. For the other states, we have $x_{s_{2,3},1} = x_{s_{4,5},1} = x_i = 1$ for $1 \leqslant i \leqslant 6$ and

$$x_{s'_{1,2,3},1} = \tfrac{1}{2} \cdot x_{s_{1,2,3},1} + \tfrac{1}{2} \cdot x_{s_{2,3},1} = \tfrac{1}{2} \cdot \tfrac{3}{4} + \tfrac{1}{2} \cdot 1 = \tfrac{7}{8},$$

and similarly $x_{s'_{4,5,6},1} = \frac{7}{8}$. The probability $x_{s_0,2}$ of reaching B from the initial state within two rounds is given by

$$\begin{aligned} x_{s_0,2} &= \tfrac{1}{2} \cdot x_{s_{1,2,3},2} + \tfrac{1}{2} \cdot x_{s_{4,5,6},2} \\ x_{s_{1,2,3},2} &= \tfrac{1}{2} \cdot \underbrace{x_{s_{2,3},1}}_{=1} + \tfrac{1}{2} \cdot \underbrace{x_{s'_{1,2,3},1}}_{=\frac{7}{8}} = \tfrac{15}{16} \\ x_{s_{4,5,6},1} &= \tfrac{1}{2} \cdot \underbrace{x_{s_{4,5},1}}_{=1} + \tfrac{1}{2} \cdot \underbrace{x_{s'_{4,5,6},1}}_{=\frac{7}{8}} = \tfrac{15}{16}. \end{aligned}$$

Thus, $x_{s_0,2} = \frac{15}{16}$. In fact, we have $x_{s_0,r} = 1 - \frac{1}{4^r}$ for all integers $r \geqslant 0$; see Exercise 10.19 (page 904). ■

The techniques discussed so far for cost-bounded reachability probabilities can be adapted to cost-bounded *constrained* reachability probabilities in a fairly straightforward manner. For $C, B \subseteq S$, let

$$Pr(s \models C \,\mathsf{U}_{\leqslant r} B) \;=\; Pr_s\{\, \pi \in Paths(s) \mid \pi \models C \,\mathsf{U}_{\leqslant r} B \,\}$$

where $s_0\, s_1\, s_2 \ldots \models C \,\mathsf{U}_{\leqslant r} B$ if and only if there exists a finite prefix $s_0 \ldots s_n$ such that $s_i \in C$ for $0 \leqslant i < n$, $s_n \in B$ and $rew(s_0\, s_1 \ldots s_n) \leqslant r$. In fact, the values $Pr(s \models C \,\mathsf{U}_{\leqslant r} B)$ are obtained by an analogous linear equation system as for $Pr(s \models \Diamond_{\leqslant r} B)$. The equations are

- if $s \in B$, then $x_{s,r} = 1$;
- if $s \not\models \exists(C \,\mathsf{U}\, B)$ or $s \notin B \wedge rew(s) > r$, then $x_{s,r} = 0$;
- in all other cases, i.e., if $s \in \exists(C \,\mathsf{U}\, B)$ and $rew(s) \leqslant r$:

$$x_{s,r} \;=\; \sum_{u \in S} \mathbf{P}(s,u) \cdot x_{u,\, r - rew(s).}$$

The techniques discussed so far are key ingredients for the model-checking of a variant of PCTL that incorporates rewards. This logic is called PRCTL, which is short for Probabilistic Reward CTL. It is defined as follows.

Definition 10.76. Syntax of PRCTL

PRCTL *state formulae* over the set AP of atomic propositions are formed according to the following grammar:

$$\Phi ::= \text{true} \;\Big|\; a \;\Big|\; \Phi_1 \wedge \Phi_2 \;\Big|\; \neg\Phi \;\Big|\; \mathbb{P}_J(\varphi) \;\Big|\; \mathbb{E}_R(\Phi)$$

where $a \in AP$, φ is a path formula, $J \subseteq [0,1]$ and R are intervals with rational bounds. PCTL *path formulae* are formed according to the following grammar:

$$\varphi ::= \bigcirc \Phi \;\Big|\; \Phi_1 \,\mathsf{U}\, \Phi_2 \;\Big|\; \underbrace{\Phi_1 \,\mathsf{U}^{\leqslant n} \Phi_2}_{\substack{\text{step-bounded} \\ \text{until}}} \;\Big|\; \underbrace{\Phi_1 \,\mathsf{U}_{\leqslant r} \Phi_2}_{\substack{\text{reward-bounded} \\ \text{until}}}$$

where Φ, Φ_1, and Φ_2 are state formulae and $n, r \in \mathbb{N}$. ■

The semantics of the propositional logic fragment and probability operator is defined as for PCTL. For the expectation operator $\mathbb{E}_R(\cdot)$, the semantics is defined by:

$$s \models \mathbb{E}_R(\Phi) \quad \text{iff} \quad \mathit{ExpRew}(s \models \Diamond \mathit{Sat}(\Phi)) \in R.$$

Example 10.77.

The logic PRCTL may be used to specify the simulation of a die by a coin by the requirements

$$\mathbb{E}_{\leqslant \frac{3}{2}}(\mathit{outcome}) \ \wedge \ \bigwedge_{1 \leqslant i \leqslant 6} \mathbb{P}_{=\frac{1}{6}}(\Diamond i) \ \wedge \ \mathbb{P}_{\geqslant \frac{15}{16}}(\Diamond_{\leqslant 2}\, \mathit{outcome})$$

where *outcome* is an atomic proposition that labels the six outcome states 1,2,3,4,5, and 6. The above PRCTL formula asserts that (1) the average number of rounds to obtain an outcome is bounded by $\frac{3}{2}$, (2) the correctness of the outcomes in the sense that each of the six possible outcomes has probability $\frac{1}{6}$, and (3) with probability at least $\frac{15}{16}$ the outcome is obtained within at most two rounds. ■

Example 10.78. Zeroconf Protocol

Consider again the zeroconf protocol as introduced in Example 10.5 (page 751). Consider the reward function that represents waiting times. The property "the probability of ending up with an unused address within n steps exceeds p'" can be expressed as PRCTL formula

$$\mathbb{P}_{>p'}(\Diamond_{\leqslant n}\, \mathit{ok})$$

where *ok* uniquely indicates the state in which an unused address has been selected.

Consider now the reward function that keeps track of the number of probes sent. The property "the probability of ending up with an unused address after at most n probes exceeds p'" is expressed by the PRCTL formula

$$\mathbb{P}_{>p'}(\Diamond_{\leqslant n}\, \mathit{ok}).$$

Although the formula is identical to the previous one, its interpretation is somewhat different due to the different reward function. ■

10.5.2 Long-Run Properties

This section is concerned with a different class of properties, viz. *long-run averages*. As opposed to the measures considered in the previous section, long-run averages are based

on the limiting behavior of Markov chains. Before considering long-run averages, let us first consider the long-run distribution of a Markov chain.

The long-run distribution is a limit defined on the basis of transient distributions. Recall that the transient state distribution $\Theta_n = \Theta_n^{\mathcal{M}}$ is a function that assigns to each state $t \in S$ the probability of being in state t after exactly n steps given the initial distribution ι_{init}; see Remark 10.22 (page 768). The limiting behavior of $\mathcal{M}$ is obtained when n tends to go to infinity.

For a given Markov chain $\mathcal{M}$ and states s, t in $\mathcal{M}$, let $\theta_n^{\mathcal{M}}(s,t)$ (briefly $\theta_n(s,t)$) denote the probability of being in state t after exactly n steps when starting in state s, i.e.:

$$\theta_n(s,t) \;=\; Pr_s\{\; s_0\, s_1\, s_2 \ldots \in Paths(s) \mid s_0 = s \;\wedge\; s_n = t \,\}.$$

The transient state distribution Θ_n arises by the values $\theta_n(s,t)$ by taking the initial distribution into account:

$$\Theta_n(t) \;=\; \sum_{s \in S} \iota_{\text{init}}(s) \cdot \theta_n(s,t).$$

For fixed state s, the function $t \mapsto \theta_n(s,t)$ agrees with the transient state distribution $\Theta_n^{\mathcal{M}_s}$ in the Markov chain $\mathcal{M}_s$ that is obtained from $\mathcal{M}$ by declaring s to be the unique starting state.

Definition 10.79. Long-Run Distributions

Let $\mathcal{M} = (S, \mathbf{P}, \iota_{\text{init}}, AP, L)$ be a finite Markov chain and s, $t \in S$. The *long-run average probability* $\theta^{\mathcal{M}}(s,t)$ (briefly $\theta(s,t)$) is given by

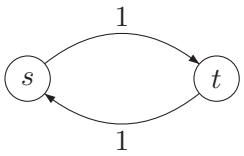
$$\theta(s,t) \;=\; \lim_{n \to \infty} \frac{1}{n} \cdot \sum_{i=1}^{n} \theta_i(s,t).$$

The function $t \mapsto \theta(s,t)$ is the *long-run distribution* for starting state s. The long-run distribution for $\mathcal{M}$ is given by

$$\Theta^{\mathcal{M}}(t) = \sum_{s \in S} \iota_{\text{init}}(s) \cdot \theta(s,t).$$

■

The limit in the above definition exists in every finite Markov chain. We will not provide a formal proof of this fact and refer to textbooks on Markov chains, e.g., [248]. Intuitively, the long-run probability $\theta(s,t)$ is the fraction of (discrete) time to be in state t when starting in state s. For example, in the following two-state Markov chain:



When starting in s, then after any even number of steps the current state is s, while after any odd number of steps the current state is t. Hence:

$$\theta_{2k}(s,s) \;=\; \theta_{2k+1}(s,t) \;=\; 1 \quad \text{for all } k \geqslant 0$$

while $\theta_{2k}(s,t) = \theta_{2k+1}(s,s) = 0$. Therefore:

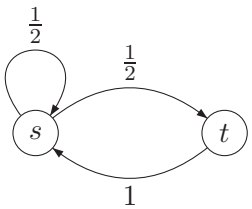
$$\tfrac{1}{2k}\sum_{i=1}^{2k}\theta_i(s,t) \;=\; \frac{k}{2k}, \qquad \tfrac{1}{2k+1}\sum_{i=1}^{2k+1}\theta_i(s,t) \;=\; \frac{k+1}{2k+1}.$$

Hence, $\lim\limits_{n\to\infty}\theta_n(s,t)$ does not exist, but

$$\theta(s,t) \;=\; \lim_{n\to\infty}\frac{1}{n}\sum_{i=1}^{n}\theta_i(s,t) \;=\; \frac{1}{2}.$$

This example illustrates that the sequence of the transient probabilities $\theta_n(s,t)$ may not converge. Hence, the definition of long-run probabilities cannot be simplified by just taking the limit of the transient probabilities $\theta_n(s,t)$.

However, if $\lim_{n\to\infty}\theta_n(s,t)$ exists, then it agrees with $\theta(s,t)$. This is illustrated by the Markov chain:



Whenever the current state is s, then both successors s and t are equally likely. If, however, the current state is t, then almost surely the next state is s. This provides an intuitive explanation that the fraction of the time being in state s is $\frac{2}{3}$, and for state t is $\frac{1}{3}$. For this example, we have

$$\theta_{n+1}(s,s) \;=\; \frac{1}{2}\theta_n(s,s) \;+\; \theta_n(s,t) \quad \text{and} \quad \theta_{n+1}(s,t) \;=\; \frac{1}{2}\theta_n(s,s).$$

The following table shows the values for $0 \leqslant n \leqslant 4$:

n	$\theta_n(s,s)$	$\theta_n(s,t)$
0	1	0
1	$\frac{3}{4}$	$\frac{1}{4}$
2	$\frac{5}{8}$	$\frac{3}{8}$
3	$\frac{11}{16}$	$\frac{5}{16}$
4	$\frac{21}{32}$	$\frac{11}{31}$
$\vdots$	$\vdots$	$\vdots$
limit	$\frac{2}{3}$	$\frac{1}{3}$

In fact, $\lim_{n\to\infty} \theta_n(s,s) = \frac{2}{3} = \theta(s,s)$ and $\lim_{n\to\infty} \theta_n(s,t) = \frac{1}{3} = \theta(s,t)$.

Long-run probabilities provide the basis for several interesting measures. The following definition considers the expected long-run reward that is earned between two successive visits to a state in the set B.

Definition 10.80. Expected Long-run Reward between B-States

Let $\mathcal{M}$ be a finite MRM with state space S, $B \subseteq S$ and $s \in S$ such that $Pr(s \models \Box\Diamond B) = 1$. The *expected long-run reward* between two B-states for s is given by

$$\mathit{LongRunER}_s(B) \;=\; \sum_{t\in B} \theta(s,t) \cdot \mathit{ExpRew}(t \models \bigcirc \Diamond B)$$

where $\mathit{ExpRew}(t \models \bigcirc \Diamond B)$ is defined as $\mathit{ExpRew}(t \models \Diamond B)$, except that only paths of length $\geqslant 1$ are considered, i.e.,

$$\mathit{ExpRew}(t \models \bigcirc \Diamond B) \;=\; \mathit{rew}(t) + \sum_{u\in S} \mathbf{P}(t,u) \cdot \mathit{ExpRew}(u \models \Diamond B).$$

■

The value $\mathit{ExpRew}(t \models \bigcirc \Diamond B)$ can be interpreted as the average reward earned by moving from t to B within one or more steps. The intuitive meaning of $\mathit{LongRunER}_s(B)$ is the average reward earned between two successive visits to a B-state, when considering the Markov chain $\mathcal{M}$ on the long run. For example, if B characterizes certain failure states, and the reward function is defined to be one for all states outside B, then $\mathit{LongRunER}_s(B)$ yields the average number of steps that are performed between two failures.

The computation of the expected long-run rewards $\mathit{LongRunER}_s(B)$ relies on techniques for computing the long-run probabilities. We will discuss now how the values $\theta(s,t)$ can be obtained—once again—by solving linear equation systems.

In the sequel, let $\mathcal{M}$ be a finite Markov chain with state space S and $s \in S$. Let us first establish that the function $t \mapsto \theta(s,t)$ is, in fact, a distribution over S. Indeed, we have $0 \leqslant \theta(s,t) \leqslant 1$, since $0 \leqslant \theta_i(s,t) \leqslant 1$ for all i and therefore $\sum_{1 \leqslant i \leqslant n} \theta_i(s,t) \leqslant n$. Moreover, for each $i \geqslant 1$:

$$\sum_{t \in S} \theta_i(s,t) \;=\; 1$$

Therefore:

$$\sum_{t \in S} \theta(s,t) \;=\; \frac{1}{n} \sum_{t \in S} \sum_{i=1}^{n} \theta_i(s,t) \;=\; \frac{1}{n} \sum_{i=1}^{n} \underbrace{\sum_{t \in S} \theta_i(s,t)}_{=1} \;=\; \frac{1}{n} \cdot n \;=\; 1$$

Furthermore, we have the following *balance equation*:

$$\theta(s,u) \;=\; \sum_{t \in S} \theta(s,t) \cdot \mathbf{P}(t,u).$$

This follows from

$$\begin{aligned}
& \sum_{t \in S} \theta(s,t) \cdot \mathbf{P}(t,u) \\
= \;& \lim_{n \to \infty} \tfrac{1}{n} \sum_{t \in S} \sum_{i=1}^{n} \theta_i(s,t) \cdot \mathbf{P}(t,u) \\
= \;& \lim_{n \to \infty} \tfrac{1}{n} \sum_{i=1}^{n} \sum_{t \in S} \theta_i(s,t) \cdot \mathbf{P}(t,u) \\
= \;& \lim_{n \to \infty} \tfrac{1}{n} \sum_{i=1}^{n} \theta_{i+1}(s,u) \\
= \;& \lim_{n \to \infty} \underbrace{\frac{n+1}{n}}_{\text{tends to } 1} \cdot \Big(\underbrace{\frac{1}{n+1} \cdot \sum_{j=1}^{n+1} \theta_j(s,u)}_{\text{tends to } \theta(s,u)} - \underbrace{\frac{\theta_1(s,u)}{n+1}}_{\text{tends to } 0} \Big) \;=\; \theta(s,u).
\end{aligned}$$

As almost surely a BSCC will be reached, $\theta(s,t) = 0$ if t does not belong to a BSCC that is reachable from s. Note that then $\theta_i(s,t)$ tends to zero if i tends to infinity. In particular, if $s \in T$ and $T \in \mathit{BSCC}(\mathcal{M})$, then:

$$\sum_{t \in T} \theta(s,t) \;=\; 1.$$

This and the balance equation (applied to the sub-Markov chain consisting of the states in B) yields that the values $x_{s,t} = \theta(s,t)$ for s, $t \in T$ and $T \in \mathit{BSCC}(\mathcal{M})$ yield a solution

of the equation system:

$$\sum_{t \in T} x_{s,t} \;=\; 1 \quad \text{and} \quad x_{s,u} \;=\; \sum_{t \in T} x_{s,t} \cdot \mathbf{P}(t,u)$$

where s and u range over all states of T. In fact, the above equation system has a unique solution. In particular:

$$\theta(s,t) \;=\; x_{s,t} \;=\; x_{s',t} \;= \theta(s',t)$$

for all states s, s', t of a BSCC. Thus, the long-run distributions $t \mapsto \theta(s,t)$ for the states s that are contained in some BSCC T can be solved by means of the linear equation system with variables x_t for $t \in T$ and the equations

- $\sum\limits_{t \in T} x_t \;=\; 1$;
- $\sum\limits_{t \in T} x_t \cdot \mathbf{P}(t,u) \;=\; x_u$ for all states $u \in T$.

This equation system has a unique solution $(x_t)_{t \in T}$ and $x_t = \theta(s,t)$ for all states s, $t \in T$. Furthermore, $\theta(s,u) = 0$ for all states $s \in T$ and $u \notin T$. For all states s which do not belong to a BSCC, the long-run probabilities $\theta(s,\cdot)$ are obtained as follows:

- $\theta(s,t) = 0$ if t is not contained in BSCC, and
- $\theta(s,t) = Pr(s \models \Diamond T) \cdot x_t$ if $t \in T$ and $T \in \mathit{BSCC}(\mathcal{M})$, where $x_t = \theta(s,t)$ for all/some $s \in T$.

A corresponding operator to specify lower or upper bounds for the expected long-run rewards, say $\mathbb{L}_R(\Phi)$ where R is a reward interval, may be added to the logic PRCTL with the semantics:

$$s \models \mathbb{L}_R(\Phi) \;\text{ if and only if }\; \mathit{LongRunER}_s(\mathit{Sat}(\Phi)) \in R.$$

The above-sketched technique can then be used to compute the satisfaction set of $\mathbb{L}_R(\Phi)$.

10.6 Markov Decision Processes

Nondeterminism is absent in Markov chains. Markov decision processes (MDPs, for short) can be viewed as a variant of Markov chains that permits both probabilistic and nondeterministic choices. As in Markov chains, the probabilistic choices may serve to model and

quantify the possible outcomes of randomized actions such as tossing a coin or sending a message over a lossy communication channel. Probabilistic choices may also be adequate for modeling the interface of a system with its environment. For example, for a vending machine it might be reasonable to assign probability $\frac{9}{10}$ for the option chocolate bar, and probability $\frac{1}{10}$ for the option apple. This, however, requires statistical experiments to obtain adequate distributions that model the average behavior of the environment, i.e., the user of the vending machine. In cases where this information is not available, or where it is needed to guarantee system properties which should hold for all potential environments, a natural choice is to model the interface with the environment by *nondeterminism*.

Another important motivation for the incorporation of nondeterminism in probabilistic models such as Markov chains is provided by the field of randomized distributed algorithms. Such algorithms are concurrent and hence nondeterministic in nature. This is due to the interleaving of the behavior of the distributed processes involved, i.e., the nondeterministic choice to determine which of the concurrent processes performs the next step (see Chapter 2). Besides, they are probabilistic in the sense that typically a rather restricted set of actions (like tossing a coin or selecting a number from a certain range) has a random nature. A simple example is a two-process mutual exclusion protocol where access to the critical section is governed by an arbiter that on the basis of tossing a coin decides which process acquires access.

Finally, nondeterminism is crucial for *abstraction* techniques of Markov chains. Abstraction is typically based on the grouping of states. If this abstraction is based on probabilistic bisimulation, then there is no need for nondeterminism, as the transition probabilities between groups (i.e., equivalence classes) of states is uniquely determined. If, however, a coarser abstraction is considered where e.g., states are grouped on the basis of their atomic propositions, one obtains a range of probabilities for the transition probabilities—nondeterminism, so to speak. In the case of data abstraction, e.g., one might replace probabilistic branching by a nondeterministic choice.

Several operational models have been studied in the literature that all can be viewed as variants of Markov chains with both nondeterministic choices and discrete probabilities for the transition relation. We adopt the traditional notion of Markov decision processes (see e.g., the textbook [346]), extended by atomic propositions:

Definition 10.81. Markov Decision Process (MDP)

A *Markov decision process* is a tuple $\mathcal{M} = (S, Act, \mathbf{P}, \iota_{\text{init}}, AP, L)$ where

- S is a countable set of states,
- Act is a set of actions,

- $\mathbf{P} : S \times Act \times S \to [0,1]$ is the transition probability function such that for all states $s \in S$ and actions $\alpha \in Act$:

$$\sum_{s' \in S} \mathbf{P}(s, \alpha, s') \in \{\, 0, 1 \,\},$$

- $\iota_{\text{init}} : S \to [0,1]$ is the initial distribution such that $\sum_{s \in S} \iota_{\text{init}}(s) = 1$,
- AP is a set of atomic propositions and $L : S \to 2^{AP}$ a labeling function.

An action α is *enabled* in state s if and only if $\sum_{s' \in S} \mathbf{P}(s, \alpha, s') = 1$. Let $Act(s)$ denote the set of enabled actions in s. For any state $s \in S$, it is required that $Act(s) \neq \varnothing$. Each state s' for which $\mathbf{P}(s, \alpha, s') > 0$ is called an α-*successor* of s. ■

The transition probabilities $\mathbf{P}(s, \alpha, t)$ can be arbitrary real numbers in $[0,1]$ (that sum up to either zero or 1 for fixed s and α). For algorithmic purposes, the transition probabilities are supposed to be rational. Furthermore, the verification algorithms are restricted to finite MDPs. An MDP is finite whenever the state space S, the set Act of actions, and the set AP of atomic propositions are finite.

An MDP has a unique initial distribution ι_{init}. In fact, this could be generalized by allowing a set of initial distributions. A computation then starts by choosing one of these initial distributions nondeterministically. For the sake of simplicity, we just consider a single initial distribution.

The intuitive operational behavior of an MDP $\mathcal{M}$ is as follows. A stochastic experiment according to the initial distribution ι_{init}, yields a starting state s_0 such that $\iota_{\text{init}}(s_0) > 0$. On entering state s, say, first a nondeterministic choice between the enabled actions needs to be resolved. That is to say, it needs to be determined which action in $Act(s)$ is to be performed next. In the absence of any information about the frequency of actions—given actions α and β in $Act(s)$ it is unknown how often α needs to be selected—this choice is purely nondeterministic. Suppose action $\alpha \in Act(s)$ has been selected. On performing α in state s, one of the α-successors of s is selected randomly according to the distribution $\mathbf{P}(s, \alpha, \cdot)$. That is, with probability $\mathbf{P}(s, \alpha, t)$ the next state is t. If t is the unique α-successor of s, then almost surely t is the successor after selecting α, i.e., $\mathbf{P}(s, \alpha, t) = 1$. In this case, $\mathbf{P}(s, \alpha, u) = 0$ for all states $u \neq t$.

Any Markov chain is an MDP in which for any state s, $Act(s)$ is just a singleton set. Vice versa, any MDP with this property is a Markov chain. The action names are then irrelevant and can be omitted. Markov chains are thus a proper subset of MDPs.

The direct successors and predecessors of a state are defined as follows. For $s \in S$, $\alpha \in Act$ and $T \subseteq S$, let $\mathbf{P}(s, \alpha, T)$ denote the probability of moving to a state in T via α, i.e.,

$$\mathbf{P}(s, \alpha, T) \;=\; \sum_{t \in T} \mathbf{P}(s, \alpha, t) \quad .$$

$Post(s, \alpha)$ denotes the set of α-successors of s, i.e.,

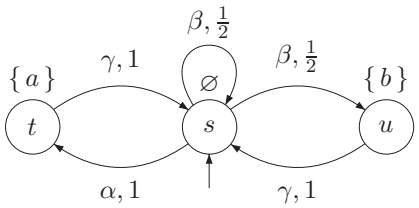
$$Post(s, \alpha) \;=\; \{\, t \in S \mid \mathbf{P}(s, \alpha, t) > 0 \,\}.$$

Note that $Post(s, \alpha) = \varnothing$ if and only if $\alpha \notin Act(s)$. $Pre(t)$ denotes the set of pairs (s, α) with $s \in S$ and $\alpha \in Act(s)$ such that $t \in Post(s, \alpha)$, i.e.,

$$Pre(t) \;=\; \{\, (s, \alpha) \in S \times Act \mid \mathbf{P}(s, \alpha, t) > 0 \,\}.$$

Example 10.82. An Example of an MDP

Consider the MDP $\mathcal{M}$ depicted below:



State s is the only initial state, i.e., $\iota_{\text{init}}(s) = 1$ and $\iota_{\text{init}}(t) = \iota_{\text{init}}(u) = 0$. The sets of enabled actions are

- $Act(s) = \{\, \alpha, \beta \,\}$ with $\mathbf{P}(s, \alpha, t) = 1$, $\mathbf{P}(s, \beta, u) = \mathbf{P}(s, \beta, s) = \frac{1}{2}$, and
- $Act(t) = Act(u) = \{\, \gamma \,\}$ with $\mathbf{P}(t, \gamma, s) = \mathbf{P}(u, \gamma, s) = 1$.

In state s, a nondeterministic choice between actions α and β exists. On selecting action α, the next state is t; on selecting action β, the successor states s and u are equally probable. Some successor and predecessor sets are $Post(s, \alpha) = \{\, t \,\}$, $Post(s, \beta) = \{\, s, u \,\}$, and $Pre(s) = \{\, (s, \beta), (t, \gamma), (u, \gamma) \,\}$. ■

Example 10.83. A Randomized Mutual Exclusion Protocol

Consider a simple randomized mutual exclusion protocol for two concurrent processes P_1

and P_2. The coordination of accessing their critical sections is provided by a randomized arbiter which permits process P_i to enter its critical section if the other process is in its noncritical section. If both processes acquire access to the critical section, the arbiter tosses a fair coin to decide which of the two processes has to wait and which process may enter the critical section.

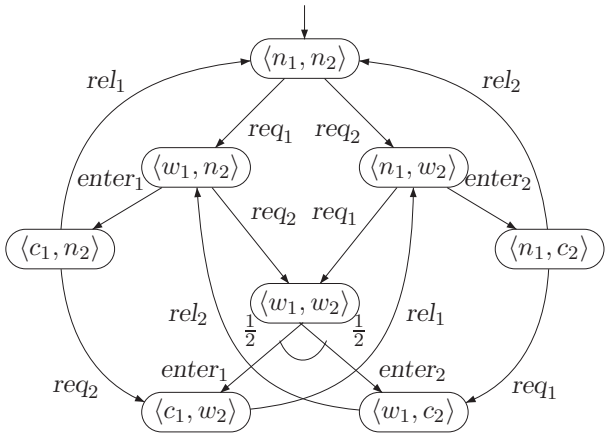


Figure 10.15: MDP for a randomized mutual exclusion protocol.

The composite behavior of the two concurrent processes and the arbiter can be modeled by the MDP depicted in Figure 10.15. All states of the MDP where at least one of the processes is in its noncritical location exhibit a nondeterministic choice between the enabled actions of P_1 and P_2. The corresponding *req*, *enter* or *rel* action does not have a proper probabilistic effect, since it yields a unique successor. The transition probabilities that equal 1 are omitted from the figure. Only in state $\langle wait_1, wait_2 \rangle$ is there a proper probabilistic choice performed by the arbiter to select the next process to enter the critical section. ■

Remark 10.84. Actions in an MDP

The action names $\alpha \in Act$ are needed here only for technical reason, viz. to group all edges belonging to the same probabilistic choice. When using MDPs in a compositional framework, where the MDP for a complex system arises through the parallel composition of several other MDPs, then it is more appropriate to allow for sets of distributions per action and state rather than just a single distribution. Note that, e.g., the interleaving of two equally named actions performed by different processes yields a global state that has two outgoing transitions with the same action label. This more general approach can be formalized by replacing $\mathbf{P}$ with a transition relation of the form

$$\rightarrow \ \subseteq \ S \times Act \times Distr(S)$$

where $Distr(S)$ denotes the set of distributions over S, i.e., functions $\mu : S \to [0,1]$ such that $\sum_{s \in S} \mu(s) = 1$. Labeled transition systems appear as special case of this type of transition relation. This follows from the fact that any transition $s \xrightarrow{\alpha} t$ in a transition system can be identified with the transition $s \xrightarrow{\alpha} \mu_t^1$ where $\mu_t^1 \in Distr(S)$ denotes the distribution with $\mu_t^1(t) = 1$ and $\mu_t^1(u) = 0$ for any $u \neq t$. The notion of MDP in this monograph (see Definition 10.81) is obtained by considering transitions $s \xrightarrow{\alpha} \mu_{s,\alpha}$ for the actions $\alpha \in Act(s)$, where $\mu_{s,\alpha}$ denotes the distribution $t \mapsto \mathbf{P}(s,\alpha,t)$. Since compositional approaches for MDPs fall outside the scope of this monograph, the action names are irrelevant. The actions may be assumed to be renamed such that per state and action there is at most one distribution. ■

Remark 10.85. Probmela

In a similar way as transition systems for nonprobabilistic (concurrent) programs can be described by higher-level modeling languages such as (nano)Promela, (see Section 2.2.5 on page 63), probabilistic systems can be specified by various higher-level description techniques. We consider the main features of a probabilistic variant of nanoPromela, called Probmela. As in nanoPromela, a Probmela model consists of finitely many concurrent processes that may communicate via either shared variables or channels (or both). The processes are described by statements of a *probabilistic guarded command language*. The core of this language is as in nanoPromela and consists of assignments, conditional commands (i.e., statements built by the keywords **if**–**fi**), loops (i.e., statements built by **do**–**od**), communication actions $c?x$ (for input) and $c!\mathsf{expr}$ (for output), and atomic regions. The language features three probabilistic features: *randomized assignments*, *probabilistic choice*, and *lossy channels*. Let us discuss these features in somewhat more detail.

- A random assignment has the form $x := \texttt{random}(V)$ where x is a variable and V is a finite nonempty subset of $dom(x)$. On executing this assignment, a value $v \in V$ is chosen probabilistically according to a uniform distribution over V and assigned to x. The probability that value $v \in V$ is assigned to x is thus $\frac{1}{|V|}$.
- The probabilistic choice operator is a probabilistic variant of **if**–**fi**-statements where the resolution of the choices is by probabilities rather than by guards. The syntax is

  $$\textbf{pif } [p_1] \Rightarrow stmt_1 \ \ldots \ [p_n] \Rightarrow stmt_n \textbf{ fip}$$

  where p_i is a non-negative real number such that $\sum_i p_i = 1$. The above statement models the probabilistic choice between the statements $stmt_1$ through $stmt_n$, where $stmt_i$ is selected with probability p_i.
- Communication channels may either be perfect (as in nanoPromela) or lossy. This is indicated upon declaring a channel. A perfect channel never loses a message,

$$
\begin{array}{llllllll}
c := s_0; & & & & & & & \\
\textbf{do} & :: c = s_0 & \Rightarrow \textbf{pif} & [\tfrac{1}{2}] \Rightarrow & c := s_{1,2,3} & [\tfrac{1}{2}] \Rightarrow & c := s_{4,5,6} & \textbf{fip} \\
 & :: c = s_{1,2,3} & \Rightarrow \textbf{pif} & [\tfrac{1}{2}] \Rightarrow & c := s'_{1,2,3} & [\tfrac{1}{2}] \Rightarrow & c := s_{2,3} & \textbf{fip} \\
 & :: c = s_{2,3} & \Rightarrow \textbf{pif} & [\tfrac{1}{2}] \Rightarrow & c := 2 & [\tfrac{1}{2}] \Rightarrow & c := 3 & \textbf{fip} \\
 & :: c = s'_{1,2,3} & \Rightarrow \textbf{pif} & [\tfrac{1}{2}] \Rightarrow & c := s_{1,2,3} & [\tfrac{1}{2}] \Rightarrow & c := 1 & \textbf{fip} \\
 & :: c = s_{4,5,6} & \Rightarrow \textbf{pif} & [\tfrac{1}{2}] \Rightarrow & c := s'_{4,5,6} & [\tfrac{1}{2}] \Rightarrow & c := s_{4,5} & \textbf{fip} \\
 & :: c = s_{4,5} & \Rightarrow \textbf{pif} & [\tfrac{1}{2}] \Rightarrow & c := 4 & [\tfrac{1}{2}] \Rightarrow & c := 5 & \textbf{fip} \\
 & :: c = s'_{4,5,6} & \Rightarrow \textbf{pif} & [\tfrac{1}{2}] \Rightarrow & c := s_{4,5,6} & [\tfrac{1}{2}] \Rightarrow & c := 6 & \textbf{fip} \\
\textbf{od} & & & & & & &
\end{array}
$$

Figure 10.16: Probmela specification of Knuth and Yao's simulation of a die by a fair coin.

whereas via a lossy channel a message is lost with probability p, a fixed probability that is defined on declaring the channel. The effect of the communication action $c!v$ along lossy channel c is that v is written into the buffer for c with probability $1-p$, and it fails (i.e., the content of the buffer for c remains unchanged while performing the action $c!v$) with probability p.

The stepwise behavior of a Probmela program can be formalized by means of an MDP. The states are tuples $\langle \ell_1, \ldots, \ell_n, \eta, \xi \rangle$ where ℓ_i is the location of process i, η is a variable evaluation, and ξ is a channel evaluation. This is the same as for nanoPromela. The transitions in the MDP specify a probability distribution over the successor states (i.e., an enabled action of a state and its probabilistic effect). As Probmela is only used for the examples, we refrain from presenting these inference rules here.

To illustrate the language Probmela, consider the simulation of a die by a coin; see Figure 10.2 on page 750. The Probmela code is shown in Figure 10.16. Here, c is a variable whose domain is the state space of the MC in Figure 10.2. The statements

$$\textbf{pif}\ [\tfrac{1}{2}] \Rightarrow c := \ldots\ [\tfrac{1}{2}] \Rightarrow c := \ldots\ \textbf{fip}$$

represent the coin-tossing experiments that are performed in the inner nodes. The final value of c stands for the die outcome. Note that the Probmela model just consists of a single process which does not behave nondeterministically. The semantics is thus a Markov chain. ■

Example 10.86. Alternating Bit Protocol with Lossy Channel

The alternating bit protocol (see Example 2.32 on page 57) can be modeled by an MDP,

assuming that the probability p for a channel to lose a message is known. An MDP of the alternating bit protocol can be obtained by specifying the behavior of the sender, the receiver, and the timer by Probmela statements. As in Example 2.32, the asynchronous channels c and d are used for connecting the sender and the receiver. The synchronous channel e is used for activating the timer and the timeout signal. Channel c is declared to be lossy with failure probability p, while channel d is supposed to be perfect. The sender is described by the following Probmela fragment:

$x := 0$
do :: true $\Rightarrow$ $c!x;\ e!on;$
if :: $d!y \Rightarrow x := \neg x;\ e!timeroff$
:: $e?y \Rightarrow c!x;\ e!on$
fi
od

The behavior of the receiver and the timer can be specified by analogous Probmela statements. As c is a lossy channel with probability p, the statement $c!x$ will not result in inserting message x into c. ■

Example 10.87. Randomized Dining Philosophers

Example 3.2 (page 90) treats the dining philosophers problem. We have seen that a naive symmetric protocol fails to ensure deadlock freedom as all philosophers might, e.g., pick up their left chopsticks simultaneously (or in any order). A deadlock-free solution is obtained when using variables that control the sticks and make them available to a single philosopher only. In order to break the symmetry, the initial values of these variables have to be chosen in such a way that initially at least one philosopher can pick up both sticks.

In the probabilistic setting, there are simpler solutions which are fully symmetric and that need neither global control nor additional shared variables. Consider, e.g., the proposal by Lehmann and Rabin [268]. The idea of their algorithm is that the philosophers toss a fair coin to decide which of the sticks they pick up first. If they fail to take the chosen stick, they then repeat the random choice. Otherwise they try to get the other stick. If the missing stick is not available, they return the taken stick and repeat the coin-tossing experiment. The Probmela code of philosopher i, shown in Figure 10.17 on page 840, uses Boolean variables $stick_i$ for the sticks that indicate whether stick i is available, where initially $stick_i = \text{true}$ for all sticks.

This randomized algorithm is deadlock-free as whenever a philosopher attempts to access an occupied stick, then he does not wait but repeats the coin-tossing experiment. Starvation freedom is guaranteed in the sense that almost surely any hungry philosopher who tries infinitely often to get the sticks will eventually eat.

```
mode_i := think;
do  :: true ⇒
    mode_i := try;
    pif  [1/2]  ⇒
                if  :: stick_i        ⇒  stick_i := false;
                                         if  :: stick_{i+1} ⇒  stick_{i+1} := false;
                                                               mode_i := eat;
                                                               stick_i := true;
                                                               stick_{i+1} := true;
                                                               mode_i := think;
                                             :: ¬stick_{i+1} ⇒  stick_i := true;
                                         fi
                    :: ¬stick_i       ⇒  skip
                fi
            [1/2]  ⇒
                if  :: stick_{i+1}    ⇒  stick_{i+1} := false;
                                         if  :: stick_i ⇒  stick_i := false;
                                                           mode_i := eat;
                                                           stick_i := true;
                                                           stick_{i+1} := true;
                                                           mode_i := think;
                                             :: ¬stick_i ⇒  stick_{i+1} := true;
                                         fi
                    :: ¬stick_{i+1}   ⇒  skip
                fi
    fip
od
```

Figure 10.17: Probmela code for philosopher i.

The semantics of the Probmela specification for n philosophers is an MDP. In each state, each of the philosophers has an enabled action. Most of the actions are nonprobabilistic in the sense that they yield a unique successor state. Only when the current location of some philosopher is the **pif**–**fip**-statement does its enabled action stand for the coin-tossing experiment to decide which stick to pick up first. This yields two equally likely successors. ■

Notation 10.88. Finiteness, Size, and Graph of an MDP

Let $\mathcal{M}$ be an MDP as in Definition 10.81 (page 833). $\mathcal{M}$ is called *finite* if the state space S, the action set Act, and the set AP of atomic propositions are finite. The *size* of $\mathcal{M}$ is

defined as the number of edges in the underlying graph of $\mathcal{M}$, i.e., the number of triples (s, α, t) such that $\mathbf{P}(s, \alpha, t) > 0$. Here, the underlying graph of $\mathcal{M}$ denotes the directed graph (S, E) where the states of $\mathcal{M}$ act as vertices and $(s, t) \in E$ if and only if there exists an action α such that $\mathbf{P}(s, \alpha, t) > 0$. ■

Notation 10.89. LTL/CTL-like Notations

As earlier in this chapter, we will often use LTL- or CTL-like notations with states or sets of states as atomic propositions. The satisfaction relation then refers to the underlying graph of the MDP, e.g., for $s \in S$ and $B \subseteq S$, the statement $s \models \exists\Diamond B$ means that some state in B is reachable from s in the underlying graph of the MDP $\mathcal{M}$. ■

The paths in an MDP $\mathcal{M}$ describe the potential computations that arise by resolving both the nondeterministic and probabilistic choices in $\mathcal{M}$. They are obtained by traversing the underlying graph via consecutive edges. Paths and path fragments in an MDP are defined as alternating sequences of states and actions.[2] More precisely,

Definition 10.90. Path in an MDP

An infinite *path fragment* in an MDP $\mathcal{M} = (S, Act, \mathbf{P}, \iota_{\text{init}}, AP, L)$ is an infinite sequence $s_0\, \alpha_1\, s_1\, \alpha_2\, s_2\, \alpha_3 \ldots \in (S \times Act)^\omega$, written as

$$\pi \;=\; s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots,$$

such that $\mathbf{P}(s_i, \alpha_{i+1}, s_{i+1}) > 0$ for all $i \geqslant 0$. Any finite prefix of π that ends in a state is a finite path fragment. $\mathit{Paths}(s)$ denotes the set of infinite path fragments that start in state s; $\mathit{Paths}_{\mathit{fin}}(s)$ denotes the set of finite path fragments that start in s. Let $\mathit{Paths}(\mathcal{M}) = \bigcup_{s \in S} \mathit{Paths}(s)$ and $\mathit{Paths}_{\mathit{fin}}(\mathcal{M}) = \bigcup_{s \in S} \mathit{Paths}_{\mathit{fin}}(s)$. ■

For Markov chains, the set of paths is equipped with a σ-algebra and a probability measure that reflects the intuitive notion of probabilities for (measurable sets of) paths. For MDPs, this is slightly different since no constraints are imposed on the resolution of the nondeterministic choices. Let us explain this phenomenon by means of a simple example. Suppose that $\mathcal{M}$ is an MDP with a single initial state s_0 with two enabled actions α and β both representing a coin-tossing experiment. The coin used for action α is supposed to be fair and yields *heads* and *tails* with equal probability, while action β represents flipping an unfair coin that provides the outcome *heads* with probability $\frac{1}{6}$ and *tails* with probability $\frac{5}{6}$. What is the probability for obtaining the outcome *tails* for the initial state s_0? This question cannot be answered; in fact, it is senseless, as this probability depends on whether

[2] As opposed to transition systems, no distinction is made between an execution and a path. The action names in the paths are used to keep track of the stepwise probabilities.

action α or action β will be chosen. This, however, is not specified! It is guaranteed that *heads* will appear with probability at least $\frac{1}{6}$ and at most $\frac{1}{2}$.

Also under the assumption that after performing α and β we return to state s_0 and repeat the selection of one of the actions α or β ad infinitum, the nondeterminism in s_0 can be resolved arbitrarily. Any sequence of the actions α and β constitutes a legal behavior of the MDP. Thus, it might even be the case that action β is never taken, in which case the probability of obtaining *heads* within the first n coin tosses is $1 - (\frac{1}{2})^n$. For the other extreme, if β is chosen the first n times, then the probability of obtaining the outcome *heads* at least once in the first n rounds is $1 - (\frac{5}{6})^n$. However, several other values are possible for the probability of the event "at least once outcome *heads* in the first n rounds". For instance, the value $1 - \frac{5^k}{6^k \cdot 2^{n-k}}$ is obtained if during the first $k \leqslant n$ steps, action β is chosen and from the $(k{+}1)$st step on action α is selected.

This example illustrates that MDPs are *not augmented with a unique probability measure*. Instead, reasoning about probabilities of sets of paths of an MDP relies on the resolution of nondeterminism. This resolution is performed by a *scheduler*. A scheduler chooses in any state s one of the enabled actions $\alpha \in Act(s)$. (Recall that $Act(s)$ is nonempty for any state s.) It does not impose any constraint on the probabilistic choice that is resolved once α has been chosen.

Definition 10.91. Scheduler

Let $\mathcal{M} = (S, Act, \mathbf{P}, \iota_{\text{init}}, AP, L)$ be an MDP. A *scheduler* for $\mathcal{M}$ is a function $\mathfrak{S} : S^+ \to Act$ such that $\mathfrak{S}(s_0\, s_1 \ldots s_n) \in Act(s_n)$ for all $s_0\, s_1 \ldots s_n \in S^+$.

The path (fragment)

$$\pi \;=\; s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots$$

is called a $\mathfrak{S}$-path (fragment) if $\alpha_i = \mathfrak{S}(s_0 \ldots s_{i-1})$ for all $i > 0$. ■

In the literature, a scheduler is sometimes referred to as adversary, policy, or strategy. Note that for any scheduler, the actions are dropped from the *history* $s_0\, s_1 \ldots s_n$. This is not a restriction as for any sequence $s_0\, s_1 \ldots s_n$ the relevant actions α_i are given by $\alpha_{i+1} = \mathfrak{S}(s_0\, s_1 \ldots s_i)$. Hence, the scheduled action sequence can be constructed from prefixes of the path at hand. Any path fragment $s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n$ where $\alpha_i \neq \mathfrak{S}(s_0\, s_1 \ldots s_i)$ for some i does not describe a path fragment that can be obtained from $\mathfrak{S}$.

As a scheduler resolves all nondeterministic choices in an MDP, it induces a Markov chain. That is to say, the behavior of an MDP $\mathcal{M}$ under the decisions of scheduler $\mathfrak{S}$ can be formalized by a Markov chain $\mathcal{M}_{\mathfrak{S}}$. Intuitively, this Markov chain arises by unfolding $\mathcal{M}$ into a tree, or forest if there are two or more states s with $\iota_{\text{init}}(s) > 0$. The paths in the

Markov chain represent the $\mathfrak{S}$-paths; the states in the Markov chain are state sequences of the Markov decision process $\mathcal{M}$. Formally:

Definition 10.92. Markov Chain of an MDP Induced by a Scheduler

Let $\mathcal{M} = (S, Act, \mathbf{P}, \iota_{\text{init}}, AP, L)$ be an MDP and $\mathfrak{S}$ a scheduler on $\mathcal{M}$. The Markov chain $\mathcal{M}_{\mathfrak{S}}$ is given by

$$\mathcal{M}_{\mathfrak{S}} \;=\; (S^+, \mathbf{P}_{\mathfrak{S}}, \iota_{\text{init}}, AP, L')$$

where for $\sigma = s_0 s_1 \ldots s_n$:

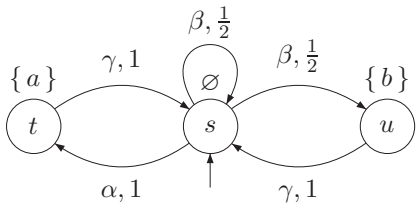
$$\mathbf{P}_{\mathfrak{S}}(\,\sigma,\; \sigma\, s_{n+1}\,) \;\;=\;\; \mathbf{P}(\,s_n,\, \mathfrak{S}(\sigma),\, s_{n+1}\,)$$

and $L'(\sigma) \,=\, L(s_n)$. ■

Note that $\mathcal{M}_{\mathfrak{S}}$ is infinite, even if the MDP $\mathcal{M}$ is finite. Intuitively, state $s_0\, s_1 \ldots s_n$ of $\mathcal{M}_{\mathfrak{S}}$ represents the configuration where the MDP $\mathcal{M}$ is in state s_n and $s_0\, s_1 \ldots s_{n-1}$ stands for the history, i.e., the path fragment that leads from the starting state s_0 to the current state s_n. Since $\mathfrak{S}$ might select different actions for path fragments that end in the same state s, a scheduler as in Definition 10.91 is also referred to as *history-dependent*.

Example 10.93. Markov Chain Induced by a Scheduler

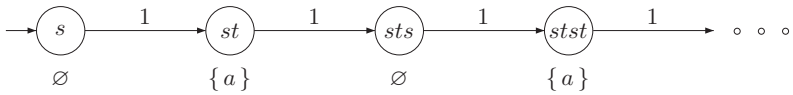
Consider the MDP of Example 10.82 (page 835), illustrated again below:



Let us consider some examples of schedulers for this MDP. Scheduler $\mathfrak{S}_\alpha$ always selects action α in state s. It is defined by $\mathfrak{S}_\alpha(\sigma) = \alpha$ if $last(\sigma) = s$, otherwise $\mathfrak{S}_\alpha(\sigma) = \gamma$. In a similar way, $\mathfrak{S}_\beta$ is defined by $\mathfrak{S}_\beta(\sigma) = \beta$ if $last(\sigma) = s$, otherwise $\mathfrak{S}_\beta(\sigma) = \gamma$. The only $\mathfrak{S}_\alpha$-path in $\mathcal{M}$ is $s \xrightarrow{\alpha} t \xrightarrow{\gamma} s \xrightarrow{\alpha} \ldots$. The path $s \xrightarrow{\beta} s \xrightarrow{\beta} s \xrightarrow{\beta} u \xrightarrow{\gamma} s \xrightarrow{\beta} u \ldots$ is a $\mathfrak{S}_\beta$-path. Likewise $s \xrightarrow{\beta} u \xrightarrow{\gamma} s \xrightarrow{\beta} u \ldots$.

Finally, let $\mathfrak{S}$ be a scheduler that selects α in s when just returning from u, and β otherwise. Thus: $\mathfrak{S}(s_0 \ldots s_n\, s) = \alpha$ if $s_n = u$ and $\mathfrak{S}(s_0 \ldots s_n\, s) = \beta$ otherwise. Let $\mathfrak{S}(s) = \alpha$. Note that this scheduler decides on the basis of the one-but-last visited state. In the states u and t, the only enabled action γ is chosen.

The Markov chain $\mathcal{M}_{\mathfrak{S}_\alpha}$ is the infinite chain:



An initial fragment of the Markov chain $\mathcal{M}_{\mathfrak{S}_\beta}$ is depicted in Figure 10.18 on page 844. ■

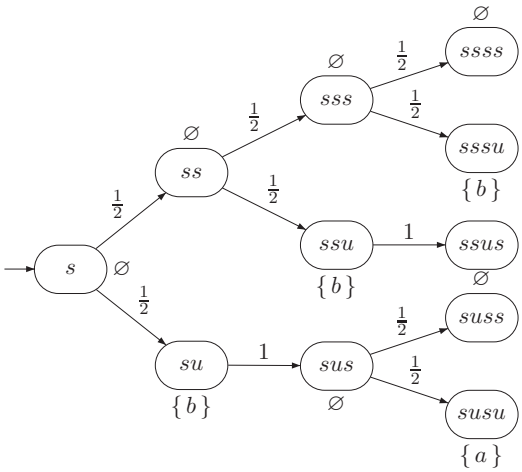


Figure 10.18: Initial fragment of the Markov chain $\mathcal{M}_{\mathfrak{S}_\beta}$.

There is a one-to-one correspondence between the $\mathfrak{S}$-paths of the MDP $\mathcal{M}$ and the paths in the Markov chain $\mathcal{M}_{\mathfrak{S}}$. For a $\mathfrak{S}$-path

$$\pi \;=\; s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots,$$

the corresponding path in the Markov chain $\mathcal{M}_{\mathfrak{S}}$ is given by

$$\pi^{\mathfrak{S}} \;=\; \widehat{\pi}_0\, \widehat{\pi}_1\, \widehat{\pi}_2\, \ldots$$

where $\widehat{\pi}_n \;=\; s_0\, s_1 \ldots s_n$. Vice versa, for a path $\widehat{\pi}_0\, \widehat{\pi}_1\, \widehat{\pi}_2\, \ldots$ in the Markov chain $\mathcal{M}_{\mathfrak{S}}$, $\widehat{\pi}_0 = s_0$ for some state s_0 such that $\iota_{\text{init}}(s_0) > 0$ and, for each $n > 0$, $\widehat{\pi}_n \;=\; \widehat{\pi}_{n-1}\, s_n$ for some state s_n in the MDP $\mathcal{M}$ such that $\mathbf{P}(s_{n-1}, \mathfrak{S}(\widehat{\pi}_{n-1}), s_n) > 0$. Hence:

$$s_0 \xrightarrow{\mathfrak{S}(\widehat{\pi}_0)} s_1 \xrightarrow{\mathfrak{S}(\widehat{\pi}_1)} s_2 \xrightarrow{\mathfrak{S}(\widehat{\pi}_2)} \ldots$$

is a $\mathfrak{S}$-path in $\mathcal{M}$. In the sequel, we often identify the paths in $\mathcal{M}_{\mathfrak{S}}$ with the corresponding $\mathfrak{S}$-paths in $\mathcal{M}$.

When we consider an ordinary transition system as an MDP (where any action taken in a state yields a unique successor), the concept of a scheduler is just another formalization of a path. In the general setting, i.e., for MDPs with proper probabilistic actions, however, any scheduler induces a *set* of paths.

As $\mathcal{M}_{\mathfrak{S}}$ is a Markov chain, one can now reason about the probabilities for measurable sets of $\mathfrak{S}$-paths. Let $Pr^{\mathcal{M}}_{\mathfrak{S}}$, or simply $Pr^{\mathfrak{S}}$, denote the probability measure $Pr^{\mathcal{M}_{\mathfrak{S}}}$ associated with the Markov chain $\mathcal{M}_{\mathfrak{S}}$. This measure is the basis for associating probabilities with events in the MDP $\mathcal{M}$. Let, e.g., $P \subseteq (2^{AP})^{\omega}$ be an ω-regular property. Then $Pr^{\mathfrak{S}}(P)$ is defined (see Notation 10.49, page 796) as the probability measure of the set of $\mathfrak{S}$-paths π in the Markov chain $\mathcal{M}_{\mathfrak{S}}$ such that $\mathit{trace}(\pi) \in P$:

$$Pr^{\mathfrak{S}}(P) \;=\; Pr^{\mathcal{M}_{\mathfrak{S}}}(P) \;=\; Pr_{\mathcal{M}_{\mathfrak{S}}}\{\, \pi \in \mathit{Paths}(\mathcal{M}_{\mathfrak{S}}) \mid \mathit{trace}(\pi) \in P \,\}.$$

Similarly, for fixed state s of $\mathcal{M}$, which is considered as the unique starting state,

$$Pr^{\mathfrak{S}}(s \models P) \;=\; Pr^{\mathcal{M}_{\mathfrak{S}}}_{s}\{\, \pi \in \mathit{Paths}(s) \mid \mathit{trace}(\pi) \in P \,\}$$

where we identify the paths in $\mathcal{M}_{\mathfrak{S}}$ with the corresponding $\mathfrak{S}$-paths in $\mathcal{M}$. This explains the notation above where the paths $\pi \in \mathit{Paths}(s)$ in $\mathcal{M}$ are identified with the associated paths $\pi^{\mathfrak{S}}$ in the Markov chain $\mathcal{M}_{\mathfrak{S}}$.

The quantitative analysis of an MDP $\mathcal{M}$ against ω-regular specifications amounts to establishing the best lower and/or upper probability bounds that can be guaranteed, when ranging over all schedulers. This corresponds to computing

$$\inf_{\mathfrak{S}} Pr^{\mathfrak{S}}(s \models P) \quad \text{and} \quad \sup_{\mathfrak{S}} Pr^{\mathfrak{S}}(s \models P)$$

where the infimum and the supremum are taken over all schedulers $\mathfrak{S}$ for $\mathcal{M}$. Later in this section, we will establish that the infimum and supremum may be replaced by minimum and maximum, respectively. Ranging over all schedulers and considering minimal or maximal probabilities corresponds to a *worst-case* analysis. This is due to the fact that the full class of schedulers covers all possible resolutions of the nondeterminism that is present.

Example 10.94. A Randomized Mutual Exclusion Protocol (Revisited)

Consider again the MDP modeling the mutual exclusion with a randomized arbiter; see Example 10.83 (page 835). Consider the ω-regular property stating that the first process infinitely often enters its critical section, i.e., $\Box\Diamond \mathit{crit}_1$. Let scheduler $\mathfrak{S}_1$ be such that in any state it only selects one of the enabled actions of the first process, and never an action of the second process. There is a single $\mathfrak{S}_1$-path where the first process successively passes

its three phases (noncritical, waiting, and critical), while the second process does nothing. Hence,

$$Pr_{\mathfrak{S}_1}(\Box\Diamond \mathit{crit}_1) \;=\; 1.$$

On the other extreme, for the scheduler $\mathfrak{S}_2$ which always selects one of the actions of the second process, but ignores the first process, we have

$$Pr_{\mathfrak{S}_2}(\Box\Diamond \mathit{crit}_1) \;=\; 0.$$

Let us now consider the probability for the event that the first process enters its critical section within at most three waiting rounds from state $\langle \mathit{wait}_1, \mathit{noncrit}_2\rangle$. A waiting round is interpreted as any path fragment of length three where the first process does nothing, while the second process passes through all three phases. For scheduler $\mathfrak{S}_1$, this event almost surely holds. The maximal probability is thus 1. The minimal probability is $1 - \left(\frac{1}{2}\right)^3$ and is obtained by the scheduler $\mathfrak{S}_3$ which selects the action of the second process whenever entering state $\langle \mathit{wait}_1, \mathit{noncrit}_2\rangle$. For the states $\langle \mathit{wait}_1, \mathit{wait}_2\rangle$ and $\langle \mathit{wait}_1, \mathit{crit}_2\rangle$, there is no proper choice and $\mathfrak{S}_3$ selects the uniquely enabled action. For the other states, the decisions of $\mathfrak{S}_3$ are irrelevant. ■

Example 10.95. Randomized Leader Election

Many communication protocols rely on the presence of a certain node (process) that acts as a leader. In order to select such a leader, a distributed algorithm is employed. We consider here the symmetric leader election protocol proposed by Itai and Rodeh [223]. It considers n processes in a unidirectional ring topology. Each node can act in either of two modes: active or passive. Process i is connected to its neighbors by FIFO channels c_i and c_{i+1}, each of capacity 1. Addition should be considered modulo the ring size n.

In the active mode, process i randomly chooses a bit and sends it via channel c_{i+1} to its unique successor on the ring. Process i receives the randomly chosen bit by process $i{-}1$ along channel c_i. If the bit of process i is 1, while the obtained bit is zero, then process i switches to the passive mode. Otherwise, process i takes part in the next round.

In the passive mode, process i just acts as a relay node: it passes each bit obtained from its predecessor $i{-}1$ to its successor $i{+}1$. The leader has been elected as soon as there is only one active process left. The Probmela-specification for process i is shown in Figure 10.19 on page 847.

The semantics of the composite Probmela program obtained by the specifications for the n processes is an MDP. The correctness of the algorithm can formally be established by showing that under each scheduler almost surely eventually the number of active processes (as indicated by the variable *#active*) equals one. ■

```
mode_i := active;
do  :: mode_i = active ⇒
           x_i := random(0,1);
           c_{i+1}!x_i; c_i?y_i;
           if  :: y_i = 1 ∧ x_i = 0  ⇒  mode_i := passive;
                                        #active := #active − 1
               :: y_i = 0 ∨ x_i = 1  ⇒  skip
           fi
    :: mode_i = passive ⇒  c_i?y_i; c_{i+1}!y_i
od
```

Figure 10.19: Probmela code for the randomized leader election protocol.

Definition 10.91 provides a rather general notion of scheduler and does not impose any restrictions on the decisions of a scheduler. The function from histories (i.e., finite path fragments) to actions does not require any consistency in the decisions and even allows for schedulers for which the decisions are not computable. For the verification of a large class of qualitative and quantitative properties of finite MDPs, however, a simple subclass of schedulers suffices. This means, e.g., that the maximal and minimal reachability probabilities (when ranging over the full class of schedulers) are obtained by schedulers of this subclass. The full generality of schedulers, as in Definition 10.91, is thus not needed.

Definition 10.96. Memoryless Scheduler

Let $\mathcal{M}$ be an MDP with state space S. Scheduler $\mathfrak{S}$ on $\mathcal{M}$ is *memoryless* (or: *simple*) iff for each sequence $s_0\, s_1\, \ldots s_n$ and $t_0\, t_1 \ldots t_m \in S^+$ with $s_n = t_m$:

$$\mathfrak{S}(s_0\, s_1\, \ldots s_n) \;=\; \mathfrak{S}(t_0\, t_1 \ldots t_m).$$

In this case, $\mathfrak{S}$ can be viewed as a function $\mathfrak{S} : S \to Act$. ■

Stated in words, scheduler $\mathfrak{S}$ is memoryless if it always selects the same action in a given state. This choice is independent of what has happened in the history, i.e., which path led to the current state.

For instance, the schedulers $\mathfrak{S}_1$, $\mathfrak{S}_2$ and $\mathfrak{S}_3$ described informally in Example 10.94 (page 845) are all memoryless. The schedulers $\mathfrak{S}_\alpha$ and $\mathfrak{S}_\beta$ in Example 10.93 (page 843) are also memoryless. The scheduler $\mathfrak{S}$ in Example 10.93 is not memoryless as it bases its decisions on the one-but-last state.

Memoryless schedulers are somehow extreme as they simply select one alternative (i.e., action) per state while ignoring all others. For instance, for the mutual exclusion example,

memoryless schedulers always select the same process to proceed in the states where both processes are enabled to perform an action. A variant of memoryless schedulers are so-called *finite-memory schedulers*, briefly fm-schedulers. The behavior of a fm-scheduler is described by a deterministic finite automaton (DFA). The selection of the action to be performed in the MDP $\mathcal{M}$ depends on the current state of $\mathcal{M}$ (as before) and the current state (called *mode*) of the scheduler, i.e., the DFA.

Definition 10.97. Finite-Memory Scheduler

Let $\mathcal{M}$ be an MDP with state space S and action set Act. A *finite-memory scheduler* $\mathfrak{S}$ for $\mathcal{M}$ is a tuple $\mathfrak{S} = (Q, act, \Delta, start)$ where

- Q is a finite set of *modes*,
- $\Delta : Q \times S \to Q$ is the transition function,
- $act : Q \times S \to Act$ is a function that selects an action $act(q, s) \in Act(s)$ for any mode $q \in Q$ and state s of $\mathcal{M}$,
- $start : S \to Q$ is a function that selects a starting mode for state s of $\mathcal{M}$.

■

The behavior of an MDP $\mathcal{M}$ under a finite-memory scheduler $\mathfrak{S} = (Q, act, \Delta, start)$ is as follows. Initially, a starting state s_0 is randomly determined according to the initial distribution ι_{init}, i.e., $\iota_{\text{init}}(s_0) > 0$. The fm-scheduler $\mathfrak{S}$ initializes its DFA to the mode $q_0 = start(s_0) \in Q$. Assume that $\mathcal{M}$ is in state s and the current mode of $\mathfrak{S}$ is q. The decision of $\mathfrak{S}$, i.e., the selected action, is now given by $\alpha = act(q, s) \in Act(s)$. The scheduler subsequently changes to mode $\Delta(q, s)$, while $\mathcal{M}$ performs the selected action α and randomly moves to the next state according to the distribution $\mathbf{P}(s, \alpha, \cdot)$.

Let us briefly explain how finite-memory schedulers are related to the (general) notion of a scheduler, see Definition 10.91 (page 842). A finite-memory scheduler $\mathfrak{S} = (Q, act, \Delta, start)$ is identified with the function, i.e., scheduler, $\mathfrak{S}' : Paths_{fin} \to Act$ which is defined as follows. For the starting state s_0, let $\mathfrak{S}'(s_0) = act(start(s_0), s_0)$. For path fragment $\widehat{\pi} = s_0\, s_1 \ldots s_n$ let

$$\mathfrak{S}'(\widehat{\pi}) \;=\; act(q_n, s_n)$$

where $q_0 = start(s_0)$ and $q_{i+1} = \Delta(q_i, s_i)$ for $0 \leqslant i \leqslant n$.

Finite-memory schedulers enjoy the property that the Markov chain $\mathcal{M}_{\mathfrak{S}}$ can be identified with a Markov chain where the states are just pairs $\langle s, q \rangle$ where s is a state in the MDP

$\mathcal{M}$ and q a mode of $\mathfrak{S}$. Formally, $\mathcal{M}'_{\mathfrak{S}}$ is the Markov chain with state space $S \times Q$, labeling $L'(\langle s, q\rangle) = L(s)$, the starting distribution ι_{init}, and the transition probabilities:

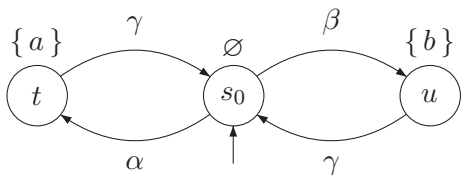
$$\mathbf{P}'_{\mathfrak{S}}(\langle s, q\rangle, \langle t, p\rangle) \;=\; \mathbf{P}(s, act(q, s), t).$$

It can be shown that the infinite Markov chain $\mathcal{M}_{\mathfrak{S}}$ is probabilistic bisimulation-equivalent to $\mathcal{M}'_{\mathfrak{S}}$. This justifies to identifying $\mathcal{M}_{\mathfrak{S}}$ with $\mathcal{M}'_{\mathfrak{S}}$. Hence, if $\mathcal{M}$ is a finite MDP, then we consider $\mathcal{M}_{\mathfrak{S}}$ as a finite MC.

Memoryless schedulers can be considered as finite-memory schedulers with just a single mode. That is, the Markov chain $\mathcal{M}_{\mathfrak{S}}$ induced by the memoryless scheduler $\mathfrak{S}$ can be viewed as a Markov chain with state space S. In particular, for finite $\mathcal{M}$, $\mathcal{M}_{\mathfrak{S}}$ can be viewed as a finite Markov chain which results by selecting a single enabled action per state (and discarding the other enabled actions). For finite MDP $\mathcal{M}$ the number of memoryless schedulers is finite, albeit sometimes very large. Assuming, e.g., there are exactly two enabled actions per state, then the total number of memoryless schedulers is 2^n where $n = |S|$.

Example 10.98. Memoryless vs. Finite-Memory Schedulers

Consider the MDP $\mathcal{M}$ depicted below:



We have

- $Act(s_0) = \{\, \alpha, \beta \,\}$, $\mathbf{P}(s_0, \alpha, t) = \mathbf{P}(s_0, \beta, u) = 1$, and
- $Act(t) = Act(u) = \{\, \gamma \,\}$ with $\mathbf{P}(t, \gamma, s_0) = \mathbf{P}(u, \gamma, s_0) = 1$.

The MDP is deterministic apart from state s_0 where a nondeterministic choice between actions α and β exists. There are only two memoryless schedulers for $\mathcal{M}$: scheduler $\mathfrak{S}_\alpha$, which always chooses α in s_0, and scheduler $\mathfrak{S}_\beta$, which always chooses β for s_0.

The β-successor u of s_0 is not reachable from s_0 in the Markov chain $\mathcal{M}_{\mathfrak{S}_\alpha}$ while by symmetry the α-successor t of s_0 is not accessible from s_0 under scheduler $\mathfrak{S}_\beta$. For both memoryless schedulers, the probability of satisfying the ω-regular property $\Diamond a \wedge \Diamond b$ is thus zero:

$$Pr_{\mathfrak{S}_\alpha}(s_0 \models \Diamond a \wedge \Diamond b) \;=\; Pr_{\mathfrak{S}_\beta}(s_0 \models \Diamond a \wedge \Diamond b) \;=\; 0.$$

However, for the—nonmemoryless—scheduler $\mathfrak{S}_{\alpha\beta}$, which alternates between selecting α and β (starting with α, say) on visiting s_0, the event $\Diamond a \wedge \Diamond b$ holds almost surely. In fact, since $\mathcal{M}$ does not contain a proper probabilistic choice there is exactly one $\mathfrak{S}_{\alpha\beta}$-path starting in s_0:

$$\pi \;=\; s_0 \xrightarrow{\alpha} t \xrightarrow{\gamma} s_0 \xrightarrow{\beta} u \xrightarrow{\gamma} s_0 \xrightarrow{\alpha} t \xrightarrow{\gamma} s_0 \xrightarrow{\beta} \ldots,$$

and $\pi \models \Diamond a \wedge \Diamond b$. Thus: although the event $\Diamond a \wedge \Diamond b$ holds with probability zero under all memoryless schedulers, there is a nonmemoryless scheduler for which this event almost surely holds. This shows that the class of memoryless schedulers is insufficiently powerful to characterize the minimal (or, dually, maximal) probability for ω-regular events.

The precise definition of the finite-memory scheduler $\mathfrak{S}_{\alpha\beta} = (Q, act, \Delta, start)$ is as follows. It has two modes: one in which it is only able to select α, while in the other mode only β can be selected. The scheduler switches mode whenever visiting s_0. Formally, the state space is $Q = \{ q_\alpha, q_\beta \}$. The action function is given by

$$act(q_\beta, s_0) = \beta \quad \text{and} \quad act(q_\alpha, s_0) = \alpha,$$

and $act(q, t) = act(q, u) = \gamma$ for $q \in Q$. Swapping modes is formalized by $\Delta(q_\beta, s_0) = q_\alpha$ and $\Delta(q_\alpha, s_0) = q_\beta$. If $\mathcal{M}$ is in state t or u, then $\mathfrak{S}$ stays in its current mode. This is formalized by $\Delta(q_\beta, t) = \Delta(q_\beta, u) = q_\beta$ and $\Delta(q_\alpha, t) = \Delta(q_\alpha, u) = q_\alpha$. As we assume that on the first visit of s_0, the action α is selected, the starting mode is defined by $start(s) = q_\alpha$ for all $s \in S$. ■

Remark 10.99. Randomized Schedulers

Schedulers, as in Definition 10.91 (page 842), are deterministic since they select a unique action for the current state. This can be generalized by allowing schedulers to select enabled actions probabilistically. That is, given a history a randomized scheduler returns a probability for each action. (The reader should not confuse this aspect of randomness with the random selection of the successor state within an MDP.) Mathematically, this means that randomized schedulers are functions $\mathfrak{S} : S^+ \to Distr(Act)$, where $Distr(Act)$ is a distribution over the set Act. It is required that any action α for which $\mathfrak{S}(s_0 \ldots s_n)(\alpha) > 0$ is enabled in the state s_n. We do not consider randomized schedulers any further and just mention that they can be approximated by deterministic schedulers and yield the same extreme probabilities for ω-regular properties as deterministic schedulers. Thus, although they are more general, randomized schedulers do not yield any extra power for our purposes. This justifies not treating them any further here. ■

10.6.1 Reachability Probabilities

This section treats the computation of (constrained) reachability probabilities in MDPs. That is, we are concerned with the following problem. Let $\mathcal{M} = (S, Act, \mathbf{P}, \iota_{\text{init}}, AP, L)$ be a finite MDP and $B \subseteq S$ a set of target states. The measure of interest is the maximal, or dually, the minimal probability of reaching a state in B when starting in state $s \in S$. For maximal probabilities this amounts to determining

$$Pr^{\max}(s \models \Diamond B) \quad = \quad \sup_{\mathfrak{S}} Pr^{\mathfrak{S}}(s \models \Diamond B).$$

Note that the supremum ranges over all, potentially infinitely many, schedulers for $\mathcal{M}$. This section will show that these maximal probabilities can be computed by solving a linear program. (Recall that reachability probabilities in MCs can be determined by solving a linear equation system.) Furthermore, it will be shown that rather than considering all schedulers—including history-dependent, finite-memory schedulers, and so forth—it suffices to only consider the subclass of memoryless schedulers. That is, there exists a memoryless scheduler that maximizes the probabilities to reach B. This holds for any state s. The supremum can thus be replaced by a maximum.

Reasoning about the maximal probabilities for $\Diamond B$ is needed, e.g., for showing that $Pr^{\mathfrak{S}}(s \models \Diamond B) \leqslant \varepsilon$ for all schedulers $\mathfrak{S}$ and some small upper bound $0 < \varepsilon \leqslant 1$. Then:

$$Pr^{\mathfrak{S}}(s \models \Box \neg B) \;\geqslant\; 1 - \varepsilon \quad \text{for all schedulers } \mathfrak{S}.$$

The task to compute $Pr^{\max}(s \models \Diamond B)$ can thus be understood as showing that a safety property (namely $\Box \neg B$) holds with sufficiently large probability, viz. $1 - \varepsilon$, regardless of the resolution of nondeterminism.

Theorem 10.100. Equation System for Max Reachability Probabilities

Let $\mathcal{M}$ be a finite MDP with state space S, $s \in S$ and $B \subseteq S$. The vector $(x_s)_{s \in S}$ with $x_s = Pr^{\max}(s \models \Diamond B)$ yields the unique solution of the following equation system:

- *If $s \in B$, then $x_s = 1$.*
- *If $s \not\models \exists \Diamond B$, then $x_s = 0$.*
- *If $s \notin B$ and $s \models \exists \Diamond B$, then*

$$x_s \;=\; \max \Big\{ \sum_{t \in S} \mathbf{P}(s, \alpha, t) \cdot x_t \;\mid\; \alpha \in Act(s) \Big\}.$$

The CTL-like notations in the last two items refer to the underlying digraph of the MDP $\mathcal{M}$. That is, $s \models \exists\Diamond B$ asserts that B is reachable from s. Obviously, $x_s = Pr^{\max}(s \models \Diamond B)$ is a solution of the above equation system. The proof of its uniqueness is rather technical and omitted here. It uses similar arguments as for Markov chains, see Theorem 10.19 on page 766. As for Markov chains, the second item could be omitted and replaced by the requirement that the equations for x_s in the third item hold for all states $s \in S \setminus B$. The uniqueness of $x_s = Pr^{\max}(s \models \Diamond B)$ is then, however, no longer guaranteed. As for Markov chains, one can prove that $(x_s)_{s\in S}$ is the least solution in $[0,1]^S$.

Example 10.101. Equation System for Max Reachability Probabilities

Consider the MDP $\mathcal{M}$ depicted in Figure 10.20 on page 852. Assume we are interested in $Pr^{\max}(s \models \Diamond B)$ with $B = \{ s_2 \}$. The vector $(x_s)_{s\in S}$ with $x_s = Pr^{\max}(s \models \Diamond B)$ yields the unique solution of the following equation system: $x_3 = 0$, $x_2 = 1$ and

$$x_0 \;=\; \max\{ \tfrac{3}{4}x_2 + \tfrac{1}{4}x_3, \tfrac{1}{2}x_2 + \tfrac{1}{2}x_1 \} \quad \text{and} \quad x_1 = \tfrac{1}{2}x_0 + \tfrac{1}{2}x_3 = \tfrac{1}{2}x_0$$

where x_i denotes x_{s_i}. The unique solution of this set of linear equations is: $(x_s)_{s\in S} = (\frac{3}{4}, \frac{3}{8}, 1, 0)$. ■

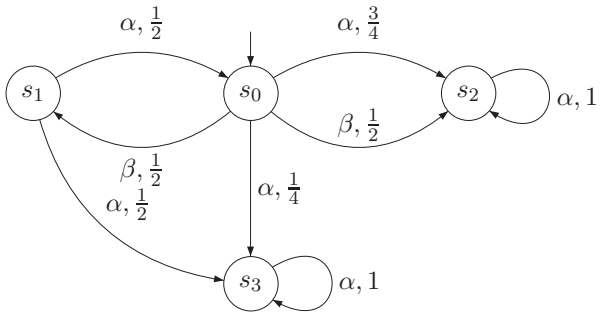


Figure 10.20: Example MDP for max reachability.

The following result asserts that there exists a memoryless scheduler that maximizes the probabilities of reaching B eventually. This holds for any state s in MDP $\mathcal{M}$.

Lemma 10.102. Existence of Optimal Memoryless Schedulers

Let $\mathcal{M}$ be a finite MDP with state space S, and $B \subseteq S$. There exists a memoryless scheduler $\mathfrak{S}$ such that for any $s \in S$

$$Pr^{\mathfrak{S}}(s \models \Diamond B) \;=\; Pr^{\max}(s \models \Diamond B).$$

Proof: Let $x_s = Pr^{\max}(s \models \Diamond B)$. The proof is by constructing a *memoryless* scheduler $\mathfrak{S}$ such that $Pr^{\mathfrak{S}}(s \models \Diamond B) = Pr^{\max}(s \models \Diamond B)$. This goes as follows. For any state s, let $Act^{\max}(s)$ be the set of actions $\alpha \in Act(s)$ such that

$$x_s = \sum_{t \in S} \mathbf{P}(s, \alpha, t) \cdot x_t \quad .$$

We first observe that it does not suffice to select arbitrary actions from the set $Act^{\max}(s)$. For example, consider a state with $Act^{\max}(s) = \{\alpha, \beta\}$ where $\mathbf{P}(s, \beta, t) = 1$ for some $t \in B$, while via α the set B cannot be reached, e.g., since $\mathbf{P}(s, \alpha, s) = 1$. A selection of actions is thus needed that ensures reachability of B in the induced Markov chain under $\mathfrak{S}$.

Consider the MDP $\mathcal{M}^{\max}$ that results from $\mathcal{M}$ by removing the actions $\beta \in Act(s) \setminus Act^{\max}(s)$ from $Act(s)$ for any state s for which $B \cap Post^*(s) \neq \varnothing$. This simplification of $\mathcal{M}$ does not change the maximal probabilities to reach B. For $s \models \exists\Diamond B$, let $\|s\|$ be the length of a shortest path fragment from s to B in the MDP $\mathcal{M}^{\max}$. It follows that $\|s\| = 0$ if and only if $s \in B$. By induction on $n \geqslant 1$ we define actions $\mathfrak{S}(s)$ for the states s with $s \models \exists\Diamond B$ and $\|s\| = n$. If $\|s\| = n \geqslant 1$, then choose an action $\mathfrak{S}(s) \in Act^{\max}(s)$ such that $\mathbf{P}(s, \mathfrak{S}(s), t) > 0$ for some state t with $t \models \exists\Diamond B$ and $\|t\| = n-1$. For the states s that cannot reach B, we select an arbitrary action $\mathfrak{S}(s) \in Act(s)$. This yields a memoryless scheduler $\mathfrak{S}$. The induced Markov chain $\mathcal{M}_{\mathfrak{S}}$ is finite with state space S. Moreover, the probabilities for $\Diamond B$ provide the unique solution of the linear equation system:

- If $s \in B$, then $y_s = 1$.
- If $s \not\models \exists\Diamond B$, then $y_s = 0$.
- If $s \notin B$ and $s \models \exists\Diamond B$, then $y_s = \sum_{t \in S} \mathbf{P}(s, \mathfrak{S}(s), t) \cdot y_t$.

Since the vector $(x_s)_{s \in S}$ also solves the above equation system we obtain:

$$Pr^{\mathfrak{S}}(s \models \Diamond B) = y_s = x_s = Pr^{\max}(s \models \Diamond B).$$

■

An optimal memoryless scheduler for the MDP in Figure 10.20 and the event $\Diamond B$ is the scheduler that selects α in any state.

Theorem 10.100 suggests an iterative approximation technique, called *value iteration*, to calculate the values $x_s = Pr^{\max}(s \models \Diamond B)$. By means of a backward reachability analysis, the set of states s is determined such that $s \models \exists\Diamond B$. These are precisely the states for which $x_s > 0$. For the states $s \in Pre^*(B) \setminus B$ we have

$$x_s \;=\; \lim_{n\to\infty} x_s^{(n)}$$

where

$$x_s^{(0)} = 0 \quad \text{and} \quad x_s^{(n+1)} \;=\; \max\Big\{ \sum_{t\in S} \mathbf{P}(s,\alpha,t)\cdot x_t^{(n)} \;\mid\; \alpha \in Act(s) \Big\}.$$

In the above equation also states in $B \cup (S \setminus Pre^*(B))$ might appear. For these states we have

$$x_s^{(n)} = 1 \text{ if } s \in B \quad \text{and} \quad x_s^{(n)} = 0 \text{ if } s \notin Pre^*(B).$$

Note that $x_s^{(0)} \leqslant x_s^{(1)} \leqslant x_s^{(2)} \leqslant \ldots$. Thus, the values $Pr^{\max}(s \models \Diamond B)$ can be approximated by successively computing the vectors

$$(\, x_s^{(0)} \,),\ (\, x_s^{(1)} \,),\ (\, x_s^{(2)} \,),\ \ldots,$$

until $\max_{s\in S} |x_s^{(n+1)} - x_s^{(n)}|$ is below a certain (typically very small) threshold.

Example 10.103. Value Iteration

To illustrate the value iteration, we consider the MDP depicted in Figure 10.21 on page 855. As the atomic propositions are irrelevant, they have been omitted. Let $B = \{\, s_3 \,\}$. It follows that $x_3^{(i)} = 1$ and $x_2^{(i)} = 0$ for any i. The latter follows from the fact that $s_2 \not\models \exists\Diamond B$. For the remaining states we obtain

$$\begin{array}{lcllcl}
x_0^{(i+1)} &=& \max\{\, \frac{2}{3}x_4^{(i)}, x_1^{(i)} \,\} & x_5^{(i+1)} &=& \max\{\, \frac{2}{3}x_7^{(i)}, x_6^{(i)} \,\} \\
x_1^{(i+1)} &=& \frac{1}{2}x_1^{(i)} + \frac{1}{9} & x_6^{(i+1)} &=& \frac{3}{5}x_6^{(i)} + \frac{2}{5}x_5^{(i)} \\
x_4^{(i+1)} &=& \frac{1}{4}x_5^{(i)} + \frac{3}{4}x_6^{(i)} & x_7^{(i+1)} &=& \frac{1}{2}.
\end{array}$$

The successive computation of the vectors $(x_s)_{s\in S}$ yields

$$(x^{(0)}) = (0,0,0,1,0,0,0,0) \quad \text{and} \quad (x^{(1)}) = (0, \tfrac{1}{9}, 0, 1, 0, 0, 0, \tfrac{1}{2})$$

and

$$(x^{(2)}) = (\tfrac{1}{9}, \tfrac{1}{6}, 0, 1, 0, \tfrac{1}{3}, 0, \tfrac{1}{2}) \quad \text{and} \quad (x^{(3)}) = (\tfrac{1}{6}, \tfrac{7}{36}, 0, 1, \tfrac{1}{12}, \tfrac{1}{3}, \tfrac{2}{15}, \tfrac{1}{2}) \ldots$$

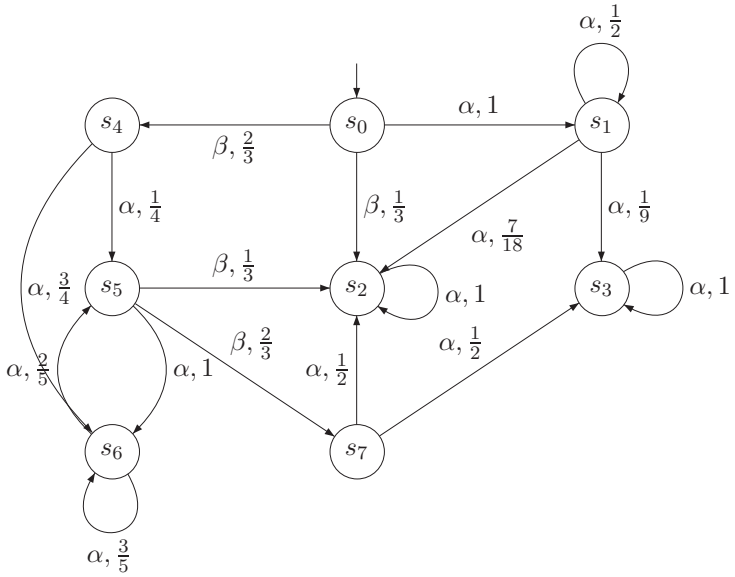


Figure 10.21: Example of an MDP.

■

Remark 10.104. Value Iteration for Step-Bounded Reachability Properties

The value iteration approach also yields a method to compute the maximal probabilities for the event $\Diamond^{\leqslant n} B$. In fact, we have

$$x_s^{(n)} \;=\; \sup_{\mathfrak{S}} Pr^{\mathfrak{S}}(s \models \Diamond^{\leqslant n} B)$$

where $\mathfrak{S}$ ranges over all schedulers. Furthermore, there exists a finite-memory scheduler $\mathfrak{S}$ that maximizes the probabilities for $\Diamond^{\leqslant n} B$. (Thus, the supremum can be replaced by maximum.) Such an optimal fm-scheduler can be composed by using modes $0, 1, \ldots, n$. The starting mode is 0 for each state in $\mathcal{M}$. The next-mode function changes from mode $i{-}1$ to i for $1 \leqslant i \leqslant n$. As soon as mode n has been reached, $\mathfrak{S}$ stays forever in mode n. For mode $i \in \{0, 1, \ldots, n{-}1\}$ and current state s in $\mathcal{M}$, $\mathfrak{S}$ selects an action $\alpha \in Act(s)$ that maximizes the value $\sum_{t \in S} \mathbf{P}(s, \alpha, t) \cdot x_t^{(i-1)}$. The actions $\mathfrak{S}(n, s)$ are arbitrary. Then, $Pr^{\mathfrak{S}}(s \models \Diamond^{\leqslant n} B) = x_s^{(n)}$. ■

An alternative method to compute $Pr^{\max}(s \models \Diamond B)$ is obtained by rewriting the equation system in Theorem 10.100 into a *linear program*.

Theorem 10.105. Linear Program for Max Reachability Probabilities

Let $\mathcal{M}$ be a finite MDP with state space S, and $B \subseteq S$. The vector $(x_s)_{s\in S}$ with $x_s = Pr^{\max}(s \models \Diamond B)$ yields the unique solution of the following linear program*:*

- *If $s \in B$, then $x_s = 1$.*
- *If $s \not\models \exists\Diamond B$, then $x_s = 0$.*
- *If $s \notin B$ and $s \models \exists\Diamond B$, then $0 \leqslant x_s \leqslant 1$ and for all actions $\alpha \in Act(s)$:*

$$x_s \;\geqslant\; \sum_{t\in S} \mathbf{P}(s,\alpha,t) \cdot x_t$$

where $\sum_{s\in S} x_s$ is minimal.

Proof: It is not difficult to see that the vector $(x_s)_{s\in S}$ with $x_s = Pr^{\max}(s \models \Diamond B)$ is a solution to the equations in the first two items, and satisfies the inequalities in the last item. Hence, there exists a solution $(y_s)_{s\in S}$ of the above linear program. Since the sum of the elements of y_s is minimal under all vectors that satisfy the linear program, it follows that $\sum_s x_s \geqslant \sum_s y_s$.

It remains to show the uniqueness of the solution. In fact, any solution $(y_s)_{s\in S}$ of the linear program is also a solution of the equation system in Theorem 10.100. Due to the minimality of $\sum_{s\in S} y_s$, the value y_s in the third item agrees with $\max\{\, \sum_t \mathbf{P}(s,\alpha,t)\cdot y_t \mid \alpha \in Act(s)\,\}$.

This can be seen as follows. If this would not hold, then we could apply the value iteration process starting with $x_s^{(0)} = y_s$ for all s. This yields a decreasing sequence of vectors $(x_s^{(n)})_{n\geqslant 0}$. The limit $(y'_s)_{s\in S}$ solves both the first three items of the linear program and the equations in Theorem 10.100. As $y'_s = \lim_{n\to\infty} x_s^{(n)} \leqslant y_s$, the minimality of $\sum_{s\in S} y_s$ as a solution of the inequalities in Theorem 10.105 yields $y'_s = y_s$.

Thus, Theorem 10.100 yields that $y_s = x_s = Pr^{\max}(s \models \Diamond B)$ for all states s. ∎

Example 10.106. Linear Program

Consider again the MDP in Figure 10.21 and let $B = \{\, s_3 \,\}$. The equations of the linear

program for $Pr^{\max}(s \models \Diamond B)$ are:

$$\begin{array}{lcllcllcl} x_0 & \geqslant & \frac{2}{3}x_4 & x_0 & \geqslant & x_1 & x_1 & \geqslant & \frac{1}{2}x_1 + \frac{1}{9} \\ x_2 & = & 0 & x_3 & = & 1 & x_4 & \geqslant & \frac{1}{4}x_5 + \frac{3}{4}x_6 \\ x_5 & \geqslant & \frac{2}{3}x_7 & x_5 & \geqslant & x_6 & x_6 & \geqslant & \frac{3}{5}x_6 + \frac{2}{5}x_5 \\ x_7 & \geqslant & \frac{1}{2} & & & & & & \end{array}$$

■

Note that the third item in Theorem 10.105 can be rewritten into

$$\sum_{t \in S_? \setminus \{ s \}} \mathbf{P}(s,\alpha,t) \cdot x_t \; + \; (1 - \mathbf{P}(s,\alpha,s)) \cdot x_s \quad \geqslant \quad \mathbf{P}(s,\alpha,B)$$

where $\mathbf{P}(s,\alpha,B) = \sum_{t \in B} \mathbf{P}(s,\alpha,t)$. The set $S_?$ contains all states such that the value $x_s = Pr^{\max}(s \models \Diamond B)$ is not fixed to 0 or 1 by the first two items, i.e., $S_? = \{ s \in S \setminus B \mid s \models \exists \Diamond B \}$. Hence:

$$s \in S_? \quad \text{if and only if} \quad s \notin B \text{ and } Pr^{\max}(s \models \Diamond B) > 0.$$

Thus, the third item in the above theorem can be read as a linear inequality $\mathbf{A} \cdot \mathbf{x} \geqslant \mathbf{b}$ where $\mathbf{x}$ is the vector $(x_s)_{s \in S_?}$ and $\mathbf{A}$ is a matrix with a row for each pair (s,α) with $s \in S_?$ and $\alpha \in Act(s)$, and two extra rows for each state $s \in S_?$ and a column for each state $t \in S_?$. The entry of $\mathbf{A}$ in the row for (s,α) and column for state t equals $\mathbf{P}(s,\alpha,t)$ provided $s \neq t$. For $s = t$ the entry of $\mathbf{A}$ equals $1 - \mathbf{P}(s,\alpha,s)$. The two extra rows represent the inequality $0 \leqslant x_s \leqslant 1$, which can be split into the two constraints $x_s \geqslant 0$ and $-x_s \geqslant -1$. Similarly, $\mathbf{b}$ is a vector with a component for each pair (s,α) where $s \in S_?$ and $\alpha \in Act(s)$ and two extra components per state $s \in S_?$. The value of $\mathbf{b}$ for (s,α) is the probability of moving from s via action α to a state in B (i.e., the value $\mathbf{P}(s,\alpha,B)$).

In this sense, Theorem 10.105 yields a characterization of the values $Pr(s \models \Diamond B)$ as the linear optimization problem which asks for the unique solution of $\mathbf{A} \cdot \mathbf{x} \geqslant \mathbf{b}$ under the side condition that $\sum_{s \in S} x_s$ is minimal under all solutions of $\mathbf{Ax} \geqslant \mathbf{b}$. The precise values for $Pr^{\max}(s \models \Diamond B)$ can thus be computed by standard algorithms to solve linear programs, e.g., the simplex algorithm or polytime methods [367].

Corollary 10.107. Complexity of Computing Max Reachability Probabilities

For finite MDP $\mathcal{M}$ with state space S, $B \subseteq S$ and $s \in S$, the values $Pr^{\max}(s \models \Diamond B)$ can be computed in time polynomial in the size of $\mathcal{M}$.

As a consequence, the question whether $Pr^{\mathfrak{S}}(s \models \Diamond B) \leqslant p$ for some upper bound $p \in [0,1[$ is decidable in polynomial time. This result, however, is more of theoretical interest

than of practical relevance. In practice, the simplex method often outperforms polytime algorithms for linear programs, although its worst-case time complexity is exponential. Experiments with randomized distributed algorithms (modeled as MDPs) indicates that the value iteration method is often faster than the simplex method.

The value iteration or linear program approach can be improved by first computing the set of states s with $Pr^{\max}(s \models \Diamond B) = 1$. This can be efficiently done by standard graph algorithms. For these states, the corresponding equation and inequalities, respectively, for x_s in the third item of Theorem 10.100 and 10.105 respectively can be omitted. This simplifies the value iteration procedure and decreases the size of the linear program.

Let us now consider the *qualitative reachability analysis* of a finite MDP $\mathcal{M}$. The goal is to compute the set of states s for which $Pr^{\max}(s \models \Diamond B) = 1$. Similar to the case with Markov chains, the states in B are first made absorbing. This means that any state $s \in B$ is equipped with a single enabled action α_s with $\mathbf{P}(s, \alpha_s, s) = 1$. Since there exists a memoryless scheduler that maximizes the probabilities for the event $\Diamond B$, it suffices to compute the set of states s such that $Pr^{\mathfrak{S}}(s \models \Diamond B) = 1$ for some memoryless scheduler $\mathfrak{S}$, say. The latter is equivalent to the requirement that no state $t \not\models \exists\Diamond B$ is reachable from s in the Markov chain $\mathcal{M}^{\mathfrak{S}}$ via a path fragment in $S \setminus B$. As the number of memoryless schedulers may be exponential (in the size of $\mathcal{M}$), considering all such schedulers is not an adequate solution.

However, the analysis of the underlying graph of $\mathcal{M}$ suffices to determine all states s where $Pr^{\max}(s \models \Diamond B) = 1$. This approach is iterative and successively removes all vertices u with $Pr^{\max}(u \models \Diamond B) < 1$. Initially, each vertex $u \in U_0$ with $U_0 = S \setminus Sat(\exists\Diamond B)$ is removed. The set U_0 can simply be determined by a graph analysis. Then, for all states t, all actions α are deleted from $Act(t)$ that satisfy $Post(t, \alpha) \cap U_0 \neq \varnothing$. If after deleting these actions, $Act(t) = \varnothing$, then t will be removed. This procedure is repeated as long as possible, and yields the MDP $\mathcal{M}_1$. Then, we restart the whole procedure with $\mathcal{M}_1$ rather than $\mathcal{M}$, etc., until all states in the obtained MDP $\mathcal{M}_i$ can reach B.

This approach is outlined in Algorithm 45. This algorithm treats the MDP as a directed graph where the successors of any vertex $t \in S$ are the pairs (t, α) with $\alpha \in Act(t)$. The outgoing edges of the auxiliary vertices (t, α) lead to the vertices $u \in Post(t, \alpha) = \{\, u \in S \mid \mathbf{P}(t, \alpha, u) > 0 \,\}$. For $u \in S$, $Pre(u)$ denotes the set of state action pairs $(t, \alpha) \in S \times Act$ such that $\mathbf{P}(t, \alpha, u) > 0$. Removing α from $Act(t)$ means removing the auxiliary vertex (t, α), the edge from vertex t to vertex (t, α), and the outgoing edges from (t, α).

Algorithm 45 Computing the set of states s with $Pr^{\max}(s \models \Diamond B) = 1$

Input: MDP $\mathcal{M}$ with finite state space S, $B \subseteq S$ for $s \in B : Act(s) = \{ \alpha_s \}$ and $\mathbf{P}(s, \alpha_s, s) = 1$ (i.e., B is absorbing)
Output: $\{ s \in S \mid Pr^{\max}(s \models \Diamond B) = 1 \}$

$U := \{ s \in S \mid s \not\models \exists \Diamond B \}$;
repeat
  $R := U$;
  while $R \neq \varnothing$ **do**
    let $u \in R$;
    $R := R \setminus \{ u \}$;
    for all $(t, \alpha) \in Pre(u)$ such that $t \notin U$ **do**
      remove α from $Act(t)$;
      if $Act(t) = \varnothing$ **then**
        $R := R \cup \{ t \}$;
        $U := U \cup \{ t \}$;
      fi
    od
    (* all incoming edges of u have been removed *)
    remove u and its outgoing edges from $\mathcal{M}$
  od
  (* determine the states s that cannot reach B in the modified MDP *)
  $U := \{s \in S \setminus U \mid s \not\models \exists \Diamond B\}$;
until $U = \varnothing$
(* all states can reach B in the generated sub-MDP of $\mathcal{M}$ *)
return all states in the remaining MDP

Lemma 10.108. Correctness of Algorithm 45

For finite MDP $\mathcal{M}$ and B, a set of states in $\mathcal{M}$, Algorithm 45 returns the set of all states s in $\mathcal{M}$ such that $Pr^{\max}(s \models \Diamond B) = 1$.

Proof: The correctness of Algorithm 45 relies on the following two facts: (i) for any state t which is removed, $Pr^{\max}(t \models \Diamond B) < 1$, and (ii) for any action α removed from $Act(t)$ there is no memoryless scheduler $\mathfrak{S}$ with $\mathfrak{S}(t) = \alpha$ and $Pr^{\mathfrak{S}}(t \models \Diamond B) = 1$.

Now let $\mathcal{M}_0 = \mathcal{M}, \mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_i$ be the sequence of MDPs that are generated by Algorithm 45. More precisely, $\mathcal{M}_j$ is the MDP obtained by the first j iterations of the repeat loop. The final MDP $\mathcal{M}_i$ still contains all states s where $Pr^{\max}(s \models \Diamond B) = 1$ and all actions $\alpha \in Act(s)$ that can be used by an optimal (memoryless) scheduler for $\Diamond B$.

Let $\mathfrak{S}$ be a finite-memory scheduler for $\mathcal{M}_i$ that treats all enabled actions in a fair way (i.e., if t is visited infinitely often, then any action that is enabled in t is taken infinitely often). Then the (finite) Markov chain $\mathcal{M}_i^{\mathfrak{S}}$ induced by $\mathfrak{S}$ enjoys the property that all states can reach B. Corollary 10.30 on page 777 then yields that $Pr^{\mathfrak{S}}(s \models \Diamond B) = 1$ for all states s in $\mathcal{M}_i$. Thus, the set of states in $\mathcal{M}_i$ agrees with the set of states where $Pr^{\max}(s \models \Diamond B) = 1$. ∎

The worst-case complexity of Algorithm 45 is quadratic in the size of $\mathcal{M}$. This can be seen as follows. The maximal number of iterations of the outermost loop is $N = |S|$, as in each iteration at least one state is eliminated. In each iteration, the set of states that can reach B needs to be computed. This takes $\mathcal{O}(size(\mathcal{M}))$ time. The other operations cause the overall cost $\mathcal{O}(size(\mathcal{M}))$, since any state t and state action pair (t, α) can be removed at most once.

So far, this section has addressed the problem of computing the maximal probabilities for reaching a certain set B of states in an MDP. Typically, B represents a set of bad states and the aim is to show that, regardless of the scheduling policy, the probability of entering a B-state is sufficiently small. We will now show that similar techniques are applicable to compute the *minimal* probabilities for reaching a certain set of states. When, e.g., B stands for a set of good states then

$$Pr^{\min}(s \models \Diamond B) \;=\; \inf_{\mathfrak{S}} Pr^{\mathfrak{S}}(s \models \Diamond B)$$

yields the best lower bound that can be guaranteed for the probability of eventually reaching B. In analogy to Theorem 10.100, the minimal probabilities can be characterized by an equation system. To ensure the uniqueness of the solution of this equation system it is required that all states s such that $Pr^{\min}(s \models \Diamond B) = 0$ have the value $x_s = 0$.

Theorem 10.109. Equation System for Min Reachability Probabilities

Let $\mathcal{M}$ be a finite MDP with state space S and $B \subseteq S$. The vector $(x_s)_{s\in S}$ with $x_s = Pr^{\min}(s \models \Diamond B)$ yields the unique solution of the following equation system:

- *If $s \in B$, then $x_s = 1$.*
- *If $Pr^{\min}(s \models \Diamond B) = 0$, then $x_s = 0$.*
- *If $Pr^{\min}(s \models \Diamond B) > 0$ and $s \notin B$, then:*

$$x_s \;=\; \min\Big\{ \sum_{t\in S} \mathbf{P}(s,\alpha,t)\cdot x_t \;\mid\; \alpha \in Act(s) \Big\}.$$

Theorem 10.109 suggests that we first compute the set containing all states s satisfying $Pr^{\min}(s \models \Diamond B) = 0$, followed by a *value iteration* method to obtain an approximation of the values $x_s = Pr^{\min}(s \models \Diamond B)$ for the states with $x_s > 0$ and $s \notin B$. For this, we put $x_s^{(i)} = 0$ if $Pr^{\min}(s \models \Diamond B) = 0$, and $x_s^{(i)} = 1$ if $s \in B$, for all i. For the remaining states, the iteration is started with $x_s^{(0)} = 0$. For successive iterations, let

$$x_s^{(n+1)} \;=\; \min\Big\{ \sum_{t\in S} \mathbf{P}(s,\alpha,t)\cdot x_t^{(n)} \;\mid\; \alpha \in Act(s) \Big\}.$$

Then: $x_s^{(0)} \leqslant x_s^{(1)} \leqslant x_s^{(2)} \leqslant \ldots$ and

$$\lim_{n\to\infty} x_s^{(n)} \;=\; Pr^{\min}(s \models \Diamond B).$$

Moreover, $x_s^{(n)}$ agrees with the minimal probability of reaching B within at most n steps, where the minimum is taken over all schedulers. That is:

$$x_s^{(n)} \;=\; \min_{\mathfrak{S}} Pr^{\mathfrak{S}}(s \models \Diamond^{\leqslant n} B).$$

In fact, this minimum always exists and can be established by a finite-memory scheduler. This scheduler has n modes, 0 through $n-1$. The starting mode is 0, the next-mode function switches from mode $i-1$ to mode i for $0 < i \leqslant n$ and stays forever in mode n after the nth step. $\mathfrak{S}(i,s)$ is an action $\alpha \in Act(s)$ that minimizes the value $\sum_{t\in S} \mathbf{P}(s,\alpha,t)\cdot x_t^{(i-1)}$. The actions $\mathfrak{S}(n,s) \in Act(s)$ are irrelevant.

The preprocessing required to compute the set

$$S_{=0}^{\min} \;=\; \{\, s \in S \mid Pr^{\min}(s \models \Diamond B) \,\} = 0$$

can be performed by graph algorithms. The set $S^{\min}_{=0}$ is given by $S \setminus T$ where

$$T = \bigcup_{n \geqslant 0} T_n$$

and $T_0 = B$ and, for $n \geqslant 0$:

$$T_{n+1} \;=\; T_n \cup \{\, s \in S \mid \forall \alpha \in Act(s)\, \exists t \in T_n.\, \mathbf{P}(s,\alpha,t) > 0 \,\}.$$

As $T_0 \subseteq T_1 \subseteq T_2 \subseteq \ldots \subseteq S$ and S is finite, the sequence $(T_n)_{n \geqslant 0}$ eventually stabilizes, i.e.,there exists some $n \geqslant 0$ such that $T_n = T_{n+1} = \ldots = T$.

Lemma 10.110. Characterization of $S^{\min}_{=0}$

Let $\mathcal{M}$ and T be as above. Then, $S^{\min}_{=0} = S \setminus T$, i.e., for all states $s \in S$:

$$Pr^{\min}(s \models \Diamond B) > 0 \quad \textit{iff} \quad s \in T.$$

Proof: We show by induction on n that $Pr^{\min}(s \models \Diamond B) > 0$ for all $s \in T_n$. Base case: For n=0 the claim trivially holds as $T_0 = B$ and $Pr^{\mathfrak{S}}(s \models \Diamond B) = 1$ for each scheduler $\mathfrak{S}$ and state $s \in B$. Induction step: Assume the claim holds for 0 through $n-1$. Let $n \geqslant 1$. Consider $s \in T_n \setminus T_{n-1}$ and let $\mathfrak{S}$ be a scheduler. Let $\alpha = \mathfrak{S}(s)$ be the action selected by $\mathfrak{S}$ when $\mathcal{M}$ starts in s. By definition of T_n, $T_{n-1} \cap Post(s,\alpha) \neq \varnothing$. Let $t \in T_{n-1} \cap Post(s,\alpha)$. By induction hypothesis, it holds that $Pr^{\min}(t \models \Diamond B) > 0$. For $s \in T_n$, it then holds that

$$Pr^{\mathfrak{S}}(s \models \Diamond B) \;\geqslant\; \mathbf{P}(s,\alpha,t) \cdot Pr^{\min}(t \models \Diamond B) \;>\; 0.$$

Therefore, $Pr^{\min}(s \models \Diamond B) > 0$.

Vice versa, if $s \in S \setminus T$, then $Post(s,\alpha) \cap T = \varnothing$ for some action $\alpha \in Act(s)$. Hence, we may consider a memoryless scheduler $\mathfrak{S}$ which selects for each state $s \in S \setminus T$ such action $\alpha \in Act(s)$. Then, in the Markov chain $\mathcal{M}_{\mathfrak{S}}$ the states in $S \setminus T$ cannot reach any state in T. Therefore, $Pr^{\mathfrak{S}}(s \models \Diamond B) = 0$ for all $s \in S \setminus T$. This yields $Pr^{\min}(s \models \Diamond B) = 0$ for $s \in S \setminus T$. ∎

The computation of $S \setminus S^{\min}_{=0} = T = \bigcup_{n \geqslant 0} T_n$ can be performed in time linear in the size of $\mathcal{M}$. The main steps of such a linear-time algorithm are summarized in Algorithm 46.

To speed up the convergence of the value iteration process, one might also compute $S^{\min}_{=1} = \{\, s \in S \mid Pr^{\min}(s \models \Diamond B) = 1 \,\}$ using graph analysis techniques. Note that the computation of $S^{\min}_{=1}$ also solves the qualitative verification problem which asks whether $\Diamond B$ almost surely holds under all schedulers.

Algorithm 46 Computing the set of states s with $Pr^{\min}(s \models \Diamond B) = 0$

Input: finite MDP $\mathcal{M}$ with state space S and $B \subseteq S$
Output: $\{ s \in S \mid Pr^{\min}(s \models \Diamond B) = 0 \}$

```
T := B;
R := B;
while R ≠ ∅ do
  let t ∈ R;
  R := R \ { t };
  for all (s, α) ∈ Pre(t) with s ∉ T do
    remove α from Act(s)
    if Act(s) = ∅ then
      add s to R and T
    fi
  od
od
return T
```

Lemma 10.111. Characterization of $S^{\min}_{=1}$

Let $\mathcal{M}$ be a finite MDP with state space S, $B \subseteq S$ and $s \in S$. The following statements are equivalent:

(a) $Pr^{\min}(s \models \Diamond B) < 1$.

(b) There exists a memoryless scheduler $\mathfrak{S}$ *such that* $Pr^{\mathfrak{S}}(s \models \Box\neg B) > 0$.

(c) $s \models \exists((\neg B) \,\mathsf{U}\, t)$ *for some state* t *such that* $Pr^{\mathfrak{S}}(t \models \Box\neg B) = 1$ *for some memoryless scheduler* $\mathfrak{S}$.

Proof: (a) $\Longrightarrow$ (b): If $Pr^{\min}(s \models \Diamond B) < 1$ then, by Lemma 10.113 (page 865), $Pr^{\mathfrak{S}}(s \models \Diamond B) < 1$ for some memoryless scheduler $\mathfrak{S}$. But then

$$Pr^{\mathfrak{S}}(s \models \Box\neg B) \;=\; 1 - Pr^{\mathfrak{S}}(s \models \Diamond B) \;>\; 0.$$

(b) $\Longrightarrow$ (c): Let $\mathfrak{S}$ be a memoryless scheduler such that $Pr^{\mathfrak{S}}(s \models \Box\neg B) > 0$. Consider the Markov chain $\mathcal{M}_{\mathfrak{S}}$. As $\mathfrak{S}$ is memoryless, $\mathcal{M}_{\mathfrak{S}}$ is finite. By Theorem 10.27 on page 775, almost all paths in $\mathcal{M}_{\mathfrak{S}}$ eventually enter a BSCC and visit all its states infinitely often. Hence, there is a BSCC C of $\mathcal{M}_{\mathfrak{S}}$ such that $C \cap B = \varnothing$ and

$$Pr^{\mathfrak{S}}(s \models \exists((\neg B) \,\mathsf{U}\, C)) > 0.$$

Furthermore, $Pr^{\mathfrak{S}}(t \models \Box\neg B) = 1$ for all states $t \in C$.

(c) $\Longrightarrow$ (a): Assume $s \models \exists((\neg B) \,\mathsf{U}\, t)$ for some state t such that $Pr^{\mathfrak{S}}(t \models \Box\neg B) = 1$ for some memoryless scheduler $\mathfrak{S}$. Let

$$\widehat{\pi} \;=\; s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n$$

such that $s_0 = s$, $s_i \notin B$ for $0 \leqslant i < n$ and $s_n = t$. Let scheduler $\mathfrak{S}'$ generate this path fragment with positive probability, i.e., $\mathfrak{S}'(s_0 s_1 \ldots s_i) = \alpha_{i+1}$ for $0 \leqslant i < n$. Moreover, $\mathfrak{S}'$ behaves as $\mathfrak{S}$ for all state sequences that extend $s_0 s_1 \ldots s_n$. Then:

$$Pr^{\mathfrak{S}'}(s \models \Box\neg B) \geqslant \prod_{1 \leqslant i \leqslant n} \mathbf{P}(s_{i-1}, \alpha_i, s_i) \;>\; 0$$

and therefore $Pr^{\mathfrak{S}'}(s \models \Diamond B) < 1$. ∎

Lemma 10.111 suggests computing the set $S^{\min}_{=1}$ as the complement of the set of states that can reach T with

$$T \;=\; \{\, s \in S \mid Pr^{\max}(t \models \Box\neg B) = 1 \,\}$$

via a path fragment through $S \setminus B$. The set T can be obtained as follows. Initially, let $A(s) = Act(s)$ for any $s \in S \setminus B$ and $A(t) = \varnothing$ for $t \in B$. Any state $t \in B$ is removed from T. This entails that for all state action pairs $(s, \alpha) \in Pre(t)$ such that $\alpha \in A(s)$ and α is removed from $A(s)$. If in this way $A(s)$ becomes empty for some state $s \in T$, then s is removed from T by the same procedure. That is, all state action pairs $(u, \beta) \in Pre(s)$ with $\beta \in A(u)$ are considered and β is removed from $Act(u)$. This elimination process is repeated as long as there are states s with $A(s) = \varnothing$. This results in a sub-MDP with state space $T \subseteq S \setminus B$ and nonempty action sets $A(t) \subseteq Act(t)$ such that $Post(t, \alpha) \subseteq T$ for all $t \in T$ and $\alpha \in A(t)$.

Clearly, $Pr^{\mathfrak{S}}(t \models \Box\neg B) = 1$ for any state $t \in T$ and any scheduler $\mathfrak{S}$ which only selects actions from $A(t)$ for state sequences $t_0 t_1 \ldots t_n$ with $t_n = t$. Vice versa, by induction on the number of iterations, it can be shown that $Pr^{\max}(s \models \Box\neg B) < 1$ for all states s that have been removed during the elimination process. Hence, indeed:

$$T \;=\; \{\, s \in S \mid Pr^{\max}(t \models \Box\neg B) = 1 \,\}.$$

We conclude that

$$S^{\min}_{=1} \;=\; \{\, s \in S \mid s \not\models \exists(\neg B) \,\mathsf{U}\, T \,\} \;=\; S \setminus Sat(\exists((\neg B) \,\mathsf{U}\, T))$$

and obtain the following theorem:

Theorem 10.112. Qualitative Analysis for Min Reachability Problems

The sets $S^{\min}_{=0}$ and $S^{\min}_{=1}$ can be computed using graph-based algorithms in time $\mathcal{O}(size(\mathcal{M}))$.

Using similar techniques as in Lemma 10.102 (page 852), the above equation system can be used to show the existence of a memoryless scheduler $\mathfrak{S}$ that minimizes the probabilities of reaching B for all states.

Lemma 10.113. Existence of Optimal Memoryless Schedulers

Let $\mathcal{M}$ be a finite MDP, $B \subseteq S$, and $s \in S$. There exists a memoryless scheduler $\mathfrak{S}$ that minimizes the probabilities of eventually reaching B, i.e., for all states s:

$$Pr^{\mathfrak{S}}(s \models \Diamond B) \;=\; Pr^{\min}(s \models \Diamond B).$$

Furthermore, similar to the case for maximal reachability probabilities, the above equation system can be rewritten into a linear program. This linear program is defined by

- If $Pr^{\min}(s \models \Diamond B) = 1$, then $x_s = 1$.
- If $Pr^{\min}(s \models \Diamond B) = 0$, then $x_s = 0$.
- If $0 < Pr(s \models \Diamond B) < 1$, then $0 < x_s < 1$ and for all actions $\alpha \in Act(s)$:

$$x_s \;\leqslant\; \sum_{t \in S} \mathbf{P}(s, \alpha, t) \cdot x_t$$

where $\sum_{s \in S} x_s$ is maximal.

Remark 10.114. Constrained Reachability

The techniques for the events $\Diamond B$ or $\Diamond^{\leqslant n} B$ are also applicable to treat the constrained reachability properties $C \,\mathsf{U}\, B$ or $C \,\mathsf{U}^{\leqslant n} B$. This works as follows. As a first step, all states in $s \in S \setminus (C \cup B)$ are made absorbing. That is, their enabled actions are replaced by a single action, α_s, say, with $\mathbf{P}(s, \alpha_s, s) = 1$. In the thus obtained MDP, $Pr^*(s \models \Diamond B)$ or $Pr^*(s \models \Diamond^{\leqslant n} B)$, respectively, are determined in the described manner. (Here, $*$ is either max or min.)

By Lemmas 10.102 and 10.113, there exists a memoryless scheduler that optimizes (i.e., maximizes or minimizes) the unbounded constrained reachability probabilities $C \,\mathsf{U}\, B$. For

the step-bounded event $C \,\mathsf{U}^{\leqslant n} B$, finite-memory schedulers exist that yield the extreme probabilities. ■

In fact, the techniques sketched in this section provide the key ingredients for PCTL model checking of finite MDPs.

10.6.2 PCTL Model Checking

In Section 10.2, the probabilistic computation tree logic (PCTL) has been introduced as a CTL-like branching time logic for Markov chains. Without changing the syntax, PCTL can also serve as a temporal logic to specify important properties of finite MDPs. The main (and only) difference with the setting for Markov chains is that the probabilistic operator $\mathbb{P}_J(\cdot)$ ranges over *all* schedulers. Thus, $\mathbb{P}_J(\varphi)$ asserts that the probability bounds given by J for the event specified by φ are satisfied, regardless of the resolution of the nondeterminism.

The syntax of PCTL as a logic for MDPs is the same as for Markov chains, see Definition 10.36 on page 781. For an MDP $\mathcal{M} = (S, Act, \mathbf{P}, \iota_{\text{init}}, AP, L)$, the satisfaction relation for PCTL state- and path formulae is defined as follows:

$$
\begin{array}{lll}
s \models \text{true} & & \\
s \models a & \text{iff} & a \in L(s) \\
s \models \neg\Phi & \text{iff} & s \not\models \Phi \\
s \models \Phi_1 \wedge \Phi_2 & \text{iff} & s \models \Phi_1 \text{ and } s \models \Phi_2 \\
s \models \mathbb{P}_J(\varphi) & \text{iff} & \text{for all schedulers } \mathfrak{S} \text{ for } \mathcal{M} : Pr^{\mathfrak{S}}(s \models \varphi) \in J.
\end{array}
$$

Here, $Pr^{\mathfrak{S}}(s \models \varphi)$ is a shorthand for $Pr_s^{\mathfrak{S}}\{\, \pi \in Paths(s) \mid \pi \models \varphi \,\}$. We write $Sat_{\mathcal{M}}(\Phi)$, or briefly $Sat(\Phi)$, for the satisfaction set of Φ in $\mathcal{M}$, i.e.,

$$Sat_{\mathcal{M}}(\Phi) \;=\; \{\, s \in S \mid s \models \Phi \,\}.$$

The semantics of path formulae is exactly the same as for PCTL interpreted over Markov chains. Thus, the probabilistic operator $\mathbb{P}_J(\cdot)$ imposes probability bounds for *all* schedulers. In particular, we have

- $s \models \mathbb{P}_{\leqslant p}(\varphi)$ if and only if $Pr^{\max}(s \models \varphi) \leqslant p$, and
- $s \models \mathbb{P}_{\geqslant p}(\varphi)$ if and only if $Pr^{\min}(s \models \varphi) \geqslant p$

where $Pr^{\max}(s \models \varphi) = \sup_{\mathfrak{S}} Pr^{\mathfrak{S}}(s \models \varphi)$ with $\mathfrak{S}$ ranging over all possible schedulers for $\mathcal{M}$. Similarly, $Pr^{\min}(s \models \varphi)$ denotes the infimum of the probability for the event specified by φ under all schedulers. For finite MDPs, the same holds for strict lower and upper probability bounds (i.e., $< p$ and $> p$), since for any PCTL path formula φ there exists a finite-memory scheduler that maximizes or minimizes the probabilities for φ (see Remark 10.114 on page 865). Thus for finite MDPs:

$$Pr^{\max}(s \models \varphi) = \max_{\mathfrak{S}} Pr^{\mathfrak{S}}(s \models \varphi) \text{ and } Pr^{\min}(s \models \varphi) = \min_{\mathfrak{S}} Pr^{\mathfrak{S}}(s \models \varphi).$$

The always operator can be derived as in the setting of Markov chains. For instance, $\mathbb{P}_{\leqslant p}(\Box\Phi)$ can be defined as $\mathbb{P}_{\geqslant 1-p}(\Diamond\neg\Phi)$. Then:

$$s \models \mathbb{P}_{\leqslant p}(\Box\Phi) \quad \text{iff} \quad Pr^{\mathfrak{S}}(s \models \Box Sat(\Phi)) \leqslant p \text{ for all schedulers } \mathfrak{S}.$$

For example, for the mutual exclusion protocol with a randomized arbiter, the PCTL specification

$$\mathbb{P}_{=1}(\Box(\neg crit_1 \vee \neg crit_2)) \wedge \bigwedge_{i=1,2} \mathbb{P}_{=1}(\,\Box\,(\,wait_i \rightarrow \mathbb{P}_{\geqslant \frac{7}{8}}(\Diamond^{\leqslant 9} crit_i)\,)\,)$$

asserts the mutual exclusion property (first conjunct) and that every waiting process will enter its critical section within the next nine steps with at least probability $\frac{7}{8}$ (regardless of the scheduling policy).

Equivalence of PCTL Formulae To distinguish the equivalence relations on PCTL state formulae that result from the Markov chain semantics and the MDP semantics, we use the notations $\equiv_{\text{MDP}}$ and $\equiv_{\text{MC}}$, respectively. The relation $\equiv_{\text{MDP}}$ denotes the equivalence of PCTL formulae when interpreted over Markov decision processes. That is, $\Phi \equiv_{\text{MDP}} \Psi$ if and only if for all MDPs $\mathcal{M}$, it holds that $Sat_{\mathcal{M}}(\Phi) = Sat_{\mathcal{M}}(\Psi)$. Similarly, $\Phi \equiv_{\text{MC}} \Psi$ if and only if for all Markov chains $\mathcal{M}$, $Sat_{\mathcal{M}}(\Phi) = Sat_{\mathcal{M}}(\Psi)$. Since any Markov chain can be viewed as an MDP where each state has a single enabled action, $\equiv_{\text{MDP}}$ is finer than $\equiv_{\text{MC}}$. That is:

$$\Phi \equiv_{\text{MDP}} \Psi \quad \text{implies} \quad \Phi \equiv_{\text{MC}} \Psi.$$

The converse, however, does not hold. This can be shown as follows. Consider a formula of the form $\mathbb{P}_{\leqslant p}(\varphi)$ with $0 \leqslant p < 1$ and assume φ is a satisfiable, but not valid, path formula such as $\bigcirc a$ or $a \,\mathsf{U}\, b$. Then:

$$\mathbb{P}_{\leqslant p}(\varphi) \equiv_{\text{MC}} \neg\mathbb{P}_{>p}(\varphi).$$

This equivalence is trivial since the probability for an event E in a Markov chain is $\leqslant p$ if and only if the probability for E is not $> p$. An analogous argument is not applicable to Markov decision processes, as universal quantification is inherent in the semantics of the probabilistic operator $\mathbb{P}_J(\cdot)$:

$$\begin{aligned} s \models \mathbb{P}_{\leqslant p}(\varphi) \quad &\text{iff} \quad Pr^{\mathfrak{S}}(s \models \varphi) \leqslant p \text{ for } \textit{all} \text{ schedulers } \mathfrak{S} \\ s \models \neg\mathbb{P}_{>p}(\varphi) \quad &\text{iff} \quad \text{not} \Big(Pr^{\mathfrak{S}}(s \models \varphi) > p \text{ for all schedulers } \mathfrak{S} \Big) \\ &\text{iff} \quad Pr^{\mathfrak{S}}(s \models \varphi) \leqslant p \text{ for } \textit{some} \text{ scheduler } \mathfrak{S} \end{aligned}$$

Hence, $\mathbb{P}_{\leqslant p}(\varphi) \not\equiv_{\text{MDP}} \neg\mathbb{P}_{>p}(\varphi)$. Nonetheless, several equivalences established for MCs also hold for MDPs, such as, e.g.,

$$\mathbb{P}_{]p,q]}(\varphi) \;\equiv_{\text{MDP}}\; \mathbb{P}_{>p}(\varphi) \wedge \mathbb{P}_{\leqslant q}(\varphi).$$

Let us briefly consider the relationship between the qualitative fragment of PCTL and CTL. The qualitative fragment for MDPs contains four qualitative properties built by the probabilistic operators:

$$\mathbb{P}_{=1}(\varphi) \text{ and } \mathbb{P}_{>0}(\varphi) \text{ and } \mathbb{P}_{<1}(\varphi) \text{ and } \mathbb{P}_{=0}(\varphi).$$

As opposed to the setting for Markov chains, these operators cannot be derived from each other. Thus, the syntax of the qualitative fragment of PCTL has to be extended with $\mathbb{P}_{<1}(\varphi)$ and $\mathbb{P}_{=0}(\varphi)$. As for Markov chains, the formulae $\mathbb{P}_{=1}(\Box a)$ and $\forall\Box a$ are equivalent. The same applies to $\mathbb{P}_{>0}(a \,\mathsf{U}\, b)$ and $\exists(a \,\mathsf{U}\, b)$. The qualitative PCTL formulae $\mathbb{P}_{>0}(\Box a)$ and $\mathbb{P}_{=1}(\Diamond a)$ are not definable in CTL and, vice versa, the CTL formulae $\exists\Box a$ and $\forall\Diamond a$ cannot be specified by a qualitative PCTL formula.

PCTL Model Checking Given a finite MDP $\mathcal{M}$ and a PCTL state formula Φ, to check whether all initial states s satisfy Φ, an adapted version of the PCTL model-checking algorithm for Markov chains can be employed. That is, as for other CTL-like branching time logics, one successively computes the satisfaction sets $\mathit{Sat}(\Psi)$ for the state subformulae of Φ. The only difference from PCTL model checking for MCs is the treatment of the probabilistic operator. Consider, e.g., $\Psi = \mathbb{P}_{\leqslant p}(\bigcirc \Psi')$. The model checker determines the values

$$x_s \;=\; Pr^{\max}(s \models \bigcirc \Psi') \;=\; \max\{ \sum_{t \,\in\, \mathit{Sat}(\Psi')} \mathbf{P}(s,\alpha,t) \mid \alpha \in \mathit{Act}(s)\}$$

and returns $\mathit{Sat}(\Psi) = \{\, s \in S \mid x_s \leqslant p \,\}$. For until formulae $\Psi = \mathbb{P}_{\leqslant p}(\Psi_1 \,\mathsf{U}\, \Psi_2)$ and $\Psi = \mathbb{P}_{\leqslant p}(\Psi_1 \,\mathsf{U}^{\leqslant n}\, \Psi_2)$, we apply the techniques explained in the previous section to compute $x_s = Pr^{\max}(s \models C \,\mathsf{U}\, B)$ or $x_s = Pr^{\max}(s \models C \,\mathsf{U}^{\leqslant n} B)$. This can be done by either solving a linear program or by means of value iteration. The model checker returns $\mathit{Sat}(\Psi) = \{\, s \in S \mid x_s \leqslant p \,\}$. The treatment of strict upper probability bounds $< p$ or lower probability bounds ($\geqslant p$ or $> p$) is similar. Formulae $\mathbb{P}_J(\cdot)$ with probability intervals $J \subseteq [0,1]$ different from $[0,p[$, $[0,p]$, $]p,1]$, $[p,1]$ can be treated by splitting $\mathbb{P}_J(\cdot)$ into a conjunction of a formula with an upper bound and a formula with a lower probability bound.

The overall time complexity of this approach is linear in the length of the formula Φ and the maximal step-bound $n_{\max}$ of a subformula $\mathbb{P}_J(\Psi_1 \,\mathsf{U}^{\leqslant n}\, \Psi_2)$ of Φ and polynomial in the size of $\mathcal{M}$, provided that a linear program solver with polynomial worst-case running time is used.

Theorem 10.115. Time Complexity of PCTL Model Checking for MDPs

For finite MDP $\mathcal{M}$ and PCTL formula Φ, the PCTL model-checking problem $\mathcal{M} \models \Phi$ can be determined in time

$$\mathcal{O}(\,\mathrm{poly}(\mathit{size}(\mathcal{M})) \;\cdot\; n_{\max} \cdot |\Phi|\,)$$

where $n_{\max}$ is the maximal step bound that appears in a sub-path formula $\Psi_1 \,\mathsf{U}^{\leqslant n}\, \Psi_2$ of Φ (and $n_{\max} = 1$ if Φ does not contain a step-bounded until operator).

For CTL we explained the concept of witnesses and counterexamples for state formulae $\exists\varphi$ and $\forall\varphi$; see Section 6.6 (page 373). Witnesses and counterexamples have been defined as "sufficiently long" initial finite path fragments $\widehat{\pi}$ that can be extended to a path π where $\pi \models \varphi$ (witness) and $\pi \not\models \varphi$ (counterexample). In the probabilistic case, the situation is analogous, but now memoryless (or finite-memory) schedulers play the role of the finite path fragments. If, e.g., $s \not\models \mathbb{P}_{\leqslant p}(\Psi_1 \,\mathsf{U}\, \Psi_2)$, then $Pr^{\max}(s \models \Psi_1 \,\mathsf{U}\, \Psi_2) > p$ and there exists a memoryless scheduler $\mathfrak{S}$ such that $Pr^{\mathfrak{S}}(s \models \Psi_1 \,\mathsf{U}\, \Psi_2) > p$. Note that such a memoryless scheduler is computed implicitely by the algorithms presented for the computation of $Pr^{\max}(s \models \Psi_1 \,\mathsf{U}\, \Psi_2) > p$. This scheduler $\mathfrak{S}$ can be viewed as a counterexample for $s \not\models \mathbb{P}_{\leqslant p}(\Psi_1 \,\mathsf{U}\, \Psi_2)$. Of course, it suffices to represent only some information on $\mathfrak{S}$, e.g., $\mathfrak{S}$'s decisions on states where $\Psi_1 \wedge \neg\Psi_2$ holds. Counterexamples for other probability bounds ($\geqslant p$, $> p$ or $< p$) and other path formulae can be obtained in an analogous way. Memoryless schedulers are sufficient as long as the step-bounded until operator is not used. For $s \not\models \mathbb{P}_J(\Psi_1 \,\mathsf{U}^{\leqslant n}\, \Psi_2)$ a counterexample can be obtained by a finite-memory scheduler $\mathfrak{S}$ where $Pr^{\mathfrak{S}}(s \models \Psi_1 \,\mathsf{U}^{\leqslant n}\, \Psi_2) \notin J$. The concept of witnesses can be realized for formulae of the form $\neg\mathbb{P}_J(\varphi)$, since they require the existence of a scheduler where the probability for φ is not in J. In fact, any memoryless (or finite-memory) scheduler $\mathfrak{S}$ with $Pr^{\mathfrak{S}}(s \models \varphi) \notin J$ constitutes a proof for $s \models \neg\mathbb{P}_J(\varphi)$. Again, memoryless schedulers are sufficient for PCTL without the step-bounded until operator.

10.6.3 Limiting Properties

Recall that for analyzing finite MCs against liveness properties, the key observation is that almost surely a BSCC will be reached whose states are visited infinitely often; see Theorem 10.27 (page 775). An analogous result can be established for finite MDPs by means of so-called *end components*. An end component is a sub-MDP that is closed under probabilistic choices and that is strongly connected.

Definition 10.116. Sub-MDP

Let $\mathcal{M} = (S, Act, \mathbf{P}, \iota_{\text{init}}, AP, L)$ be a Markov decision process. A *sub-MDP* of $\mathcal{M}$ is a pair (T, A) where $\varnothing \neq T \subseteq S$ and $A : T \to 2^{Act}$ is a function such that:

- $\varnothing \neq A(s) \subseteq Act(s)$ for all states $s \in T$,
- $s \in T$ and $\alpha \in A(s)$ implies $Post(s, \alpha) = \{ t \in S \mid \mathbf{P}(s, \alpha, t) > 0 \} \subseteq T$.

A sub-MDP (T', A') is said to be contained in a sub-MDP (T, A) if $T' \subseteq T$ and $A'(t) \subseteq A(t)$ for all states $t \in T$.

The digraph $G_{(T,A)}$ induced by a sub-MDP (T, A) is the directed graph with the vertex set

$$T \cup \{ \langle s, \alpha \rangle \in T \times Act \mid \alpha \in A(s) \}$$

and the edges $s \to \langle s, \alpha \rangle$ for $s \in S$ and $\alpha \in A(s)$ and $\langle s, \alpha \rangle \to t$ for all $t \in Post(s, \alpha)$. ■

Definition 10.117. End Component

An *end component* of MDP $\mathcal{M}$ is a sub-MDP (T, A) such that the digraph $G_{(T,A)}$ induced by (T, A) is strongly connected.

Let $EC(\mathcal{M})$ denote the set of end components of $\mathcal{M}$. ■

Example 10.118. End Component

Consider the MDP $\mathcal{M}$ of Figure 10.21. The sub-MDP (T, A) with $T = \{ s_5, s_6 \}$ and $A(s_6) = A(s_5) = \{ \alpha \}$ is an end component of $\mathcal{M}$. The digraph $G_{(T,A)} = (V, E)$ where $V = \{ s_5, \langle s_5, \alpha \rangle, s_6, \langle s_6, \alpha \rangle \}$ where $s_5 \to \langle s_5, \alpha \rangle$, $\langle s_5, \alpha \rangle \to s_6$, $s_6 \to \langle s_6, \alpha \rangle$ and $\langle s_6, \alpha \rangle \to s_6$ and $\langle s_6, \alpha \rangle \to s_5$. ■

For each end component (T, A) there is a scheduler, even a finite-memory one, that almost surely enforces staying forever in T while visiting all states in T infinitely often.

Lemma 10.119. Recurrence Property of End Components

For end component (T, A) of finite MDP $\mathcal{M}$, there exists a finite-memory scheduler $\mathfrak{S}$ such that for any $s \in S$:

$$Pr^{\mathfrak{S}}(s \models \Box T \wedge \bigwedge_{t \in T} \Box \Diamond t) \;=\; 1.$$

Proof: The goal is to define an fm-scheduler $\mathfrak{S}$ that schedules all actions $\alpha \in A(s)$ infinitely often, but never an action $\beta \notin A(s)$. For $s \in T$ let

$$A(s) \;=\; \{\, \alpha_0^s, \ldots, \alpha_{k_s-1}^s \,\}.$$

The modes of $\mathfrak{S}$ are the functions $q : T \to \mathbb{N}$ such that $0 \leqslant q(s) < k_s$. The scheduler $\mathfrak{S}$ selects the action in the current state s according to a round-robin policy. That is, if α_i is the last selected action in state s, the next action to be selected is α_{i+1} (modulo k_s). Formally:

$$act(q,s) \;=\; \alpha_i \quad \text{where} \quad i \;=\; q(s) \bmod k_s.$$

The next mode function is given by $\Delta(q,s) \;=\; p$ where $p(t) = q(t)$ for $t \in T \setminus \{\, s \,\}$ and $p(s) \;=\; (q(s)+1) \bmod k_s$. The starting modes are given by $start(\cdot) = q_0$ where $q_0(s) = 0$ for all $s \in T$. The decisions of $\mathfrak{S}$ for states outside T are irrelevant. For instance, $act(q,s) = \alpha_s$, $\Delta(q,s) = q$ for all modes q and states $s \notin T$.

Since $\mathfrak{S}$ only selects actions in $A(s)$ for $s \in T$, all $\mathfrak{S}$-paths that start in a state $s \in T$ never visit a state outside T. As the digraph induced by (T, A) is strongly connected, the (finite) MC induced by $\mathfrak{S}$ consists of a single BSCC. By Theorem 10.27 (page 775), almost surely any state in $\mathcal{M}_{\mathfrak{S}}$ will be visited infinitely often. ■

For an infinite path

$$\pi \;=\; s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots$$

in a finite MDP $\mathcal{M}$, the *limit* of π, denoted $Limit(\pi)$, is defined as the pair (T, A) where T is the set of states that are visited infinitely often in π:

$$T \;=\; \inf(s_0\, s_1\, s_2 \ldots) \;=\; \{\, s \in S \mid \overset{\infty}{\exists}\, n \geqslant 0.\, s_n = s \,\}$$

and $A : T \to Act$, the function that assigns to each state $s \in T$ the set of actions that are taken infinitely often in s, i.e., for $s \in T$:

$$A(s) \;=\; \{\, \alpha \in Act(s) \mid \overset{\infty}{\exists}\, n \geqslant 0.\, s_n = s \,\wedge\, \alpha_{n+1} = \alpha \,\}.$$

The following theorem is the MDP analogue to Theorem 10.27 (page 775). It states that under each scheduler the limit of almost all paths constitutes an end component.

Theorem 10.120. Limiting Behavior of MDPs

For each state s of a finite MDP $\mathcal{M}$ and scheduler $\mathfrak{S}$ for $\mathcal{M}$:

$$Pr_s^{\mathfrak{S}}\{\, \pi \in Paths(s) \mid Limit(\pi) \in EC(\mathcal{M}) \,\} \;=\; 1.$$

Proof: Let $\mathcal{M}$ be a finite MDP and π an infinite path starting in s. Given that $\mathcal{M}$ is finite, there is a state t, say, and an action α, say, such that π visits t and leaves it via α infinitely often. That is, the event $\Box\Diamond(t,\alpha)$ holds. Consider an α-successor u of t, i.e., $p = \mathbf{P}(t,\alpha,u) > 0$. Given $\Box\Diamond(t,\alpha)$, the probability of entering u finitely often is zero. This follows from the fact that the probability of not taking the transition $t \xrightarrow{\alpha} u$ from some moment on is bounded by $\lim_{n\to\infty}(1-p)^n = 0$. Thus, almost surely, u is entered infinitely often via the edge $t \xrightarrow{\alpha} u$. Thus, for each scheduler $\mathfrak{S}$ and almost all $\mathfrak{S}$-paths π it holds that: if $\mathit{Limit}(\pi) = (T, A)$, $t \in T$, $\alpha \in A(t)$ and $\mathbf{P}(t,\alpha,u) > 0$, then $u \in T$. The claim now follows from the fact that the underlying graph of $\mathit{Limit}(\pi)$ is strongly connected. ∎

Let $\mathcal{M}$ be a finite MDP and $P \subseteq (2^{AP})^\omega$ an LT property which only depends on the labelings that are repeated infinitely often. For example, P could stand for a repeated reachability property $\Box\Diamond a$, or Boolean combinations of repeated reachability properties, such as a persistence property $\Diamond\Box b$, a strong fairness condition $\bigwedge_{1\leqslant i\leqslant k}(\Box\Diamond a_i \to \Box\Diamond b_i)$, or a Rabin condition $\bigvee_{1\leqslant i\leqslant k}(\Diamond\Box a_i \wedge \Box\Diamond b_i)$ where a, a_i, b, and b_i are atomic propositions. LT properties P for which the satisfaction only depends on the labelings that appear infinitely often, but not on their order, are called *limit* LT properties. Let $\inf(A_0A_1A_2\ldots) = \{\, A \subseteq AP \mid \forall i \geqslant 0.\exists j \geqslant i.\, A_i = A \,\}$ denote the set of labelings that appear infinitely often in the word $A_0A_1A_2\ldots \in (2^{AP})^\omega$.

Notation 10.121. Limit LT Property

LT property P over AP is a *limit LT property* if for all words σ, $\sigma' \in (2^{AP})^\omega$:

$$\sigma \in P \text{ and } \inf(\sigma) = \inf(\sigma') \quad \text{implies} \quad \sigma' \in P.$$

For limit LT properties, the satisfaction relation for the paths in a finite MDP $\mathcal{M}$ can be expressed by conditions on sets of states. Let S be the state space of $\mathcal{M}$ and $T \subseteq S$. Let

$$T \models P \quad \text{iff} \quad \forall \sigma \in (2^{AP})^\omega.\ \inf(\sigma) = L(T) \ \text{ implies } \ \sigma \in P$$

where $L(T) = \{\, L(t) \in 2^{AP} \mid t \in T \,\}$. If P is given by an LTL formula φ, i.e., $P = \mathit{Words}(\varphi)$, then we write $T \models \varphi$ instead of $T \models P$. ∎

Note that

$$T \models P \quad \text{implies} \quad \text{for all paths } \pi\text{: } \inf(\pi) = T \text{ implies } \mathit{trace}(\pi) \in P.$$

By Theorem 10.120, only the sets $\inf(\pi) = T$ where T is the state set of an end component of $\mathcal{M}$ are relevant when analyzing the probabilities for a limit LT property P. An end components (T, A) is said to be *accepting* (for P) iff $T \models P$.

This permits establishing a reduction from the probabilistic reachability problem to the problem of computing extreme probabilities for limit LT properties. Let U_P denote the union of the sets T of all end components (T, A) of $\mathcal{M}$ such that $T \models P$. The set U_P is also called the *success set* of P in $\mathcal{M}$. Similarly, let V_P be the union of the sets T of all end components (T, A) of $\mathcal{M}$ such that $\neg(T \models P)$.

Theorem 10.122. Verifying Limit LT Properties

Let $\mathcal{M}$ be a finite MDP and P a limit LT property. For any state s of $\mathcal{M}$:

(a) $Pr^{\max}(s \models P) = Pr^{\max}(s \models \Diamond U_P)$, *and*

(b) $Pr^{\min}(s \models P) = 1 - Pr^{\max}(s \models \Diamond V_P)$.

Furthermore, there exist finite-memory schedulers $\mathfrak{S}_{\max}$ and $\mathfrak{S}_{\min}$ such that for any state s of $\mathcal{M}$:

$$(c)\ Pr^{\max}(s \models P) = Pr^{\mathfrak{S}_{\max}}(s \models P) \quad \textit{and} \quad (d)\ Pr^{\min}(s \models P) = Pr^{\mathfrak{S}_{\min}}(s \models P).$$

Proof: First consider the statement for maximal probabilities. For each scheduler $\mathfrak{S}$ we have $Pr^{\mathfrak{S}}(s \models P) \leqslant Pr^{\mathfrak{S}}(s \models \Diamond U_P)$. By Theorem 10.120 it holds that

$$Pr^{\mathfrak{S}}(s \models P) = Pr^{\mathfrak{S}}\{\, \pi \in \mathit{Paths}(s) \mid \mathit{Limit}(\pi) \in \mathit{EC}(\mathcal{M}) \ \wedge \ \inf(\pi) \models P \,\}.$$

By definition of U_P, $\pi \models \Diamond U_P$ for each path π with $\mathit{Limit}(\pi)$ is an end component and $\inf(\pi) \models P$.

Vice versa, there exists a finite-memory scheduler such that $Pr^{\mathfrak{S}}(s \models P) = Pr^{\max}(s \models \Diamond U_P)$. To see this, consider a memoryless scheduler $\mathfrak{S}_0$ that maximizes the probabilities of reaching U_P for all states s in $\mathcal{M}$. In addition, for each end component (T, A) let the fm-scheduler $\mathfrak{S}_{(T,A)}$ be such that it ensures staying forever in T while visiting all states $t \in T$ infinitely often, once started in some state in T. By Lemma 10.119 (page 870), such an fm-scheduler does exist. Furthermore, for each state $u \in U_P$, select an end component $\mathit{EC}(u) = (T, A)$ with $u \in T$ and $T \models P$.

Let $\mathfrak{S}$ be the scheduler which first mimics $\mathfrak{S}_0$ until a state u in U_P has been reached. From this moment on, $\mathfrak{S}$ behaves as $\mathfrak{S}_{\mathit{EC}(u)}$. For this scheduler $\mathfrak{S}$, almost all paths that eventually enter U_P will visit all states of an end component (T, A) with $T \models P$ infinitely

often. In particular, the condition $\inf(\pi) \models P$ holds for almost all $\mathfrak{S}$-paths, provided they eventually reach U_P. This yields

$$\begin{aligned} Pr^{\mathfrak{S}}(s \models P) &= \sum_{u \in U_P} Pr^{\mathfrak{S}_1}(s \models (\neg U_P) \,\mathsf{U}\, u) \cdot \underbrace{Pr^{\mathfrak{S}_2}(u \models P)}_{=1} \\ &= Pr^{\max}(s \models \Diamond U_P). \end{aligned}$$

Since $Pr^{\max}(s \models \Diamond U_P)$ is an upper bound for the probabilities for P and starting state s under all schedulers, this yields the claim.

Statement (b) for minimal probabilities can be derived from (a) using the fact that the class of limit LT properties is closed under negation (i.e., with P also $\overline{P} = (2^{AP})^\omega \setminus P$ is a limit property) and that $Pr^{\mathfrak{S}}(s \models P) = 1 - Pr^{\mathfrak{S}}(s \models \overline{P})$ for all schedulers $\mathfrak{S}$. Hence:

$$Pr^{\min}(s \models P) \;=\; 1 - Pr^{\max}(s \models \overline{P})$$

and any fm-scheduler $\mathfrak{S}$ that maximizes the probabilities for $\overline{P}$ minimizes the probabilities for P. For $T \subseteq S$ we have $T \models \overline{P}$ iff $\neg(T \models P)$. Hence, the set V_P of all states t that are contained in some end component (T, A) with $\neg(T \models P)$ agrees with the set $U_{\overline{P}}$ that arises by the union of all end components (T, A) where $T \models \overline{P}$. But then, (a) applied to $\overline{P}$ yields

$$Pr^{\max}(s \models \overline{P}) \;=\; Pr^{\max}(s \models \Diamond U_{\overline{P}}) \;=\; Pr^{\max}(s \models \Diamond V_P).$$

■

Hence, for finite MDPs the PCTL formula $\mathbb{P}_{=1}(\Box\mathbb{P}_{=1}(\Diamond a))$ asserts that under all schedulers the event $\Box\Diamond a$ holds almost surely. This follows from the fact that

$s \models \mathbb{P}_{=1}(\Box\mathbb{P}_{=1}(\Diamond a))$
iff for all end components (T, A) reachable from s, it holds that $T \cap Sat(a) \neq \varnothing$
iff $s \not\models \exists\Diamond V_P$ with $P = \Box\Diamond a$
iff $Pr^{\min}(s \models \Box\Diamond a) = 1$.

Hence, almost sure repeated reachability is PCTL-definable for finite MDPs.

For certain limit LT properties, memoryless schedulers suffice to provide extreme probabilities. This holds, e.g., for repeated reachability properties $\Box\Diamond B$; see Exercise 10.23 (page 905).

As for reachability properties, graph-based methods suffice to check qualitative limit LT properties. The simplest case is the question whether a limit LT property can be fulfilled with positive probability under some scheduler.

Corollary 10.123. Qualitative Limit LT Properties (Positive Probability)

Let P be a limit LT property and $\mathcal{M}$ a finite MDP and s a state in $\mathcal{M}$. Then, the following statements are equivalent:

(a) $Pr^{\mathfrak{S}}(s \models P) > 0$ *for some scheduler* $\mathfrak{S}$*,*

(b) $Pr^{\max}(s \models P) > 0$*, and*

(c) $s \models \exists \Diamond U_P$*.*

Checking whether $Pr^{\mathfrak{S}}(s \models P) = 1$ for some scheduler $\mathfrak{S}$ (i.e., $Pr^{\max}(s \models P) = 1$) amounts to verifying whether $Pr^{\max}(s \models \Diamond U_P) = 1$. This can be done using the techniques in Section 10.6.1 (see Algorithm 45 on page 859), provided U_P is given. The same holds for qualitative limit LT properties that refer to $Pr^{\min}(s \models P)$. They can be treated either by algorithms to compute V_P in combination with the algorithms for minimal reachability probabilities, or—as shown in the proof of Theorem 10.122 (page 873)—by using the duality of maximal and minimal probabilities for limit LT properties.

The remainder of this section is focused on the computation of the success set U_P for limit LT property P in a finite MDP $\mathcal{M}$. Recall that U_P is the union of the sets T of all end components (T, A) such that $T \models P$. Clearly, U_P results from an analysis of the set of all end components of $\mathcal{M}$. However, the number of end components can be exponential in the size of $\mathcal{M}$. This is due to the fact that end components may overlap, i.e., it is possible to have two end components (T_1, A_1) and (T_2, A_2), say, such that $(T_1, A_1) \neq (T_2, A_2)$ and $T_1 \cap T_2 \neq \varnothing$. However, for certain limit LT properties P, the set U_P can be characterized by means of *maximal* end components, i.e., end components that are not properly contained in any other end component:

Notation 10.124. Maximal end components

An end component (T, A) of a finite MDP $\mathcal{M}$ is called *maximal* if there is no end component (T', A') such that $(T, A) \neq (T', A')$ and $T \subseteq T'$ and $A(s) \subseteq A'(s)$ for all $s \in T$.

Let $MEC(\mathcal{M})$ denote the set of all maximal end components in $\mathcal{M}$. ■

Any end component is contained in exactly one maximal end component. This follows from the fact that the union of end components (T_1, A_1) and (T_2, A_2) with $(T_1, A_1) \neq (T_2, A_2)$ and $T_1 \cap T_2 \neq \varnothing$ is an end component. Here, the union of sub-MDPs (T_1, A_1) and (T_2, A_2) is the sub-MDP $(T_1 \cup T_2, A_1 \cup A_2)$ where $A_1 \cup A_2$ denotes the function $T_1 \cup T_2 \to 2^{Act}$ such that $t \mapsto A_1(t) \cup A_2(t)$ if $t \in T_1 \cap T_2$ and $t \mapsto A_1(t)$ if $t \in T_1 \setminus T_2$ and $t \mapsto A_2(t)$

if $t \in T_2 \setminus T_1$. Furthermore, maximal end components are pairwise disjoint. Hence, the number of maximal end components is bounded above by the number of states in $\mathcal{M}$.

The set U_P for the limit LT property P results from the end components (T, A) where $T \models P$. In case P is a repeated reachability property, say $\Box\Diamond B$ for some $B \subseteq S$, then $T \models P$ is equivalent to the requirement that T contains a B-state. If (T', A') is an end component with $T' \models \Box\Diamond B$, then $T \models \Box\Diamond B$, where (T, A) is the unique maximal end component that contains (T', A'). Therefore, the success set U_P for the event $P = \Box\Diamond B$ arises by the union of all maximal end components (T, A) that contain at least one B-state. Formally:

$$U_{\Box\Diamond B} = \bigcup_{\substack{(T,A) \in MEC(\mathcal{M}) \\ T \cap B \neq \varnothing}} T.$$

Note that an analogous result does not hold for persistence properties $\Diamond\Box B$, since a non-maximal end component (T', A') with $T' \subseteq B$ may be contained in a maximal end component which also contains some states not in B. However, $U_{\Diamond\Box P}$ agrees with the union of all maximal end components (T, A) where $T \subseteq B$ in a slightly modified MDP $\mathcal{M}_{\Box B}$. The MDP $\mathcal{M}_{\Box B}$ results from $\mathcal{M}$ by removing all states $s \notin B$. More precisely, $\mathcal{M}_{\Box B}$ has the state space $B \cup \{ no \}$. The fresh state no is added as a trap state. The transition probabilities are defined by

$$\mathbf{P}_{\Box B}(s, \alpha, t) = \mathbf{P}(s, \alpha, t) \text{ if } \alpha \in Act, s \in B \text{ and } Post(s, \alpha) \subseteq B.$$

Let $\mathbf{P}_{\Box B}(s, \alpha, \cdot) = 0$ for $\alpha \in Act(s)$ such that $Post(s, \alpha) \setminus B \neq \varnothing$. If state s in $\mathcal{M}$ only has transitions to $S \setminus B$, it has no enabled actions left over. For such states s, let $\mathbf{P}_{\Box B}(s, \tau, no) = 1$. Furthermore, $\mathbf{P}_{\Box B}(no, \tau, no) = 1$. Here, τ is a pseudo action that has no further importance. The maximal end components of $\mathcal{M}_{\Box B}$ which do not contain no are end components (T, A) of $\mathcal{M}$ with $T \subseteq B$. And vice versa, any end component (T, A) of $\mathcal{M}$ with $T \subseteq B$ is contained in a maximal end component of $\mathcal{M}_{\Box B}$.

In fact, a similar technique is applicable for Rabin acceptance conditions:

Lemma 10.125. Success Set for Rabin Conditions

Let $\mathcal{M}$ be a finite MDP with state space S, $B_i, C_i \subseteq S$. For a limit LT property P which—when interpreted on $\mathcal{M}$—is given by

$$\bigvee_{1 \leqslant i \leqslant k} (\Diamond\Box B_i \wedge \Box\Diamond C_i),$$

it holds that

$$U_P = \bigcup_{1 \leqslant i \leqslant k} U_{\Box\Diamond C_i}^{\mathcal{M}_{\Box B_i}}$$

where $U_{\Box\Diamond C_i}^{\mathcal{M}_{\Box B_i}}$ *is the success set of the event* $\Box\Diamond C_i$ *in the MDP* $\mathcal{M}_{\Box B_i}$*, i.e., the set of all states* $t \in S$ *that are contained in some maximal end component* (T, A) *of* $\mathcal{M}_{\Box B_i}$ *such that no* $\notin T$ *and* $T \cap C_i \neq \varnothing$*.*

The computation of the maximal end components of a finite MDP $\mathcal{M}$ can be performed by means of iterative computations of SCCs. The idea is to successively remove all states and actions that are not contained in some end component. In the first iteration, the nontrivial SCCs $T_1, \ldots, T_k$ are determined in the underlying graph of $\mathcal{M}$. (A nontrivial SCC is an SCC which contains at least one edge, i.e., a cycle.) Then, for each state $s \in T_i$, any action $\alpha \in Act(s)$ for which $Post(s, \alpha) \setminus T_i \neq \varnothing$, is removed from $Act(s)$. If $Act(s)$ becomes empty, state s is removed together with the actions β from $Act(t)$ with $(t, \beta) \in Pre(s)$. This yields a sub-MDP $\mathcal{M}_1$, say, of $\mathcal{M}$ where the action set $Act(s)$ of each state s solely consists of actions α such that all α-successors of s belong to the SCC T_i of $\mathcal{M}$ with $s \in T_i$.

Due to the removal of actions, however, the strong connectivity of T_i as a vertex set of the underlying graph of $\mathcal{M}_1$ might be lost. We therefore have to repeat the whole procedure. That is, we compute the nontrivial SCCs $T_1^1, \ldots, T_{k_1}^1$ of $\mathcal{M}_1$ and repeat the procedure described above for these SCCs. This yields a sub-MDP $\mathcal{M}_2$, say of $\mathcal{M}_1$. This procedure is repeated until a sub-MDP $\mathcal{M}_i = \mathcal{M}'$ of $\mathcal{M}$ is obtained for which nontrivial SCCs agree with the maximal end components of $\mathcal{M}$.

These steps are summarized in Algorithm 47. Here, if $T \subseteq S$ and $A : S \to 2^{\mathrm{Act}}$ is a function, then $A|_T$ denotes the restriction of A to T, i.e., $A|_T : T \to 2^{\mathrm{Act}}$ is given by $A|_T(t) = A(t)$ for all $t \in T$.

Lemma 10.126. Correctness of Algorithm 47

For finite MDP $\mathcal{M}$ *with state space* S*, Algorithm 47 returns MEC*$(\mathcal{M})$ *and requires at most* $|S|$ *(outermost) iterations.*

Proof: The termination of Algorithm 47 follows from the fact that in each iteration (except the last) of the repeat-loop, the partition induced by MEC is refined and covers at most the elements of the previous iteration. More precisely, let $\mathsf{MEC}_0 = \{ S \}$ and let MEC_i be the set MEC immediately after the ith iteration of the repeat loop. For fixed i, the sets in MEC_i are nonempty and pairwise disjoint. They constitute a partition of some subset S_i of S. The sets S_i are decreasing, i.e., $S_0 \supseteq S_1 \supseteq S_2 \supseteq \ldots$, as for each $i \geqslant 1$ and $T \in \mathsf{MEC}_i$ there exists $U \in \mathsf{MEC}_{i-1}$ such that $T \subseteq U$. Furthermore, if $T = U$, then in the ith iteration no action $\alpha \in A(t)$ has been removed from the action-sets of any state $t \in T$. This follows from the fact that the pairs $(T, A|_T)$ for $T \in \mathsf{MEC}_i$ are sub-MDPs of $\mathcal{M}$, i.e., $Post(t, \alpha) \subseteq T$ for all $\alpha \in A(t)$ and $t \in T$. Since the repeat loop terminates as soon as

Algorithm 47 Computing the maximal end components of a finite MDP

Input: finite MDP $\mathcal{M}$ with state space S
Output: the set $MEC(\mathcal{M})$

```
for all s ∈ S do A(s) := Act(s); od
MEC := ∅;   MEC_new := { S };

repeat
  MEC := MEC_new;   MEC_new := ∅;
  for all T ∈ MEC do
    R := ∅;                                          (* set of states to be removed *)
    compute the nontrivial SCCs T_1, ..., T_k of the digraph G_(T,A|_T);
    for i = 1, ..., k do
      for all states s ∈ T_i do
        A(s) := { α ∈ A(s) | Post(s, α) ⊆ T_i };
        if A(s) = ∅ then
          R := R ∪ { s };
        fi
      od
    od
    while R ≠ ∅ do
      let s ∈ R
      remove s from R and from T;
      for all (t, β) ∈ Pre(s) with t ∈ T do
        A(t) := A(t) \ { β };
        if A(t) = ∅ then
          R := R ∪ { t };
        fi
      od
    od
    for i = 1, ..., k do
      if T ∩ T_i ≠ ∅ then
        MEC_new := MEC_new ∪ {T ∩ T_i}
                                   (* (T ∩ T_i, A|_(T∩T_i)) is a sub-MDP of M *)
      fi
    od
  od
until (MEC = MEC_new)

return { (T, A|_T) | T ∈ MEC }
```

$\mathsf{MEC} = \mathsf{MEC}_{\text{new}}$, for all iterations (except the last) there is at least one set $U \in \mathsf{MEC}_{i-1}$ such that some $u \in U$ is removed in the ith iteration.

After $|S|$ iterations, $\mathsf{MEC}_{|S|}$ would consist of singleton sets only and no further refinements would be possible. Hence, the maximal number of iterations of the repeat loop equals $|S|$.

For the (partial) correctness of Algorithm 47, let us first observe that Algorithm 47 never removes states or actions that belong to some end component. That is, whenever (T', A') is an end component of $\mathcal{M}$ then in each iteration of Algorithm 47, there exists $T \in \mathsf{MEC}$ such that (T', A') is contained in $(T, A|_T)$. This is a consequence of the following observation.

Whenever (T, A) is a sub-MDP such that each end component (T', A') of $\mathcal{M}$ with $T \cap T' \neq \varnothing$ is contained in (T, A), then:

- For SCC C in $G_{(T,A)}$ it holds that for any state $s \in T$ and $\alpha \in A(s)$ such that $\mathit{Post}(s, \alpha) \setminus C \neq \varnothing$, there is no end component (T', A') of $\mathcal{M}$ such that $s \in T'$ and $\alpha \in A'(s)$. In particular, any end component (T', A') of $\mathcal{M}$ with $T' \cap T \neq \varnothing$ is contained in the sub-MDP that results from (T, A) by removing α from $A(s)$.
- If $s \in T$ and $A(s)$ becomes empty by removing all actions, then there is no end component of $\mathcal{M}$ containing s. In particular, any end component (T', A') of $\mathcal{M}$ with $T' \cap T \neq \varnothing$ is contained in the sub-MDP that results from (T, A) by removing state s and all actions β from $A(t)$ where $t \in T$ and $\mathbf{P}(t, \beta, s) > 0$.

Hence, the output of Algorithm 47 is the set of sub-MDPs $(T_1, A_1), \ldots, (T_k, A_k)$ of $\mathcal{M}$ where each end component (T', A') is contained in some (T_i, A_i).

On the other hand, since any (T_i, A_i) remains unchanged in the last iteration of the repeat loop, the graph $G_{(T_i,A_i)}$ is strongly connected. Thus, (T_i, A_i) is an end component of $\mathcal{M}$, and therefore a maximal end component of $\mathcal{M}$. Hence, Algorithm 47 returns $\mathit{MEC}(\mathcal{M})$. ■

Let us now consider the worst-case time complexity of Algorithm 47. The SCCs of a digraph with N vertices and M edges can be computed in time $\mathcal{O}(N+M)$. The cost of each iteration of the outermost loop is thus linear in the size of $\mathcal{M}$. The number of iterations is bounded by $|S|$ as shown above. Hence, the worst-case time complexity of Algorithm 47 is quadratic in the size of the MDP. More precisely, it is bounded above by

$$\mathcal{O}(\,|S| \cdot (|S| + M)\,)$$

where M is the number of triples (s, α, t) such that $\mathbf{P}(s, \alpha, t) > 0$. This shows that the success set U_P of a limit LT property P which is given by a Rabin acceptance condi-

tion $\bigvee_{1 \leqslant i \leqslant k} (\Diamond\Box B_i \wedge \Box\Diamond C_i)$ (as in Lemma 10.125 on page 876) can be computed in time $\mathcal{O}(\, size(\mathcal{M})^2 \cdot k \,)$. Thus:

Theorem 10.127. Time Complexity of Verifying Limit Rabin Properties
Let $\mathcal{M}$ be a finite MDP and P a limit LT property specified by a Rabin condition:

$$\bigvee_{0<i\leqslant k} (\, \Diamond\Box B_i \,\wedge\, \Box\Diamond C_i \,).$$

Then: the values $Pr^{\max}(s \models P)$ can be computed in time $\mathcal{O}(\, \mathrm{poly}(\, size(\mathcal{M}) \,) \cdot k \,)$.

By duality, the same holds for limit LT properties P which are given by a strong fairness condition

$$\bigwedge_{1\leqslant i\leqslant k} (\, \Box\Diamond C_i \,\rightarrow\, \Box\Diamond D_i \,) \;\equiv\; \neg \bigvee_{1\leqslant i\leqslant k} (\, \Diamond\Box\neg D_i \,\wedge\, \Box\Diamond C_i \,)$$

and the values $Pr^{\min}(s \models P)$. Recall that by part (b) of Theorem 10.122 (page 873), we have:

$$Pr^{\min}(s \models P) \;=\; 1 \,-\, Pr^{\max}(s \models \Diamond V_P)$$

where V_P is the union of the sets T of all end components (T, A) such that $\neg(T \models P)$. For a strong fairness condition as above we have $\neg(T \models P)$ if and only if there exists i such that $T \cap C_i \neq \varnothing$ and $T \cap B_i = \varnothing$. Hence, V_P arises as the union of the maximal end components (T, A) of $\mathcal{M}_{\Box\neg B_i}$ where $T \cap C_i \neq \varnothing$. Here, $\mathcal{M}_{\Box\neg B_i}$ is the MDP obtained from $\mathcal{M}$ by removing the states $s \in S \setminus B_i$ (see page 876 for the precise definition).

10.6.4 Linear-Time Properties and PCTL*

This section treats the quantitative verification of ω-regular property P against an MDP $\mathcal{M}$. This entails the computation of the values $Pr^{\min}(s \models P)$ or, dually, $Pr^{\max}(s \models P)$ for state s in $\mathcal{M}$. For example, let P describe the "good" behaviors and assume it is required to establish whether P holds in $\mathcal{M}$ under all schedulers with some sufficiently large probability $1 - \varepsilon$, say. More precisely, the requirement is to show that

$$\sum_{s\in S} \iota_{\mathrm{init}}(s) \cdot Pr^{\min}(s \models P) \;\geqslant\; 1 - \varepsilon.$$

Similarly, if P describes the bad behaviors, then a reasonable requirement is to verify whether P holds with some sufficiently small probability at most ε under all schedulers. That is:

$$\sum_{s\in S} \iota_{\mathrm{init}}(s) \cdot Pr^{\max}(s \models P) \;\leqslant\; \varepsilon.$$

The previous sections covered special cases of this setting such as reachability properties. For limit LT property P given by a Rabin condition, the values $Pr^{\max}(s \models P)$ can be obtained by computing the values $Pr^{\max}(s \models \Diamond U_P)$. The set U_P is determined using graph algorithms, and the reachability probabilities $Pr^{\max}(s \models \Diamond U_P)$ are obtained by (again) graph analysis followed by solving linear programs. This technique for limit LT properties can be generalized toward arbitrary ω-regular properties in the following way.

Let P be an arbitrary ω-regular property. As a first step, a deterministic Rabin automaton $\mathcal{A}$ for P is constructed. Likewise as for Markov chains, the idea is to reduce the problem of computing $Pr^{\min}(s \models P)$ (or its dual) to determining reachability probabilities in a product MDP. To that end, the product MDP $\mathcal{M} \otimes \mathcal{A}$ is considered (defined just below) and the maximal probabilities of reaching the success set $U_{\mathcal{A}}$ in $\mathcal{M} \otimes \mathcal{A}$ are computed. The success set $U_{\mathcal{A}}$ depends on the Rabin acceptance condition in $\mathcal{A}$.

Notation 10.128. Product MDP

For finite MDP $\mathcal{M} = (S, Act, \mathbf{P}, \iota_{\text{init}}, AP, L)$ and DRA $\mathcal{A} = (Q, 2^{AP}, \delta, q_0, Acc)$ with $Acc = \{ (L_1, K_1), \ldots, (L_k, K_k) \}$, the product $\mathcal{M} \otimes \mathcal{A}$ is an MDP

$$\mathcal{M}' = (S \times Q, Act, \mathbf{P}', \iota'_{\text{init}}, Q, L')$$

with

- $\mathbf{P}'(\langle s, q \rangle, \alpha, \langle s', q' \rangle) = \begin{cases} \mathbf{P}(s, \alpha, s') & \text{if } q' = \delta(q, L(s')) \\ 0 & \text{otherwise.} \end{cases}$
- $\iota'_{\text{init}}(\langle s, q \rangle) = \begin{cases} \iota_{\text{init}}(s) & \text{if } q = \delta(q_0, L(s)) \\ 0 & \text{otherwise.} \end{cases}$
- $L'(\langle s, q \rangle) = \{ q \}$.

■

As for the product construction of a Markov chain and a DRA, there is a one-to-one correspondence between the path

$$\pi = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots$$

in the MDP $\mathcal{M}$ and the path

$$\pi^+ = \langle s_0, q_1 \rangle \xrightarrow{\alpha_1} \langle s_1, q_2 \rangle \xrightarrow{\alpha_2} \langle s_2, q_3 \rangle \xrightarrow{\alpha_3} \ldots$$

in $\mathcal{M} \otimes \mathcal{A}$ that starts in state $\langle s_0, q_1 \rangle$ where $q_1 = \delta(q_0, L(s_0))$. Given a path π^+ in $\mathcal{M} \otimes \mathcal{A}$ the corresponding path in $\mathcal{M}$ is simply obtained by omitting all automata states q_i. Vice versa, given a path π as above, the corresponding path π^+ is obtained by adding the automata states $q_{i+1} = \delta(q_i, L(s_i))$ to π. Thus, π^+ arises by combining π with the unique run for $\mathit{trace}(\pi)$ in the DRA $\mathcal{A}$. In particular:

$$\begin{array}{ll} & \mathit{trace}(\pi) \in P = \mathcal{L}_\omega(\mathcal{A}) \\ \text{iff} & \text{the run } q_0\, q_1\, q_2\, q_3 \ldots \text{ for } \mathit{trace}(\pi) \text{ in } \mathcal{A} \text{ is accepting} \\ \text{iff} & \pi^+ \models \bigvee\limits_{0<i\leqslant k} (\, \Diamond\Box\neg L_i \wedge \Box\Diamond K_i \,). \end{array}$$

In fact, this one-to-one correspondence on the path level induces a one-to-one correspondence for schedulers on $\mathcal{M}$ and $\mathcal{M} \otimes \mathcal{A}$. That is, any scheduler $\mathfrak{S}$ for $\mathcal{M}$ induces a scheduler $\mathfrak{S}'$ for $\mathcal{M} \otimes \mathcal{A}$ such that for any $\mathfrak{S}$-path π in $\mathcal{M}$ the corresponding path π^+ in $\mathcal{M} \otimes \mathcal{A}$ is a $\mathfrak{S}'$-path, and vice versa. The scheduler $\mathfrak{S}'$ is obtained by simply ignoring the automata states, i.e.:

$$\mathfrak{S}'(\langle s_0, q_1 \rangle \langle s_1, q_2 \rangle \ldots \langle s_n, q_{n+1} \rangle) \;=\; \mathfrak{S}(s_0\, s_1 \ldots s_n).$$

We then have

$$Pr^{\mathfrak{S}}(s \models P) \;=\; Pr^{\mathfrak{S}'}\Big(\langle s, \delta(q_0, L(s)) \rangle \models \bigvee_{0<i\leqslant k} (\, \Diamond\Box\neg L_i \wedge \Box\Diamond K_i \,) \Big).$$

Vice versa, for a given scheduler $\mathfrak{S}'$ for $\mathcal{M} \otimes \mathcal{A}$, the corresponding scheduler $\mathfrak{S}$ for $\mathcal{M}$ is obtained as follows. For history $s_0\, s_1 \ldots s_n$ in $\mathcal{M}$, scheduler $\mathfrak{S}$ chooses the same action as $\mathfrak{S}'$ selects for the history $\langle s_0, q_1 \rangle \ldots \langle s_n, q_{n+1} \rangle$ in $\mathcal{M} \otimes \mathcal{A}$ (where $q_{i+1} = \delta(q_i, L(s_i))$ for $0 \leqslant i \leqslant n$).

Thus, there is a one-to-one-correspondence between the schedulers for $\mathcal{M}$ and $\mathcal{M} \otimes \mathcal{A}$. This correspondence preserves the finite-memory property—if $\mathfrak{S}$ is finite memory, then $\mathfrak{S}'$ is also. In addition, the probabilities for P (in $\mathcal{M}$) under $\mathfrak{S}$ agree with the probabilities for $\mathcal{A}$'s acceptance condition $\bigvee_{1\leqslant i\leqslant k} (\, \Diamond\Box\neg L_i \wedge \Box\Diamond K_i \,)$ under $\mathfrak{S}'$. But then, for all states s in $\mathcal{M}$:

$$\begin{array}{rcl} Pr^{\max}_{\mathcal{M}}(s \models P) & = & Pr^{\max}_{\mathcal{M}\otimes\mathcal{A}}(\, \langle s, \delta(q_0, L(s)) \rangle \models \bigvee\limits_{1\leqslant i\leqslant k} (\, \Diamond\Box\neg L_i \wedge \Box\Diamond K_i \,)\,) \\ & = & Pr^{\max}_{\mathcal{M}\otimes\mathcal{A}}(\, \langle s, \delta(q_0, L(s)) \rangle \models \Diamond U_{\mathcal{A}} \,) \end{array}$$

where $U_{\mathcal{A}}$ denotes the success set of $\mathcal{A}$'s acceptance condition $\bigvee_{1\leqslant i\leqslant k} (\, \Diamond\Box\neg L_i \wedge \Box\Diamond K_i \,)$. Hence, the techniques presented in the previous section can be applied to compute $Pr^{\max}(s \models P)$. For the special case where the aim is to check whether $Pr^{\max}(s \models P) > 0$, a graph analysis in $\mathcal{M} \otimes \mathcal{A}$ suffices that checks whether $\langle s, \delta(q_0, L(s)) \rangle$ can reach the success set $U_{\mathcal{A}}$ in $\mathcal{M} \otimes \mathcal{A}$.

The same techniques can be used to determine $Pr^{\min}(s \models P)$ by constructing a DRA for the complement $\overline{P}$ of P, i.e., $\overline{P} = (2^{AP})^{\omega} \setminus P$. As the class of ω-regular properties is closed under complementation, $\overline{P}$ is an ω-regular property. Then:

$$Pr^{\min}(s \models P) \;=\; 1 - Pr^{\max}(s \models \overline{P}).$$

In case the ω-regular property P is given as LTL formula φ, the worst-case time complexity of this technique to compute $Pr^{\max}(s \models \varphi)$ or $Pr^{\min}(s \models \varphi)$ is polynomial in the size of $\mathcal{M}$, but *double exponential* in the length of φ. (The double-exponential blowup is caused by the transformation from LTL to DRA.) From the complexity-theoretic point of view this algorithm is optimal, since the qualitative model-checking problem for MDPs—given a finite MDP $\mathcal{M}$ and an LTL formula φ is $Pr^{\max}(s \models \varphi) = 1$—is in 2EXPTIME. This result is due to Courcoubetis and Yannakakis and stated here without proof [104].

Theorem 10.129.

The qualitative model-checking problem for finite MDPs is in 2EXPTIME.

Recall that in the setting of Markov chains, this problem is PSPACE-complete.

As for Markov chains, the PCTL model-checking algorithm can be extended to treat the logic PCTL* by using the above techniques for computing extreme probabilities for LTL formulae. For PCTL* state formulae $\mathbb{P}_{\leqslant p}(\varphi)$, one first recursively computes the satisfaction sets of the maximal state subformulae of φ. These maximal state subformulae are replaced by new atomic propositions. This yields an LTL formula φ', say. Subsequently, construct a DRA for φ' and apply a quantitative analysis to compute $Pr^{\max}(s \models \varphi')$ for all states s in the MDP $\mathcal{M}$. Then, $Sat(\mathbb{P}_{\leqslant p}(\varphi))$ is the set of all states s in $\mathcal{M}$ where $Pr^{\max}(s \models \varphi') \leqslant p$. The treatment of strict upper bounds $< p$ is similar. For lower bounds $\geqslant p$ or $> p$, we have to compute $Pr^{\min}(s \models \varphi')$ for all states s in $\mathcal{M}$. This yields a PCTL* model checking procedure that runs in time polynomial in the size of the MDP and double exponential in the length of the input PCTL* state formula.

10.6.5 Fairness

This chapter is completed by discussing fairness assumptions in MDPs. Let us first remark that for each scheduler $\mathfrak{S}$, almost all paths that visit state s infinitely often and take action α in s infinitely often will visit each α-successor of s infinitely often. This is similar to the setting of Markov chains, cf. Theorem 10.25 (page 772). (This fact has been already used in the proof of Theorem 10.120.) Thus, probabilistic choices are almost surely strongly fair. However, this does not address the resolution of the nondeterministic choices. As for

transition systems, fairness assumptions for the resolution of the nondeterministic choices are often necessary to establish liveness properties. This typically applies to distributed systems modeled by an MDP that relies on an *interleaving semantics*, and where (process) fairness simply serves to rule out unrealistic behaviors where certain processes eventually stop their execution without having reached a terminal state.

Consider, e.g., the simple mutual exclusion protocol with a randomized arbiter (cf. Example 10.83, page 835). In the absence of any fairness assumption, this protocol cannot guarantee that each process eventually enters its critical section almost surely. For instance, a scheduler which only selects the actions of the second process while completely ignoring the first one, is not excluded. Similarly, for the randomized dining philosophers algorithm (Example 10.87 on page 839), one cannot guarantee that each philosopher eats infinitely often almost surely, as there exist schedulers which only select the actions of one of the philosophers and never an action from one of the other philosophers.

For MDPs, fairness assumptions on the resolution of the nondeterministic choices are constraints on the schedulers. Instead of ranging over all schedulers, only the schedulers that generate fair paths are considered and are taken into account for the analysis. The underlying notion of fairness for paths is as for transition systems. In the sequel, it is assumed that fairness constraints are given as LTL fairness assumptions, i.e., conjunctions of unconditional fairness assumptions $\Box\Diamond\Psi$, strong fairness assumptions $\Box\Diamond\Phi \rightarrow \Box\Diamond\Psi$, and weak fairness assumptions $\Diamond\Box\Phi \rightarrow \Box\Diamond\Psi$. Here, Φ and Ψ are propositional logic formulae. A scheduler is fair if it almost surely generates fair paths.

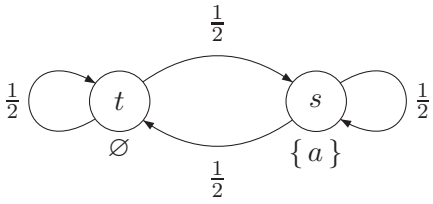
Definition 10.130. Fair Scheduler

Let $\mathcal{M}$ be a Markov decision process and *fair* an LTL fairness assumption. A scheduler $\mathfrak{F}$ for $\mathcal{M}$ is *fair* (with respect to *fair*) if for each state s of $\mathcal{M}$

$$Pr_s^{\mathfrak{F}}\{\, \pi \in Paths(s) \mid \pi \models \mathit{fair} \,\} \;=\; 1.$$

The fairness assumption *fair* is *realizable* in $\mathcal{M}$ if there exists some *fair* scheduler for $\mathcal{M}$. ■

Without any additional assumptions, fair schedulers need not exist. This is evident when there are no paths satisfying *fair*. But even if each finite path fragment can be extended to a path satisfying *fair*, the existence of fair schedulers is not guaranteed. For example, consider the following Markov chain:



Let the strong fairness assumption *fair* be defined as $\Box\Diamond a \rightarrow \Box\Diamond b$. Each path fragment can be extended to a fair path, since there is always the possibility of eventually entering state t and staying there forever (and violating $\Box\Diamond a$). On the other hand, when $\mathcal{M}$ is considered as an MDP (labeling each transition with α, say), then $\mathcal{M}$ has just one (memoryless) scheduler $\mathfrak{S}$, viz. one that in each state always selects α. But $\mathfrak{S}$ is not fair since almost surely both states s and t will be visited infinitely often. Hence, $\Box\Diamond a \wedge \Box\neg b$ holds almost surely.

In the sequel, we require the *realizability* of *fair* in the MDP $\mathcal{M}$ under consideration. As LTL fairness assumptions constitute limit LT properties, for finite MDPs realizability is equivalent to the existence of a *finite-memory fair* scheduler (see Exercise 10.28, page 907).

The first observation is that realizable fairness assumptions are irrelevant for maximal reachability probabilities:

Lemma 10.131. Fairness and Max Reachability Probabilities

Let $\mathcal{M}$ be a finite MDP with state space S, $B, C \subseteq S$ and fair a realizable fairness assumption for $\mathcal{M}$. For each state s of $\mathcal{M}$:

$$\sup_{\substack{\mathfrak{F} \text{ fair} \\ \text{scheduler for } \mathcal{M}}} Pr^{\mathfrak{F}}(s \models C \cup B) \;=\; Pr^{\max}(s \models C \cup B).$$

Furthermore, there exists a finite-memory fair scheduler that maximizes the probabilities for $C \cup B$.

Proof: Let $\mathcal{M}$ be a finite MDP with state space S, $B, C \subseteq S$ and *fair* a realizable fairness assumption for $\mathcal{M}$. For $s \in S$, it holds that

$$\sup_{\substack{\mathfrak{F} \text{ fair} \\ \text{scheduler for } \mathcal{M}}} Pr^{\mathfrak{F}}(s \models C \cup B) \;\leqslant\; \sup_{\substack{\mathfrak{S} \text{ arbitrary} \\ \text{scheduler for } \mathcal{M}}} Pr^{\mathfrak{S}}(s \models C \cup B) \;=\; Pr^{\max}(s \models C \cup B)$$

since fair schedulers are more restrictive than arbitrary ones.

The basic idea is now to construct a fair finite-memory scheduler $\mathfrak{G}$ that maximizes the probabilities for the event $C \cup B$ on the basis of a memoryless (possibly unfair) scheduler $\mathfrak{S}$ that maximizes the probabilities for that event (cf. Lemma 10.102 on page 852).

Let Π be the set of all finite path fragments $s_0\, s_1 \ldots s_n$ in the Markov chain $\mathcal{M}_{\mathfrak{S}}$ such that $s_n \in B$ and $s_i \in C \setminus B$ for $0 \leqslant i < n$. Furthermore, since *fair* is realizable and $\mathcal{M}$ is finite, there exists a fair finite-memory scheduler $\mathfrak{F}$ for $\mathcal{M}$.

The fair scheduler $\mathfrak{G}$ that maximizes the probabilities for $C \cup B$ is now defined as follows on the basis of $\mathfrak{S}$ and $\mathfrak{F}$. $\mathfrak{G}$ behaves as the memoryless scheduler $\mathfrak{S}$ for all histories $s_0\, s_1 \ldots s_n$ that are proper prefixes of some $\widehat{\pi} \in \Pi$. As soon as $\mathfrak{G}$ has generated the path fragment $s_0\, s_1 \ldots s_n \in \Pi$ then $\mathfrak{G}$ continues in a fair way by mimicking the finite-memory scheduler $\mathfrak{F}$ for $\mathcal{M}$. Similarly, once a path fragment $s_0 \ldots s_n$ is generated that is not a prefix of some $\widehat{\pi} \in \Pi$, $\mathfrak{G}$ behaves as $\mathfrak{F}$.

Since the $\mathfrak{S}$-path fragments in Π are path fragments that are also generated by $\mathfrak{G}$, we have

$$Pr^{\mathfrak{G}}(s \models C \cup B) \;=\; Pr^{\mathfrak{S}}(s \models C \cup B) \;=\; Pr^{\max}(s \models C \cup B).$$

Furthermore,

$$Pr_s^{\mathfrak{S}}\{\, \pi \in \mathit{Paths}(s) \mid \mathit{pref}(\pi) \subseteq \mathit{pref}(\Pi) \,\} \;=\; 0$$

where $\mathit{pref}(\pi)$ denotes the set of all finite prefixes of π and $\mathit{pref}(\Pi)$ the set of all finite prefixes of the path fragments in Π. The event "$\mathit{pref}(\pi) \subseteq \mathit{pref}(\Pi)$" means that $\mathfrak{G}$ never stops to mimic $\mathfrak{S}$, and thus might behave unfairly along those paths.

The latter statement follows from the following reasoning. Since the Markov chain $\mathcal{M}_{\mathfrak{S}}$ is finite (as $\mathfrak{S}$ is memoryless), almost surely one of its BSCCs is reached and all its states visited infinitely often. But there is no $\mathfrak{S}$-path π that reaches a BSCC T of $\mathcal{M}_{\mathfrak{S}}$ and visits all states of T infinitely often while fulfilling $\mathit{pref}(\pi) \subseteq \mathit{pref}(\Pi)$. This can be seen as follows. Let $\pi = s_0\, s_1\, s_2 \ldots$ be a $\mathfrak{S}$-path such that $\inf(\pi) = T$ and $\mathit{pref}(\pi) \subseteq \mathit{pref}(\Pi)$. We show that $B \cap T = \varnothing$. This goes by contraposition. Assume $B \cap T \neq \varnothing$. Select a finite prefix $\widehat{\pi} = s_0\, s_1 \ldots s_n$ of π that ends in a state $s_n \in B \cap T$. Then, by definition of Π and since $\widehat{\pi} \in \mathit{pref}(\pi) \subseteq \mathit{pref}(\Pi)$, it follows that $s_0\, s_1 \ldots s_n \in \Pi$. But then none of the prefixes $\widehat{\pi}_m = s_0\, s_1 \ldots s_n\, s_{n+1} \ldots s_m$ of π of length $m > n$ can be extended to a path fragment in Π. That is, $\widehat{\pi}_m \in \mathit{pref}(\pi) \setminus \mathit{pref}(\Pi)$ for all $m > n$. This contradicts the assumption that $\mathit{pref}(\pi) \subseteq \mathit{pref}(\Pi)$.

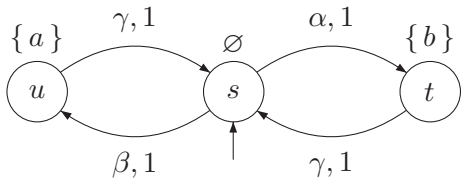
This ensures that almost surely the scheduler $\mathfrak{G}$ will stop to mimic the memoryless scheduler $\mathfrak{S}$ and will behave in a fairly manner from some moment on by simulating $\mathfrak{F}$. This yields that the finite-memory scheduler $\mathfrak{G}$ is fair. ∎

As Lemma 10.131 asserts the existence of a fair scheduler that maximizes the probabilities for $C \cup B$, the supremum in Lemma 10.131 can be replaced with a maximum.

The above result can be understood as the probabilistic counterpart to the fact that

realizable fairness assumptions are irrelevant for verifying safety properties; see Theorem 3.55 (page 140). Recall that a typical usage of $Pr^{\max}(s \models C \,\mathsf{U}\, B)$ is to show that a safety property $a_1 \,\mathsf{W}\, a_2$ holds with some sufficiently large probability $1-\varepsilon$ under all schedulers. (Then C characterizes the states where $a_1 \wedge \neg a_2$ holds, while B represents the states satisfying $\neg a_1 \wedge \neg a_2$.) In this sense, computing $Pr^{\max}(s \models C \,\mathsf{U}\, B)$ can be understood as quantitative reasoning about safety properties.

Due to the previous result, maximal probabilities can be computed without taking fairness assumptions into account. This, however, does not apply to minimal probabilities. Fairness assumptions may be essential when considering the minimal probabilities of reaching a certain set of states B. This is a typical task to show that the liveness property $\Diamond b$ holds with probability $\geqslant 1-\varepsilon$ under all schedulers, for some small ε. For instance, consider the following MDP:



Consider the strong fairness assumption:

$$\mathit{fair} \;=\; \Box\Diamond a \rightarrow \Box\Diamond b,$$

which can be read as $\mathit{fair} = \Box\Diamond u \rightarrow \Box\Diamond t$. All fair schedulers have to take action α in state s infinitely often, and thus, $\Diamond t$ almost surely holds for all fair schedulers. On the other hand, the memoryless scheduler $\mathfrak{S}$ that selects action β for state s never visits t. Hence:

$$Pr^{\min}(s \models \Diamond b) \;=\; 0 \;<\; 1 \;=\; \inf_{\substack{\mathfrak{F} \text{ fair} \\ \text{scheduler for } \mathcal{M}}} Pr^{\mathfrak{F}}(s \models \Diamond b).$$

We now show that the problem of computing the minimal probabilities of reaching B under fair schedulers is reducible to the problem of computing maximal reachability probabilities. Let $\mathcal{M} = (S, \mathit{Act}, \mathbf{P}, \iota_{\text{init}}, AP, L)$ be a finite MDP, $B \subseteq S$ and *fair* a fairness assumption that is realizable in $\mathcal{M}$, i.e., $\mathcal{M}$ is fair with respect to *fair*. Let $F^{\min}_{=0}$ be the set of all states $t \in S$ such that B will never be reached from t for some fair scheduler $\mathfrak{F}$:

$$F^{\min}_{=0} \;=\; \{\, t \in S \mid Pr^{\mathfrak{F}}(t \models \Diamond B) = 0 \text{ for some fair scheduler } \mathfrak{F} \,\}.$$

By the following result, the set $F^{\min}_{=0}$ can be characterized by the end components (T, A) of $\mathcal{M}$ such that $T \models \mathit{fair}$. Notice that the fairness assumption *fair* is a limit LT property (see Notation 10.121 on page 872). Recall that $T \models \mathit{fair}$ means that all paths π with $\inf(\pi) = T$ fulfill *fair*.

Lemma 10.132. Characterization of the Set $F_{=0}^{\min}$

Let $\mathcal{M}$ *be a finite MDP,* $B \subseteq S$*, fair and*

$$F_{=0}^{\min} \;=\; \{\, t \in S \mid Pr^{\mathfrak{F}}(t \models \Diamond B) = 0 \text{ for some fair scheduler } \mathfrak{F} \,\}.$$

For any state $s \in S$*, the following statements are equivalent:*

(a) $s \in F_{=0}^{\min}$*, i.e.,* $Pr^{\mathfrak{F}}(s \models \Diamond B) = 0$ *for some fair scheduler* $\mathfrak{F}$*.*

(b) $Pr^{\mathfrak{F}}(s \models \Diamond B) = 0$ *for some fair, finite-memory scheduler* $\mathfrak{F}$*.*

(c) $Pr^{\max}(s \models (\neg B) \,\mathsf{U}\, V) = 1$ *where* V *is the union of the state sets* T *of all end components* (T, A) *of* $\mathcal{M}$ *such that* $T \cap B = \varnothing$ *and* $T \models$ *fair.*

Proof: (a) $\Longrightarrow$ (c): Suppose $s \in F_{=0}^{\min}$ and consider a fair scheduler $\mathfrak{F}$ with $Pr^{\mathfrak{F}}(s \models \Diamond B) = 0$. The limit of almost all $\mathfrak{F}$-paths is an end component. Let (T, A) be an end component of $\mathcal{M}$ such that $Pr_s^{\mathfrak{F}}(\Pi_{(T,A)}) > 0$ where $\Pi_{(T,A)} = \{\, \pi \in \mathit{Paths}(s) \mid \mathit{Limit}(\pi) = (T, A) \,\}$. Then, $T \cap B = \varnothing$ and $T \models \mathit{fair}$, and therefore $T \subseteq V$. Moreover, all paths $\pi \in \Pi_{(T,A)}$ fulfill $\Box\neg B$, and hence also $(\neg B) \,\mathsf{U}\, V$. Hence:

$$Pr^{\mathfrak{F}}(s \models (\neg B) \,\mathsf{U}\, V) \;=\; 1.$$

In particular, $Pr^{\max}(s \models (\neg B) \,\mathsf{U}\, V) \;=\; 1$.

(c) $\Longrightarrow$ (b): Let $Pr^{\max}(s \models (\neg B) \,\mathsf{U}\, V) \;=\; 1$. Consider a finite-memory scheduler $\mathfrak{S}$ with $Pr^{\mathfrak{S}}(s \models (\neg B) \,\mathsf{U}\, V) \;=\; 1$. By definition of V, there exists a fair finite-memoryless scheduler $\mathfrak{H}$, say, such that $Pr^{\mathfrak{H}}(t \models \Diamond B) = 0$ for any state $t \in V$. (The scheduler $\mathfrak{H}$ can be constructed using the same technique as used in the proof of Theorem 10.122, page 873.) We now compose $\mathfrak{G}$ and $\mathfrak{H}$ to obtain a fair finite-memory scheduler $\mathfrak{F}$ which ensures that B will not be reached from s. For starting state s, scheduler $\mathfrak{F}$ first acts as $\mathfrak{G}$. As soon as V has been reached (which happens almost surely), $\mathfrak{F}$ behaves as $\mathfrak{H}$. It is now clear that $\mathfrak{F}$ is fair (almost surely, eventually $\mathfrak{F}$ simulates the fair scheduler $\mathfrak{H}$) and has the finite-memory property (the union of the modes of $\mathfrak{G}$ and $\mathfrak{H}$ suffices). Moreover:

$$Pr^{\mathfrak{F}}(s \models \Diamond B) \;=\; 1 - Pr^{\mathfrak{G}}(s \models (\neg B) \,\mathsf{U}\, V) \;=\; 0.$$

(b) $\Longrightarrow$ (a): obvious. ∎

The characterization of the set $F_{=0}^{\min}$ enables characterizing the minimal probabilities for the event $\Diamond B$ under fair schedulers by means of maximal probabilities for a constrained reachability property:

Theorem 10.133. Fair Min Reachability Probabilities

Let $\mathcal{M}$ be a finite MDP with state space S, $B \subseteq S$, fair a strong fairness constraint, and $F^{\min}_{=0}$ as above. Then, for any state $s \in S$:

$$\inf_{\substack{\mathfrak{F} \text{ fair} \\ \text{scheduler for } \mathcal{M}}} Pr^{\mathfrak{F}}(s \models \Diamond B) \;=\; 1 - Pr^{\max}(s \models (\neg B) \,\mathsf{U}\, F^{\min}_{=0}).$$

Furthermore, there exists a fair finite-memory scheduler $\mathfrak{F}$ with

$$Pr^{\mathfrak{F}}(s \models \Diamond B) \;=\; 1 - Pr^{\max}(s \models (\neg B) \,\mathsf{U}\, F^{\min}_{=0}).$$

Proof: We first prove the second statement. The proof is by constructing a finite-memory fair scheduler $\mathfrak{G}$ such that for any state s

$$Pr^{\mathfrak{G}}(s \models \Diamond B) \;\leqslant\; 1 - Pr^{\max}(s \models (\neg B) \,\mathsf{U}\, F^{\min}_{=0}).$$

Let $\mathfrak{S}$ be a memoryless scheduler maximizing the probabilities for $(\neg B) \,\mathsf{U}\, F^{\min}_{=0}$; see Lemma 10.102 (page 852). From the proof of Lemma 10.132, it follows that there exists a fair finite-memory scheduler $\mathfrak{H}$ such that $Pr^{\mathfrak{H}}(t \models \Diamond B) = 0$ for each state $t \in F^{\min}_{=0}$. We now combine $\mathfrak{S}$ and $\mathfrak{H}$ to obtain a finite-memory scheduler $\mathfrak{G}$ with the desired properties. In its starting mode, $\mathfrak{G}$ simulates $\mathfrak{S}$ until a path fragment $s_0\, s_1 \ldots s_n$ has been generated such that $s_i \in S \setminus (B \cup F^{\min}_{=0})$ for $0 \leqslant i < n$ and either $s_n \in F^{\min}_{=0}$ or $s_n \in B$. In the former case, $\mathfrak{G}$ switches mode and simulates $\mathfrak{H}$ from now on. In the latter case, $\mathfrak{G}$ behaves from now on as an arbitrary finite-memory fair scheduler. Obviously, $\mathfrak{G}$ is fair and is finite memory, and

$$Pr^{\mathfrak{G}}(s \models (\neg B) \,\mathsf{U}\, F^{\min}_{=0}) \;=\; Pr^{\max}(s \models (\neg B) \,\mathsf{U}\, F^{\min}_{=0}).$$

All $\mathfrak{G}$-paths π with $\pi \models (\neg B) \,\mathsf{U}\, F^{\min}_{=0}$ fulfill $\pi \models \Box(\neg B)$. Hence:

$$\begin{aligned} Pr^{\mathfrak{G}}(s \models \Diamond B) &= 1 - Pr^{\mathfrak{G}}(s \models \Box(\neg B)) \\ &\leqslant 1 - Pr^{\mathfrak{F}}(s \models (\neg B) \,\mathsf{U}\, F^{\min}_{=0}) \\ &= 1 - Pr^{\max}(s \models (\neg B) \,\mathsf{U}\, F^{\min}_{=0}). \end{aligned}$$

This shows the second claim. It remains to show that for each state s of $\mathcal{M}$:

$$\inf_{\substack{\mathfrak{F} \text{ fair} \\ \text{scheduler for } \mathcal{M}}} Pr^{\mathfrak{F}}(s \models \Diamond B) \;\geqslant\; 1 - Pr^{\max}(s \models (\neg B) \,\mathsf{U}\, F^{\min}_{=0}).$$

As for each fair scheduler $\mathfrak{F}$ we have $Pr^{\mathfrak{F}}(s \models \Diamond B) = 1 - Pr^{\mathfrak{F}}(s \models \Box(\neg B))$, it suffices to show that

$$Pr^{\mathfrak{F}}(s \models \Box(\neg B)) \;\leqslant\; Pr^{\mathfrak{F}}(s \models (\neg B) \,\mathsf{U}\, F^{\min}_{=0}).$$

By Theorem 10.120 on page 871, we have

$$\begin{aligned}
& Pr^{\mathfrak{F}}(s \models \Box(\neg B)) \\
&= \sum_{\substack{(T,A) \text{ end comp.} \\ T \cap B = \varnothing}} Pr_s^{\mathfrak{F}}\{\, \pi \in Paths(s) \mid Limit(\pi) = (T,A) \wedge \pi \models \Box(\neg B) \,\} \\
&\leqslant \sum_{\substack{(T,A) \text{ end comp.} \\ T \cap B = \varnothing}} Pr_s^{\mathfrak{F}}\{\, \pi \in Paths(s) \mid Limit(\pi) = (T,A) \wedge \pi \models (\neg B) \,\mathsf{U}\, T \,\}.
\end{aligned}$$

Furthermore, since $\mathfrak{F}$ is fair, for each end component (T,A) such that $Pr_s^{\mathfrak{F}}\{\, \pi \in Paths(s) \mid Limit(\pi) = (T,A) \,\} > 0$, we have $T \models \mathit{fair}$. Lemma 10.132 asserts that the end components (T,A) with $T \models \mathit{fair}$ and $T \cap B = \varnothing$ belong to $F_{=0}^{\min}$. Hence:

$$\begin{aligned}
& Pr^{\mathfrak{F}}(s \models \Box(\neg B)) \\
&\leqslant \sum_{\substack{(T,A) \text{ end comp.} \\ T \cap B = \varnothing,\, T \models \mathit{fair}}} Pr_s^{\mathfrak{F}}\{\, \pi \in Paths(s) \mid Limit(\pi) = (T,A) \ \wedge \ \pi \models (\neg B) \,\mathsf{U}\, T \,\} \\
&\leqslant \sum_{\substack{(T,A) \text{ end comp.} \\ T \cap B = \varnothing,\, T \models \mathit{fair}}} Pr_s^{\mathfrak{F}}\{\, \pi \in Paths(s) \mid Limit(\pi) = (T,A) \ \wedge \ \pi \models (\neg B) \,\mathsf{U}\, F_{=0}^{\min} \,\} \\
&\leqslant \sum_{(T,A) \text{ end comp.}} Pr_s^{\mathfrak{F}}\{\, \pi \in Paths(s) \mid Limit(\pi) = (T,A) \ \wedge \ \pi \models (\neg B) \,\mathsf{U}\, F_{=0}^{\min} \,\} \\
&= Pr^{\mathfrak{F}}(s \models (\neg B) \,\mathsf{U}\, F_{=0}^{\min}).
\end{aligned}$$

Since $Pr^{\mathfrak{F}}(s \models (\neg B) \,\mathsf{U}\, F_{=0}^{\min})$ is bounded above by $Pr^{\max}(s \models (\neg B) \,\mathsf{U}\, F_{=0}^{\min})$, this yields the first claim. ∎

Theorem 10.133 suggests the following recipe to determine the minimal probabilities for reachability properties $\Diamond B$ (for all fair schedulers) of a finite MDP: (1) determine the set $F_{=0}^{\min}$ and (2) solve the linear program for the maximal probabilities for $(\neg B) \,\mathsf{U}\, F_{=0}^{\min}$. The computation of $F_{=0}^{\min}$ relies on a graph analysis of the end components of the MDP.

In case *fair* consists of weak fairness constraints $\Diamond\Box\Phi_j \rightarrow \Box\Diamond\Psi_j$, $j = 1, \ldots, k$, the maximal end component may be exploited. In this case, V is the union of the sets T of all maximal end components (T,A) such that, for each $1 \leqslant j \leqslant k$, $T \cap Sat(\Psi_j) \neq \varnothing$ or $T \setminus \Phi_j \neq \varnothing$. For strong fairness constraints, the analysis of maximal end components might not be sufficient. However, the algorithm for computing maximal end components (Algorithm 47) can be reformulated to compute all end components (T,A) such that $T \models \mathit{fair}$ and (T,A) is not contained in another end component (T',A') where $T' \models \mathit{fair}$. See Exercise 10.29 (page 907).

Constrained reachability properties $C \mathbin{\mathsf{U}} B$ can be treated as simple reachability properties by making all states $s \in S \setminus (C \cup B)$ in the MDP absorbing. Let $\mathcal{M}'$ be the thus obtained MDP. The paths in $\mathcal{M}$ satisfying $C \mathbin{\mathsf{U}} B$ agree with the paths in $\mathcal{M}'$ satisfying $\Diamond B$. Thus:

$$\inf_{\substack{\mathfrak{F} \text{ fair} \\ \text{scheduler for } \mathcal{M}}} Pr^{\mathfrak{F}}_{\mathcal{M}}(s \models C \mathbin{\mathsf{U}} B) \;=\; \inf_{\substack{\mathfrak{F}' \text{ fair} \\ \text{scheduler for } \mathcal{M}'}} Pr^{\mathfrak{F}'}_{\mathcal{M}'}(s \models \Diamond B).$$

The ingredients to treat (constrained) reachability properties under fairness assumptions can be combined with the standard approach to compute satisfaction sets of CTL-like logics to obtain a model-checking algorithm for PCTL under fairness assumptions and finite MDPs. Let $\mathcal{M}$ be a finite MDP and *fair* an LTL fairness assumption that is realizable in $\mathcal{M}$. The satisfaction relation $\models_{fair}$ for PCTL state- and path formulae is defined as the standard satisfaction relation $\models$, except that the probabilistic operator ranges over all fair schedulers (instead of all schedulers). That is:

$$s \models_{fair} \mathbb{P}_J(\varphi) \quad \text{iff} \quad Pr^{\mathfrak{F}}(s \models \varphi) \in J \text{ for all fair schedulers } \mathfrak{F}.$$

The satisfaction set

$$Sat_{fair}(\mathbb{P}_J(\varphi)) = \{\, s \in S \mid s \models_{fair} \mathbb{P}_J(\varphi) \,\}$$

is obtained as follows. For path formulae with the next-step operator, the fairness constraints are irrelevant due to the realizability of *fair*. Thus,

$$Sat_{fair}(\mathbb{P}_J(\bigcirc a)) \;=\; Sat(\mathbb{P}_J(\bigcirc a))$$

for $a \in AP$. For path formulae with the until operator we apply the techniques explained above.

As a next verification problem, we consider the quantitative analysis of finite MDPs against an ω-regular property P in the presence of fairness assumptions. As for the case without fairness assumptions, we adopt the automata-based approach and first construct a deterministic Rabin automaton $\mathcal{A}$ for P. Subsequently, the product-MDP $\mathcal{M} \otimes \mathcal{A}$ is considered; see Notation 10.128 on page 881. Let us first assume that we are interested in the maximal probabilities for P, when ranging over all fair schedulers. By a graph analysis (similar to the technique sketched above), we determine the union V of all end components (T, A) in $\mathcal{M} \otimes \mathcal{A}$ that satisfy the fairness assumption of $\mathcal{M}$ and the acceptance condition of $\mathcal{A}$. Then:

$$\sup_{\substack{\mathfrak{F} \text{ fair} \\ \text{scheduler for } \mathcal{M}}} Pr^{\mathfrak{F}}_{\mathcal{M}}(s \models P) \;=\; Pr^{\max}_{\mathcal{M} \otimes \mathcal{A}}(\langle s, q_s \rangle \models \Diamond V)$$

where $q_s = \delta(q_0, L(s))$. In fact, there is a finite-memory fair scheduler for $\mathcal{M}$ that maximizes the probabilities for P. Hence, the supremum can be replaced by a maximum. Such a finite-memory fair scheduler can be derived from (i) a memoryless scheduler $\mathfrak{S}$ for

$\mathcal{M} \otimes \mathcal{A}$ that maximizes the probabilities of reaching V, and (ii) a finite-memory scheduler $\mathfrak{F}$ for $\mathcal{M} \otimes \mathcal{A}$ which ensures that whenever an end component (T, A) of $\mathcal{M} \otimes \mathcal{A}$ is reached such that $T \models \mathit{fair}$ and T satisfies the acceptance condition of $\mathcal{A}$, then T will never be left and all actions in T are visited infinitely often. These two finite-memory schedulers $\mathfrak{S}$ and $\mathfrak{F}$ for $\mathcal{M} \otimes \mathcal{A}$ can be combined to obtain a finite-memory fair scheduler $\mathfrak{G}$ for $\mathcal{M} \otimes \mathcal{A}$ that maximizes the probabilities for P. A corresponding finite-memory scheduler $\mathfrak{G}$ is obtained by encoding the states in $\mathcal{A}$ in the modes of $\mathfrak{G}$.

To compute minimal probabilities for P under all fair schedulers, we consider the complement property $\overline{P}$ and compute the maximal probabilities for $\overline{P}$ under all fair schedulers. This is sufficient, since

$$\min_{\substack{\mathfrak{F} \text{ fair} \\ \text{scheduler for } \mathcal{M}}} Pr^{\mathfrak{F}}(s \models P) \;=\; 1 - \max_{\substack{\mathfrak{F} \text{ fair} \\ \text{scheduler for } \mathcal{M}}} Pr^{\mathfrak{F}}(s \models \overline{P}).$$

Combining the techniques for model checking PCTL and those for checking ω-regular properties yields a model-checking procedure for PCTL* and finite MDPs under fairness assumptions. The worst-case time complexity is roughly the same for PCTL* model checking of MDPs without fairness. The main difference is that the graph analysis for determining the end components is more advanced and can cause an additional factor $|\mathit{fair}|$ in the cost function.

In LTL and CTL*, fairness assumptions can be encoded syntactically into the formula to be checked. This allows one to reduce the fair satisfaction relation $\models_{\mathit{fair}}$ to the standard satisfaction relation $\models$. In CTL*, e.g., it holds that

$$s \models_{\mathit{fair}} \exists\varphi \quad \text{if and only if} \quad s \models \exists(\mathit{fair} \wedge \varphi)$$

and

$$s \models_{\mathit{fair}} \forall\varphi \quad \text{if and only if} \quad s \models \mathit{fair} \rightarrow \varphi.$$

To conclude this section, we show that an analogous result can also be established for the logic PCTL*. Clearly,

$$s \models \mathbb{P}_{\geqslant p}(\mathit{fair} \rightarrow \varphi) \quad \text{implies} \quad s \models_{\mathit{fair}} \mathbb{P}_{\geqslant p}(\varphi)$$

and

$$s \models \mathbb{P}_{\leqslant p}(\mathit{fair} \wedge \varphi) \quad \text{implies} \quad s \models_{\mathit{fair}} \mathbb{P}_{\leqslant p}(\varphi).$$

The question is whether the reverse implications also hold. At first glance this does not seem to be the case since, e.g., $s \models_{\mathit{fair}} \mathbb{P}_{\geqslant p}(\varphi)$ only concerns the fair schedulers, while $s \models \mathbb{P}_{\geqslant p}(\mathit{fair} \rightarrow \varphi)$ considers all schedulers, including the unfair ones. While schedulers $\mathfrak{S}$ with $0 < Pr^{\mathfrak{S}}(s \models \mathit{fair}) < 1$ are ignored by the fair satisfaction relation $\models_{\mathit{fair}}$, satisfaction $s \models \mathbb{P}_{\geqslant p}(\mathit{fair} \rightarrow \varphi)$ under the standard relation $\models$ without fairness requires

$$Pr^{\mathfrak{S}}(s \not\models \mathit{fair}) \;+\; Pr^{\mathfrak{S}}(s \models \mathit{fair} \wedge \varphi) \geqslant p$$

for all schedulers. In fact, when dropping the realizability assumption, then $s \models_{fair} \mathbb{P}_{\geqslant p}(\varphi)$ and $s \not\models \mathbb{P}_{\geqslant p}(fair \rightarrow \varphi)$ is possible. A simple example is a Markov chain $\mathcal{M}$ (viewed as an MDP) with $Pr(s \models fair) = \frac{1}{2}$ and $Pr(s \models \varphi) = 0$. Since there is no fair scheduler, $s \models_{fair} \mathbb{P}_{=1}(\varphi)$, but $s \not\models \mathbb{P}_{=1}(fair \rightarrow \varphi)$. However, the realizability assumption permits an encoding of the fair satisfaction relation by means of the standard satisfaction relation for finite MDPs. This is stated by the following theorem:

Theorem 10.134. Reduction of $\models_{fair}$ to $\models$

Let $\mathcal{M}$ be a finite MDP and fair an LTL fairness assumption that is realizable for $\mathcal{M}$. Then, for each LTL formula φ and state s of $\mathcal{M}$:

$$\min_{\substack{\mathfrak{F} \text{ fair} \\ \text{scheduler for } \mathcal{M}}} Pr^{\mathfrak{F}}(s \models \varphi) \;=\; Pr^{\min}(s \models fair \rightarrow \varphi)$$

$$\max_{\substack{\mathfrak{F} \text{ fair} \\ \text{scheduler for } \mathcal{M}}} Pr^{\mathfrak{F}}(s \models \varphi) \;=\; Pr^{\max}(s \models fair \wedge \varphi)$$

In particular, $s \models_{fair} \mathbb{P}_{\geqslant p}(\varphi)$ iff $s \models \mathbb{P}_{\geqslant p}(fair \rightarrow \varphi)$ and $s \models_{fair} \mathbb{P}_{\leqslant p}(\varphi)$ iff $s \models \mathbb{P}_{\leqslant p}(fair \wedge \varphi)$. The same holds for strict probability bounds $< p$ and $> p$.

Proof: We first prove the statement about minimal probabilities. For any fair scheduler $\mathfrak{F}$, it holds that $Pr^{\mathfrak{F}}(s \models \varphi) = Pr^{\mathfrak{F}}(s \models fair \rightarrow \varphi)$. Hence, the minimal probability for φ under all fair schedulers is at least the minimal probability for $fair \rightarrow \varphi$ under all schedulers.

We now show that there exists a fair (finite-memory) scheduler such that the probability for φ equals the minimum for $fair \rightarrow \varphi$ under all schedulers. For the sake of simplicity, assume that φ describes a limit LT property. (This is not a restriction, as φ can be replaced by the acceptance condition of the DRA for φ.) Let $\mathfrak{S}$ be a finite-memory (possibly unfair) scheduler that minimizes the probabilities for $fair \rightarrow \varphi$. Thus, $\mathfrak{S}$ maximizes the probabilities for $fair \wedge \neg\varphi$ and:

$$\begin{aligned} Pr^{\max}(s \models fair \wedge \neg\varphi) &= Pr^{\mathfrak{S}}(s \models fair \wedge \neg\varphi) \\ &= \sum_{(T,A)} Pr_s^{\mathfrak{S}}\{\, \pi \in Paths(s) \mid Limit(\pi) = (T,A) \,\} \end{aligned}$$

where (T, A) ranges over all end components of $\mathcal{M}$ such that $T \models fair \wedge \neg(T \models \varphi)$. Since $\mathfrak{S}$ has the finite-memory property, the induced Markov chain $\mathcal{M}_{\mathfrak{S}}$ is finite, and the above sum can be rewritten as

$$\sum_{T} Pr_s^{\mathfrak{S}}\{\, \pi \in Paths(s) \mid \inf(\pi) = T \,\}$$

where T ranges over all BSCCs of $\mathcal{M}_{\mathfrak{S}}$ such that $T \models \mathit{fair} \wedge \neg(T \models \varphi)$. We now consider the finite-memory scheduler $\mathfrak{F}$ that first mimics $\mathfrak{S}$ until $\mathfrak{S}$ reaches a BSCC T. If $T \models \mathit{fair} \wedge \neg(T \models \varphi)$, then $\mathfrak{F}$ behaves forever as $\mathfrak{S}$. If $\neg(T \models \mathit{fair})$ or $T \models \varphi$, then $\mathfrak{F}$ behaves from now on in an arbitrary but fair way (by acting like some fair finite-memory scheduler). $\mathfrak{F}$ is, in fact, a fair scheduler (because $\mathfrak{S}$ reaches almost surely some BSCC). Furthermore:

$$\begin{aligned} Pr^{\mathfrak{F}}(s \models \neg\varphi) &\geqslant \sum_{T} Pr_s^{\mathfrak{S}}\{\, \pi \in Paths(s) \mid \inf(\pi) = T \,\} \\ &= Pr^{\max}(s \models \mathit{fair} \wedge \neg\varphi) \end{aligned}$$

where T ranges over all BSCCs with $T \models \mathit{fair} \wedge \neg(T \models \varphi)$. As $Pr^{\mathfrak{F}}(s \models \neg\varphi) = Pr^{\mathfrak{F}}(s \models \mathit{fair} \wedge \neg\varphi)$, the inequality $\geqslant$ in the above formula can be replaced with an equality $=$. This yields

$$\begin{aligned} Pr^{\mathfrak{F}}(s \models \varphi) &= 1 - Pr^{\mathfrak{F}}(s \models \neg\varphi) \\ &= 1 - Pr^{\mathfrak{F}}(s \models \mathit{fair} \wedge \neg\varphi) \\ &= Pr^{\min}(s \models \mathit{fair} \rightarrow \varphi). \end{aligned}$$

The statement for maximal probabilities follows by duality. Let $\mathfrak{F}$ range over all fair schedulers and $\mathfrak{S}$ over all schedulers. Then:

$$\begin{aligned} \max_{\mathfrak{F}} Pr^{\mathfrak{F}}(s \models \varphi) &= 1 - \min_{\mathfrak{F}} Pr^{\mathfrak{F}}(s \models \neg\varphi) \\ &= 1 - Pr^{\min}(s \models \mathit{fair} \wedge \neg\varphi) \\ &= Pr^{\max}(s \models \mathit{fair} \rightarrow \varphi). \end{aligned}$$

■

As a consequence:

$$s \models_{\mathit{fair}} \mathbb{P}_{=1}(\varphi) \quad \text{if and only if} \quad s \models \mathbb{P}_{=1}(\mathit{fair} \rightarrow \varphi)$$

and

$$s \models_{\mathit{fair}} \mathbb{P}_{=0}(\varphi) \quad \text{if and only if} \quad s \models \mathbb{P}_{=0}(\mathit{fair} \wedge \varphi).$$

That is to say, there exists a fair scheduler $\mathfrak{F}$ with $Pr^{\mathfrak{F}}(\varphi) > 0$ if and only if there exists a scheduler $\mathfrak{S}$ with $Pr^{\mathfrak{S}}(\mathit{fair} \wedge \varphi) > 0$. These reductions emphasize the expressiveness of PCTL* as a logic for reasoning about finite MDPs. However, for efficiency reasons it is recommended to perform a quantitative analysis of finite MDPs with fairness assumptions using the techniques described before.

10.7 Summary

- Markov chains are transition systems with fixed probability distributions over the successors of each state.

- A qualitative property is an event that either holds with probability one or zero. Checking a qualitative property in finite Markov chains can be done by graph algorithms. This does not hold for infinite Markov chains.

- (Constrained) Reachability probabilities can be computed by a graph analysis and solving a linear equation system.

- Almost surely, the long-run behavior of a finite Markov chain ends in a bottom strongly connected component (BSCC). Quantitative (and qualitative) properties about the long-run behavior—such as repeated reachability, persistence, or Boolean combinations thereof—of a finite Markov chain $\mathcal{M}$ can be checked by computing the probability of reaching an accepting BSCC in $\mathcal{M}$.

- Probabilistic Computation Tree Logic (PCTL) is a quantitative variant of CTL where the path quantifiers $\exists$ and $\forall$ are replaced by a probabilistic operator $\mathbb{P}_J(\varphi)$ that specifies lower and/or upper probability bounds (given by J) for the event φ.

- PCTL model checking for finite Markov chains relies on the standard CTL model-checking procedure in combination with methods for computing constrained reachability probabilities.

- The qualitative fragment of PCTL is obtained by only allowing bounds > 0 and $=1$. For finite Markov chains, CTL is at least as expressive as the qualitative fragment of PCTL. For infinite Markov chains, the expressivity of the qualitative fragment of PCTL and CTL is incomparable. As opposed to CTL, persistence properties can be expressed in PCTL (both qualitative and quantitative).

- Computing the probability for an LT property P on a finite Markov chain $\mathcal{M}$ can be reduced to computing the acceptance probability in the product of $\mathcal{M}$ and a deterministic Rabin automaton (DRA) for the complement of P.

- A probabilistic bisimulation equivalence only relates states that are equally labeled and whose cumulative probability of moving to the equivalence classes coincides.

- Probabilistic bisimulation on Markov chains coincides with PCTL and PCTL* equivalence. (The logic PCTL* is obtained by combining PCTL and LTL.) This holds for any arbitrary Markov chain—the restriction to finitely-branching models (as for the logical characterization of bisimulation on transition systems) is not required.

- Long-run averages and expected cost-bounded reachability in Markov reward chains can be determined using graph algorithms and solving linear equation systems.

- Markov decision processes (MDPs) are transition systems in which in any state a nondeterministic choice between probability distributions exists. A Markov chain is an MDP in which for any state the set of probability distributions is a singleton. MDPs are adequate for modeling randomized distributed algorithms.

- Reasoning about probabilities in MDPs requires the concept of schedulers. Schedulers resolve the nondeterminism and yield a stochastic process. Computing extreme (i.e., minimal or maximal) probabilities for constrained reachability properties relies on graph algorithms and linear programs. The latter can be solved by means of an iterative approximation algorithm (called value iteration).

- When interpreting PCTL on MDPs, the formula $\mathbb{P}_J(\varphi)$ ranges over all schedulers. The PCTL model-checking problem for MDPs is reducible to the reachability problem.

- Almost surely, the long-run behavior of a finite MDP ends in one of its end components. Quantitative (and qualitative) properties about the long-run behavior—such as repeated reachability, persistence, or Boolean combinations thereof—of a finite MDP $\mathcal{M}$ can be checked by computing the extreme probability of reaching an "accepting" end component in $\mathcal{M}$.

- Model-checking a finite MDP against an ω-regular property can be solved by an automata-based approach, analogous to finite Markov chains.

- A scheduler is fair if it almost surely generates only fair paths. For maximal reachability properties, fairness is irrelevant. Minimal reachability probabilities under all fair schedulers can be computed by graph-based techniques and determining maximal probabilities for a constrained reachability property. Model checking a finite MDP with fairness assumptions against an ω-regular property can be performed by an automata-based approach similar to that for MDPs without fairness assumptions.

10.8 Bibliographic Notes

Markov chains and Markov decision processes. The main principles of Markov chains as a mathematical model for stochastic processes goes back to Markov in 1906. Since then a variety of different aspects of Markov chains has been studied in the literature such as queuing theory [55], numerical algorithms [63, 378], reliability and performance analysis [196], and lumping [72, 237]. Markov chains have been applied in various areas ranging from systems biology, social sciences, and psychology, to electrical engineering and operations research. Important textbooks on Markov chains are, e.g., [237, 238, 248]. The reader should bear in mind that this monograph treats Markov chains from the state-based view as a graph (transition system) with probabilities, rather than—the more usual interpretation—as a sequence of random variables. Markov reward models are extensively described in the monograph by Howard [216]. The zeroconf protocol example has been adopted from Bohnenkamp et al. [54].

Markov decision processes (MDPs) have their roots in operations research and stochastic control theory. They are useful for studying a wide range of optimization problems. MDPs were known at least as early as in the fifties; see the seminal work by Bellman [38, 39]. MDPs are used in a variety of application areas, including robotics, automated control, economics and in manufacturing. Vardi [407] proposed using MDPs as models for concurrent probabilistic systems. Major textbooks on MDPs are by Puterman [346], Howard [215], and Bertsekas [49]. The randomized philosophers example originates from Lehmann and Rabin [268] and the randomized leader election algorithm of Itai and Rodeh [223]. Other randomized algorithms can be found in, e.g., Rabin [349] or the textbooks by Motwani and Raghavan [306], and Lynch [280].

Verifying qualitative properties. Verification techniques for probabilistic models date back to the early eighties and were focused on qualitative LT properties. Hart, Sharir, and Pnueli [191] observed that graph-based algorithms are sufficient for proving almost sure termination for finite-state concurrent probabilistic programs. The verification of qualitative ω-regular properties for finite MCs and finite MDPs has first been addressed by Vardi and Wolper [407, 409, 411, 412]. They represent ω-regular properties by means of NBAs that are deterministic in the limit. Vardi and Wolper also showed that the qualitative LTL model-checking problem for finite Markov chains is PSPACE-complete. Courcoubetis and Yannakakis [104] extended these results by techniques for the quantitative analysis of MCs against LTL and NBA specifications (see below). They also established a double-exponential lower bound for the problem of verifying whether an LTL formula holds almost surely for a finite MDP. Pnueli and Zuck have developed a tableau-based verification technique for MDPs and qualitative LTL formulae [340] as well as proof methods for MDPs that rely on the connection between probabilistic choice and fairness; see e.g., [341]. A generalization of some of these concepts has been discussed by Baier and Kwiatkowska [32]. Fairness assumptions that impose restrictions on the resolution of nondeterminism in MDPs have been first addressed by Hart, Sharir, and Pnueli [191] and Vardi [407]. The role of fairness in the context of PCTL (and PCTL*) model checking has been discussed by Baier and Kwiatkowska [31].

Verifying quantitative properties. The presented algorithm for the quantitative analysis of MCs against ω-regular specifications exploits deterministic Rabin automata (DRAs). This approach is conceptually simple as the DRA does not affect the transition probabilities in the product Markov chain. Although for many ω-regular properties DRA exist whose size is of the same order as NBA [241], in the worst case the smallest DRA can be exponentially larger than the smallest equivalent NBA. Several alternative algorithms have been presented in the literature that have a time complexity which is polynomial in the size of the Markov chain and exponential in the length of the LTL formula φ. Such algorithms have been proposed in the seminal paper by Courcoubetis and Yannakakis [104], and more recently (using different techniques) by Couvreur, Saheb, and Sutre [108],

and Bustan, Rubin, and Vardi [77]. The observation that the quantitative analysis of MDPs can be solved by linear programs goes back to an earlier paper by Courcoubetis and Yannakakis [103].

Branching-time properties. A branching-time logic for reasoning about probabilistic systems has been originally proposed by Hart and Sharir [190]. Their focus was on qualitative properties and deductive proof rules. Hansson and Jonsson introduced Probabilistic Computation Tree Logic (PCTL) and the PCTL model-checking procedure for finite Markov chains [187]. Variants of PCTL have been proposed for MDPs (or similar models). Hansson [186] and Segala and Lynch [370] presented action-based variants of PCTL. The (state-based) variant of PCTL and PCTL* for MDPs is due to Bianco and de Alfaro [51]. The concept of end components was introduced by Courcoubetis and Yannakakis [104] and has been studied in more detail by de Alfaro [115, 116]. Efficient counterexample generation algorithms for PCTL have recently been proposed by Han and Katoen [185]. A detailed account of (among others) PCTL model checking has been given by Kwiatkowska, Norman, and Parker [254]. Extensions of PCTL-like logics for MDPs (e.g., long-run properties) have been studied by de Alfaro; see e.g., [114, 115, 117]. The logic PCRTL (PCTL with rewards) which, amongst others, supports long-run averages and expected cumulative rewards has been proposed by Andova, Hermanns, and Katoen [14]. They also provide model-checking algorithms for this logic.

Probabilistic bisimulation. Bisimulation for Markov chains and an action-based model á la MDPs has been introduced in the seminal paper by Larsen and Skou [264]. Aziz et al. [23] have shown that bisimulation equivalence on Markov chains preserves the validity of PCTL* formulae. The observation that PCTL$^-$-equivalence agrees with bisimulation equivalence goes back to Desharnais et al. [121]. These authors study this for labeled Markov processes, a probabilistic model with continuous state spaces. Jonsson and Larsen [227] defined simulation relations for probabilistic models. Simulation and bisimulation relations, as well as the use of MDP-like models as semantical model for process algebras, can be found in the survey paper by Jonsson, Yi, and Larsen [228]. Further results on the connection between PCTL/PCTL* and bisimulation and simulation relations for Markov chains have been recently established by Baier et al. [30]. For MDPs (and the like) we refer to the works by Hansson [186], Segala and Lynch [370], and Desharnais and her colleagues [122, 123, 124]. Bisimulation minimization algorithms for MCs have been considered by Huynh and Tian [220], Baier, Engelen, and Majster-Cederbaum [27], and Derisavi, Hermanns, and Sanders [120]. Recently, Katoen et al. [232] have shown experimentally that substantial reductions in both memory and time can be obtained by exploiting bisimulation minimization for PCTL model checking and reward properties.

Probabilistic model checkers. One of the first prototypical PCTL model checkers has been reported by Fredlund [156]. More recent PCTL model checkers are PRISM [255]

and ETMCC [201] (and its successor, MRMC [233]). The former supports both MCs and MDPs, the latter only MCs. PRISM uses a variant of BDDs [26, 181] to enable the compact representation of transition matrices [254, 323]. MRMC is based on a sparse matrix representation and supports minimization techniques for probabilistic bisimulation. Both MRMC/ETMCC and PRISM also support CSL model checking, a continuous-time variant of PCTL proposed by Aziz et al. [22] and Baier et al. [29], and expected measures for Markov reward chains. Other model checkers for Markov chains are ProbVerus [192] and FMurφ [331]. Alternative model checkers for Markov decision processes are LiQuor [82] and Rapture [112, 226]. The latter tool focuses on reachability properties and exploits abstraction-refinement techniques. LiQuor supports the verification of quantitative and qualitative ω-regular properties of MDPs that are modeled using a probabilistic variant of PROMELA, the input language of the LTL model checker SPIN. This variant [25] is based on the probabilistic guarded command language by Morgan and McIver [304]. LiQuor uses partial order reduction techniques to combat the state-space explosion problem [28, 113].

10.9 Exercises

EXERCISE 10.1. Consider the Markov chain $\mathcal{M}$ shown in Figure 10.22. Let $C = \{ s_0, s_1, s_4, s_6 \}$

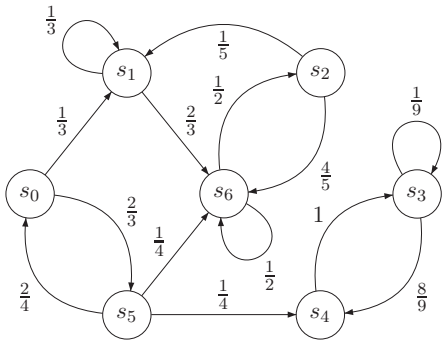


Figure 10.22: Markov chain $\mathcal{M}$ for Exercise 10.1.

and $B = \{ s_2, s_3 \}$.

(a) Compute the probability measure of the union of the following cylinder sets:

$$Cyl(s_0\, s_1),\ Cyl(s_0\, s_5\, s_6),\ Cyl(s_0\, s_5\, s_4\, s_3),\ Cyl(s_0\, s_1\, s_6)$$

given that the initial distribution is given by $i_{init}(s_0) = 1$.

(b) Compute $Pr(s_0 \models \Diamond B)$ using the least fixed point characterization.

(c) Compute $Pr(s_0 \models C \,\mathsf{U}^{\leqslant 5}\, B)$ using:

(i) the least fixed point characterization;

(ii) transient state probabilities.

(d) Determine $Pr(s_0 \models \Diamond\Box D)$ with $D = \{ s_3, s_4 \}$.

EXERCISE 10.2. Consider the Markov chain of Figure 10.23. Let $B_1 = \{ s_1, s_7 \}$, $B_2 = \{ s_6, s_7, s_8 \}$,

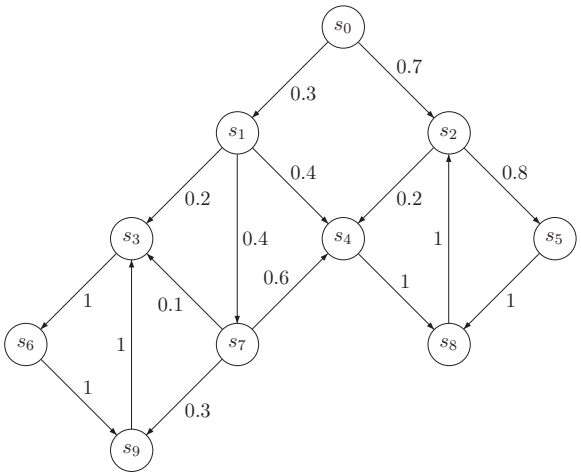


Figure 10.23: Markov chain $\mathcal{M}$ for Exercise 10.2.

$B_3 = \{ s_1, s_3, s_7, s_9 \}$ and $B_4 = \{ s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9 \}$. Determine whether:

(a) $Pr(s_0 \models \Diamond B_1) = 1$,

(b) $Pr(s_7, \Diamond B_2) = 1$,

(c) $Pr(s_0, \Box\Diamond B_i) = 1$, for $i \in \{ 1, 2, 3 \}$,

(d) $Pr(s_0, \Diamond\Box B_2) = 1$, and

(e) $Pr(s_0, \Diamond\Box B_4) = 1$.

EXERCISE 10.3. Let $\mathcal{M} = (S, \mathbf{P}, \iota_{\text{init}}, AP, L)$ be a finite Markov chain, $s \in S$ and $C, B \subseteq S$, $n \in \mathbb{N}$, $n \geqslant 1$. Let $C \,\mathsf{U}^{=n}\, B$ denote the event that a B-state will be entered after exactly n steps and all states that are visited before belong to C. That is, $s_0\, s_1\, s_2 \ldots \models C \,\mathsf{U}^{=n}\, B$ if and only if

$s_n \in B$ and $s_i \in C$ for $0 \leqslant i < n$. The event $C \,\mathsf{U}^{\geqslant n} B$ denotes the union of the events $C \,\mathsf{U}^{=k}\, B$ where k ranges over all natural numbers $\geqslant n$.

Question: Provide an algorithm to compute

(a) the values $Pr(s \models C \,\mathsf{U}^{=n}\, B)$, and

(b) an algorithm for computing $Pr(s \models C \,\mathsf{U}^{\geqslant n}\, B)$.

Exercise 10.4. Let $\mathcal{M} = (S, \mathbf{P}, \iota_{\text{init}}, AP, L)$ be a finite Markov chain and $s \in S$, a, $b \in AP$. Prove or disprove the following statements:

(a) $Pr(s \models \Box a) = 1$ iff $s \models \forall \Box a$

(b) $Pr(s \models \Diamond a) < 1$ iff $s \not\models \forall \Diamond a$

(c) $Pr(s \models \Box a) > 0$ iff $s \models \exists \Box a$

(d) $Pr(s \models \Diamond \Box a) = 1$ iff $Pr(s \models \Diamond B) = 1$ where $B = \{\, t \in S \mid t \models \forall \Box a \,\}$

(e) $Pr(s \models a \,\mathsf{U}\, b) = 1$ iff $s \models \forall (a \,\mathsf{U}\, b)$

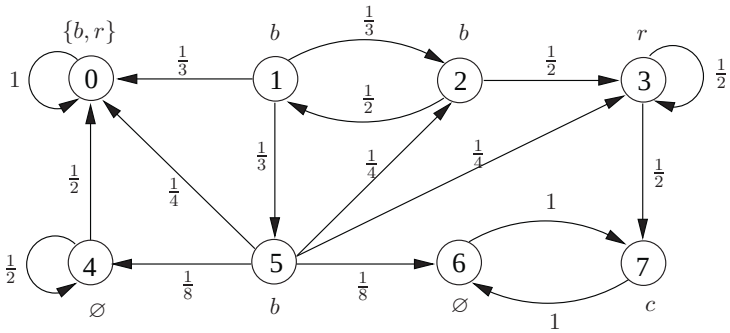
(f) $Pr(s \models a \,\mathsf{U}\, b) = 0$ iff $s \not\models \exists (a \,\mathsf{U}\, b)$

Exercise 10.5. Complete the proof of Corollary 10.33 on page 778 and provide the proof for the following statement:

$$Pr(s \models \Box \Diamond B) \;=\; 1 \quad \text{iff} \quad s \models \forall \Box \exists \Diamond B$$

provided that s is a state of a finite Markov chain $\mathcal{M}$ and B a set of states in $\mathcal{M}$.

Exercise 10.6. Consider the following MC:



Question: Determine the set of states for which the PCTL formula $\mathbb{P}_{\geqslant p}(b \,\mathsf{U}\, c)$ holds for $p = \frac{17}{19}$.

EXERCISE 10.7. Prove or disprove the following PCTL equivalences:

(a) $\mathbb{P}_{=1}(\bigcirc \mathbb{P}_{=1}(\Box a)) \equiv \mathbb{P}_{=1}(\Box \mathbb{P}_{=1}(\bigcirc a))$

(b) $\mathbb{P}_{>0.5}(\bigcirc \mathbb{P}_{>0.5}(\Diamond a)) \equiv \mathbb{P}_{>0.5}(\Diamond \mathbb{P}_{>0.5}(\bigcirc a))$

(c) $\mathbb{P}_{=1}(\bigcirc \mathbb{P}_{=1}(\Diamond a)) \equiv \mathbb{P}_{=1}(\Diamond \mathbb{P}_{=1}(\bigcirc a))$

EXERCISE 10.8. Let $\mathcal{M} = (S, \mathbf{P}, \iota_{\text{init}}, AP, L)$ be a finite Markov chain such that $S \subseteq AP$ and $L(s) \cap AP = \{ s \}$ for each state $s \in S$. Let *sfair* be the following CTL fairness assumption:

$$\mathit{sfair} \;=\; \bigwedge_{s \in S} \; \bigwedge_{t \,\in\, \mathit{Post}(s)} (\Box\Diamond s \rightarrow \Box\Diamond t).$$

Show that for a, $b \in AP$:

(a) $s \models \mathbb{P}_{=1}(a \,\mathsf{U}\, b)$ iff $s \models_{\mathit{sfair}} \forall(a \,\mathsf{U}\, b)$.

(b) $s \models \mathbb{P}_{>0}(\Box a)$ iff $s \models_{\mathit{sfair}} \exists\Box a$.

EXERCISE 10.9. Provide definitions of the weak until operator and the release operator in PCTL. That is, define PCTL formulae $\mathbb{P}_J(\Phi \,\mathsf{W}\, \Psi)$ and $\mathbb{P}_J(\Phi \,\mathsf{R}\, \Psi)$ such that for each Markov chain $\mathcal{M}$ and each state s in $\mathcal{M}$:

$$s \models \mathbb{P}_J(\Phi \,\mathsf{W}\, \Psi) \quad \text{iff} \quad \mathit{Pr}(s \models \Phi \,\mathsf{W}\, \Psi) \in J,$$
$$s \models \mathbb{P}_J(\Phi \,\mathsf{R}\, \Psi) \quad \text{iff} \quad \mathit{Pr}(s \models \Phi \,\mathsf{R}\, \Psi) \in J,$$

where

$$\begin{aligned} \mathit{Pr}(s \models \Phi \,\mathsf{W}\, \Psi) &= \mathit{Pr}_s^{\mathcal{M}}\{ \pi \in \mathit{Paths}(s) \mid \pi \models \Phi \,\mathsf{U}\, \Psi \vee \pi \models \Box\Phi \}, \\ \mathit{Pr}(s \models \Phi \,\mathsf{R}\, \Psi) &= \mathit{Pr}_s^{\mathcal{M}}\{ \pi \in \mathit{Paths}(s) \mid \pi \not\models \neg\Phi \,\mathsf{U}\, \neg\Psi \}. \end{aligned}$$

EXERCISE 10.10. As for CTL and LTL, PCTL in *positive normal form* (PNF) can be defined as a fragment of PCTL where negation is only allowed adjacent to atomic propositions. To avoid a decrease in expressiveness, the syntax of PCTL formulae in PNF contains for each operator of the base syntax of PCTL a dual operator (e.g., disjunction as the dual for conjunction, release as the dual for until, etc.).

Questions:

(a) Provide the precise definition of the syntax of PCTL formulae, and

(b) Prove that for any PCTL formula there exists an equivalent PCTL formula in PNF.

EXERCISE 10.11. Let $\mathcal{M}$ be a finite Markov chain over AP, s a state of $\mathcal{M}$ and $a, b \in AP$: Show that

$$s \models \mathbb{P}_{=1}(a \,\mathsf{U}\, b) \quad \text{iff} \quad s \models \forall(\,(\exists(a \,\mathsf{U}\, b)) \,\mathsf{W}\, b\,).$$

EXERCISE 10.12. Provide deterministic Rabin automata for the following LTL formulae: $\Box(a \rightarrow \Diamond b)$, $\neg\Box(a \rightarrow \Diamond b)$, and $a \,\mathsf{U}\, (\Box b)$.

EXERCISE 10.13. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be DRA over the same alphabet. Define the DRA $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$ such that

$$\mathcal{L}_\omega(\mathcal{A}) = \mathcal{L}_\omega(\mathcal{A}_1) \cup \mathcal{L}_\omega(\mathcal{A}_2)$$

and $\mathit{size}(\mathcal{A}) = \mathcal{O}\big(\text{poly}(\mathit{size}(\mathcal{A}_1), \mathit{size}(\mathcal{A}_2))\big)$.

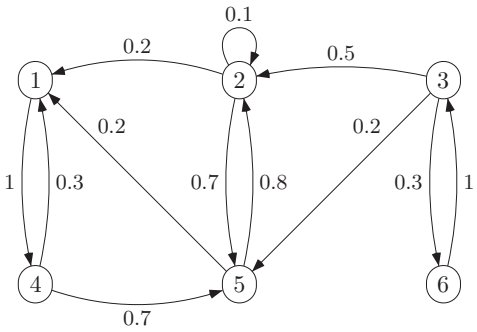
EXERCISE 10.14. Consider the Markov chain $\mathcal{M}$ in Figure 10.22 (page 899), and let the labeling be given by $L(s_2) = L(s_3) = L(s_4) = \{\, a \,\}$ and $L(s) = \varnothing$ for the remaining states. Question: Compute the probability $Pr^{\mathcal{M}}(\varphi)$ for the LTL formula $\varphi = \Box\Diamond\, a$. (*Hint: Construct a DRA $\mathcal{A}$ for φ and perform a quantitative analysis in $\mathcal{M} \otimes \mathcal{A}$.*)

EXERCISE 10.15. Prove the following statement. For finite Markov chain $\mathcal{M} = (S, \mathbf{P}, \iota_{\text{init}}, AP, L)$ and ω-regular property P over AP such that $Pr^{\mathcal{M}}(P) > 0$, there exists a finite path fragment $\widehat{\pi} = s_0\, s_1 \ldots s_n$ in $\mathcal{M}$ with $\iota_{\text{init}}(s_0) > 0$ such that almost all paths in the cylinder set $\mathit{Cyl}(\widehat{\pi})$ fulfill P, i.e.,

$$Pr^{\mathcal{M}}\{\pi \in \mathit{Cyl}(\widehat{\pi}) \mid \mathit{trace}(\pi) \in P\} = \mathbf{P}(\widehat{\pi}).$$

EXERCISE 10.16. Show that there is no PCTL formula that is equivalent to the PCTL* formula $\mathbb{P}_{\geqslant 0.5}(\bigcirc\bigcirc\, a)$ where a is an atomic proposition.

EXERCISE 10.17. Consider the Markov chain $\mathcal{M}$, which is given by



Questions:

(a) Determine the bisimulation quotient $\mathcal{M}/_{\sim_{\mathcal{M}}}$

(b) For each pair of equivalence classes C and D under $\sim_{\mathcal{M}}$, give a PCTL formula Φ such that $C \models \Phi$ and $D \not\models \Phi$.

Exercise 10.18. Let $(\mathcal{M}, rew)$ be a finite Markov reward model and s a state in $\mathcal{M}$, T a set of states in $\mathcal{M}$ such that $Pr(s \models \Diamond T) = 1$. Show that the infinite series

$$ExpRew(s \models \Diamond T) \;=\; \sum_{r=0}^{\infty} r \cdot Pr_s\{\, \pi \in Paths(s) \mid rew(\pi, \Diamond B) = r \,\}$$

converges.

Exercise 10.19. Consider the Markov reward model for the simulation of a die by coin-tossing actions as in Example 10.75 on page 825. Show that for each $r \in \mathbb{N}$, $r \geqslant 0$:

$$Pr(s_0 \models \Diamond\{\, 1,2,3,4,5,6 \,\}) \;=\; 1 - \frac{1}{4^r}$$

Exercise 10.20. In Section 10.5, we only considered rewards for the states. Let us now study reward models that rely on a finite Markov chain where the edges are augmented by rewards. I.e., we deal with reward functions $rew : S \times S \to \mathbb{N}$ such that $rew(s, s') = 0$ if $\mathbf{P}(s, s') = 0$. The *cumulative reward* of a finite path fragment $s_0\, s_1 \ldots s_n$ is now defined as the sum of the reward earned by traversing the edges $(s_0, s_1), (s_1, s_2), \ldots, (s_{n-1}, s_n)$. That is,

$$rew(s_0\, s_1 \ldots s_n) \;=\; \sum_{1 \leqslant i \leqslant n} rew(s_{i-1}, s_i)$$

Expected rewards $ExpRew(s \models \Diamond T)$ and reward-bounded reachability events $\Diamond_{\leqslant r} T$ are defined as in Section 10.5. Provide algorithms to compute expected rewards $ExpRew(s \models \Diamond T)$, and

probabilities for reward-bounded reachability properties $Pr(s \models \Diamond_{\leqslant r} T)$ for finite Markov chains with reward structures that attach rewards to the edges.

EXERCISE 10.21. Provide a definition of bisimulation equivalence on Markov reward models such that bisimulation equivalence agrees with PRCTL-equivalence. Prove the correctness of your notion of bisimulation equivalence.

EXERCISE 10.22. Consider the MDP $\mathcal{M}$ shown in Figure 10.24. Let $B = \{ s_6 \}$. Compute the values $x_s = Pr^{\max}(s \models \Diamond B)$ where s is a state in the MDP $\mathcal{M}$. Take the following approach: first determine the states for which $Pr^{\max}(s \models \Diamond B) \in \{ 0, 1 \}$; then solve the corresponding linear program for the remaining states.

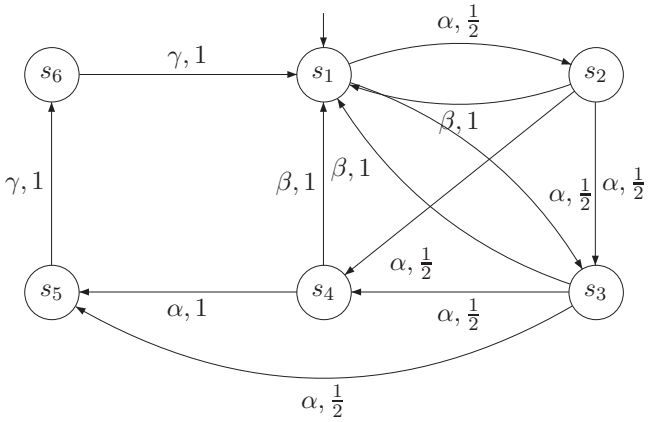


Figure 10.24: Markov decision process $\mathcal{M}$ for Exercise 10.22.

EXERCISE 10.23. Given a finite Markov decision process $\mathcal{M}$ with state space S and a subset B of S, show that there exists a memoryless scheduler $\mathfrak{S}$ such that

$$Pr^{\max}(s \models \Box\Diamond B) \;=\; Pr^{\mathfrak{S}}(s \models \Box\Diamond B).$$

For which other types of limit LT properties P exist memoryless schedulers that maximize the probabilities for P? Consider

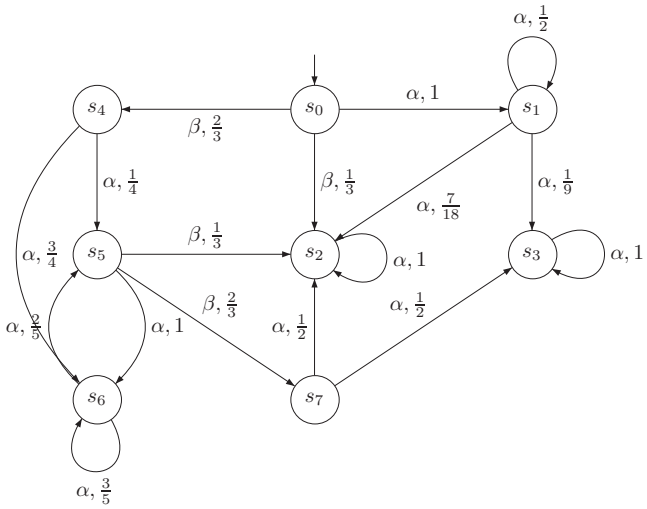
(a) persistence properties $\Diamond\Box B$,

1. Rabin conditions $\bigvee_{1 \leqslant i \leqslant k} (\Diamond\Box B_i \wedge \Box\Diamond C_i)$,

(b) strong fairness assumptions $\bigwedge_{1 \leqslant i \leqslant k} (\Box\Diamond B_i \rightarrow \Box\Diamond C_i)$.

Exercise 10.24. Show or disprove that the class of limit LT properties is closed under intersection. Do the same for union and complementation.

Exercise 10.25. Determine the maximal end components of the following MDP:



Exercise 10.26. Let $\mathcal{M}$ be a finite MDP over AP and $\mathcal{A}$ a DRA over the alphabet 2^{AP}. On page 882 we described a transformation "scheduler $\mathfrak{S}$ for $\mathcal{M} \rightsquigarrow$ scheduler $\mathfrak{S}'$ for $\mathcal{M} \otimes \mathcal{A}$", and vice versa. Questions:

(a) Is it true that if $\mathfrak{S}$ is memoryless, then so is $\mathfrak{S}'$?

(b) Is it true that if $\mathfrak{S}'$ is memoryless, then so is $\mathfrak{S}$?

(c) Assume that $\mathfrak{S}'$ is a finite-memory scheduler for $\mathcal{M} \otimes \mathcal{A}$. Describe the corresponding finite-memoryless scheduler $\mathfrak{S}$ by its modes.

Exercise 10.27. Let $\mathcal{M} = (S, Act, \mathbf{P}, \iota_{\text{init}}, AP, L)$ be a Markov decision process. A bisimulation on $\mathcal{M}$ is an equivalence $\mathcal{R}$ on S such that for all $(s_1, s_2) \in \mathcal{R}$:

(a) $L(s_1) = L(s_2)$

(b) for all $\alpha \in Act(s_1)$ there exists $\beta \in Act(s_2)$ such that for all $\mathcal{R}$-equivalence classes $T \in S/\mathcal{R}$:

$$\mathbf{P}(s_1, \alpha, T) \;=\; \mathbf{P}(s_2, \beta, T).$$

States s_1, s_2 of $\mathcal{M}$ are called bisimulation equivalent iff there exists a bisimulation $\mathcal{R}$ with $(s_1, s_2) \in \mathcal{R}$.

(a) Show that bisimulation-equivalent states satisfy the same PCTL* formulae over AP.

(b) Suppose that $\mathcal{M}$ is finite. Show that PCTL-equivalent states are bisimulation equivalent.

In (b), PCTL-equivalence of two states s_1, s_2 of $\mathcal{M}$ means that s_1 and s_2 fulfill the same PCTL formulae over AP.

EXERCISE 10.28. Let $\mathcal{M}$ be a finite Markov decision process and *fair* an LTL fairness assumption. Prove the equivalence of the following statements:

(a) There exists a fair scheduler for $\mathcal{M}$.

(b) There exists a fair finite-memory scheduler for $\mathcal{M}$.

(c) There exists a scheduler $\mathfrak{S}$ for $\mathcal{M}$ such that, for all states s of $\mathcal{M}$, $Pr^{\max}(s \models \Diamond U_{fair}) = 1$, where U_{fair} denotes the success set of *fair*, i.e., the union of the sets T of all end components (T, A) of $\mathcal{M}$ such that $T \models fair$.

Here, fairness of a scheduler $\mathfrak{F}$ for $\mathcal{M}$ means that for all states s of $\mathcal{M}$ it holds that

$$Pr_s^{\mathfrak{F}}\big\{\pi \in Paths(s) \mid \pi \models fair\big\} \;=\; 1.$$

EXERCISE 10.29. Let $\mathcal{M}$ be a finite Markov decision process and

$$sfair \;=\; \bigwedge_{1 \leqslant i \leqslant k} (\Box\Diamond a_i \rightarrow \Box\Diamond b_i)$$

a strong fairness assumption. Questions:

(a) Design an algorithm that runs in time $\mathcal{O}\big(\,\mathrm{poly}(size(\mathcal{M})) \cdot k\,\big)$ and computes all end components (T, A) of $\mathcal{M}$ such that $T \models sfair$ and such that (T, A) is not contained in another end component (T', A') with $T' \models sfair$.

(b) Design an algorithm that takes as input a finite MDP $\mathcal{M}$ and an LTL fairness assumption *sfair* and checks in time $\mathcal{O}(\,\mathrm{poly}(size(\mathcal{M})) \cdot |\varphi|\,)$ whether φ is realizable for $\mathcal{M}$.

(*Hint for part (a): combine the ideas of the algorithm to compute the maximal end components (Algorithm 47) with the ideas of model checking CTL with fairness.*)

Appendix A

Appendix: Preliminaries

A.1 Frequently Used Symbols and Notations

The symbol ■ only serves readability. It indicates the end of a definition, of a remark, or of other numbered text fragments. The abbreviation "iff" stands for "if and only if". We also often use logical symbols, such as $\wedge$ for "and", $\vee$ for "or" and the following quantifiers:

$\exists$	"there exists ..."
$\forall$	"for all"
$\overset{\infty}{\exists}$	"there exist infinitely many"
$\overset{\infty}{\forall}$	"for almost all, i.e., for all except for finitely many".

The Greek Alphabet At various places, Latin and Greek letters serve as symbols for certain mathematical objects. Although not all Greek letters will be used in this monograph, Figure A.1 shows the complete Greek alphabet. The symbols Γ, Δ, Θ, Λ, Ξ, Σ, Υ, Φ, Ψ, and Ω are capital letters. All other symbols in Figure A.1 are lowercase letters.

Natural and Real Numbers The symbol $\mathbb{N}$ denotes the set $\{0, 1, 2, \dots\}$ of natural numbers. The set of real numbers is denoted $\mathbb{R}$. Subsets of $\mathbb{N}$ and $\mathbb{R}$ are often denoted by subscripts. For instance, $\mathbb{R}_{\geqslant 0}$ denotes the set of non-negative reals, while $\mathbb{R}_{>5}$ denotes the interval $]5, \infty[$.

α	alpha	ι	iota	ρ, ϱ	rho
β	beta	κ	kappa	σ, ς, Σ	sigma
γ, Γ	gamma	λ, Λ	lambda	τ	tau
δ, Δ	delta	μ	mu	υ, Υ	upsilon
ϵ, ε	epsilon	ν	nu	ϕ, φ, Φ	phi
ζ	zeta	ξ, Ξ	xi	χ	chi
η	eta	o	omicron	ψ, Ψ	psi
θ, ϑ, Θ	theta	π, ϖ, Π	pi	ω, Ω	omega

Figure A.1: The Greek alphabet.

Asymptotic operators $\mathcal{O}$, Ω, and Θ To specify the asymptotic growth of (cost) functions, the Landau symbols $\mathcal{O}$, Ω, and Θ are used (see, e.g., [100]):

$\mathcal{O}$: asymptotic upper bound
Ω : asymptotic lower bound
Θ : asymptotic upper and lower bound

The precise meaning is as follows. If $f : \mathbb{N} \to \mathbb{N}$ is a function, then $\mathcal{O}(f)$ denotes the set of all functions $g : \mathbb{N} \to \mathbb{N}$ such that there exists a constant $C > 0$ and natural number N with $g(n) \leqslant C \cdot f(n)$ for all $n \in \mathbb{N}$ where $n \geqslant N$. The function class $\Omega(f)$ consists of all functions $g : \mathbb{N} \to \mathbb{N}$ such that there exists a constant $C > 0$ and a natural number N with $g(n) \geqslant C \cdot f(n)$ for all $n \geqslant N$. The class $\Theta(f)$ is given by $\Theta(f) = \mathcal{O}(f) \cap \Omega(f)$. It is common to use the equality symbol instead of the element symbol "$\in$" and to write, e.g., $g = \mathcal{O}(f)$ or $g(n) = \mathcal{O}(f(n))$ rather than $g \in \mathcal{O}(f)$. Thus, "equations" with the asymptotic operators have to be read from the left to the right.

We write $\mathcal{O}(\mathrm{poly}(n))$ to denote the class of all functions $g : \mathbb{N} \to \mathbb{N}$ that are polynomially bounded, i.e., $g(n) = \mathcal{O}(n^k)$ for some natural number k. Similarly, $\mathcal{O}(\exp(n))$ denotes the class of all functions $g : \mathbb{N} \to \mathbb{N}$ that are exponentially bounded, i.e., $g(n) = \mathcal{O}(a^{n^k})$ where $a > 1$ is a real constant and k a natural number.

Notation for Sets Let X be a set. The symbol 2^X stands for the *powerset* of X, i.e., the set consisting of all subsets of X. We write $|X|$ to denote the cardinality of X, i.e., the number of elements of X. (If X is infinite, then $|X| = \omega$.) Besides the standard symbols $\cup$ for union, $\cap$ for intersection, and $\setminus$ for "set-minus", i.e., $X \setminus Y = \{x \in X \mid x \notin Y\}$, the symbol $\uplus$ will be used which denotes *disjoint union*. Formally, for sets X, Y, $X \uplus Y$ is defined by $\{(x, 1) \mid x \in X\} \cup \{(y, 2) \mid y \in Y\}$. Additionally, we mention special cases for union and intersection. For $x \in X$, let Y_x be subset of a set Y. Then,

$$\bigcup_{x \in \varnothing} Y_x \;=\; \varnothing, \qquad \bigcap_{x \in \varnothing} Y_x \;=\; Y.$$

Thus, $\bigcap_{x \in \varnothing} \ldots$ depends on the chosen universal set Y.

Relations A relation denotes any set of tuples of fixed length. More precisely, let $X_1, \ldots, X_k$ be sets where $k \in \mathbb{N}$ with $k \geqslant 1$. Subsets of the Cartesian products $X_1 \times \ldots \times X_k$ are also called *relations* or *predicates*. The number k denotes the *arity*. If $X_1 = X_2 = \ldots = X_k = X$, then the subsets of

$$X^k \;=\; X_1 \times \ldots \times X_k$$

are called k-ary relations on X. Very often we have to deal with binary (2-ary) relations $\mathcal{R}$ over a set X. For these relations the infix notation $x\mathcal{R}y$ is often used instead of $(x, y) \in \mathcal{R}$. If X is a set and $\mathcal{R}$ a binary relation on X, then $\mathcal{R}$ is called

- *transitive* if for all x, y, $z \in X$, $x\mathcal{R}y$, and $y\mathcal{R}z$ implies $x\mathcal{R}z$;
- *reflexive* if for all $x \in X$, $x\mathcal{R}x$;
- *symmetric* if for all x, $y \in X$, $x\mathcal{R}y$ implies $y\mathcal{R}x$;
- *antisymmetric* if for all x, $y \in X$, $x\mathcal{R}y$, and $y\mathcal{R}x$ implies $x = y$.

Equivalences An equivalence relation (or briefly equivalence) on X is a transitive, reflexive, and symmetric binary relation on X. Symmetric symbols (like $\sim$, $\equiv$, or similar symbols) are often used to denote an equivalence relation. If $x \in X$, then $[x]_\mathcal{R} \;=\; \{\, y \in X \mid x\mathcal{R}y \,\}$ is called the *equivalence class* of x with respect to $\mathcal{R}$. If $\mathcal{R}$ follows from the context, $[x]$ is often written for short instead of $[x]_\mathcal{R}$. The *quotient space* of X with respect to $\mathcal{R}$ is the set of all equivalence classes with respect to $\mathcal{R}$ and is denoted $X/\mathcal{R}$. Then, $X/\mathcal{R} \;=\; \{[x]_\mathcal{R} \mid x \in X\}$. If $\mathcal{R}$ is an equivalence relation on X, then for all x, $y \in X$:

$$x\mathcal{R}y \;\text{ iff }\; [x]_\mathcal{R} = [y]_\mathcal{R} \;\text{ iff }\; [x]_\mathcal{R} \cap [y]_\mathcal{R} \neq \varnothing.$$

So the quotient space consists of pairwise disjoint nonempty subsets of X whose union just results in X. For each element $x \in X$, there is always exactly one element A of the quotient space with $x \in A$ (i.e. $A = [x]_\mathcal{R}$). The *index* of an equivalence relation $\mathcal{R}$ denotes the number of equivalence classes, i.e., the cardinality of $X/\mathcal{R}$. It is usual to use spellings like "$\mathcal{R}$ *is of finite index*" if the index of $\mathcal{R}$ is finite (a natural number).

Let $\mathcal{R}$ and $\mathcal{R}'$ be two equivalence relations on X. Relation $\mathcal{R}$ is a *refinement* of $\mathcal{R}'$ if $\mathcal{R} \subseteq \mathcal{R}'$, i.e., $\mathcal{R}$ "distinguishes" more elements than $\mathcal{R}'$. In this case, it is also said that $\mathcal{R}$ is *finer* than $\mathcal{R}'$ and that $\mathcal{R}'$ is *coarser* than $\mathcal{R}$. It is referred to as a *proper refinement* if $\mathcal{R}$ is a refinement of $\mathcal{R}'$ with $\mathcal{R} \neq \mathcal{R}'$. If $\mathcal{R}$ is a refinement of $\mathcal{R}'$, then every equivalence class with respect to $\mathcal{R}$ is contained in *exactly* one equivalence class with respect to $\mathcal{R}'$, because we have $[x]_\mathcal{R} \subseteq [x]_{\mathcal{R}'}$. This observation can be enforced as follows. Each equivalence class with respect to $\mathcal{R}'$ can be written as a disjoint union of equivalence classes with respect

to $\mathcal{R}$. Thus, if $\mathcal{R}$ is a refinement of $\mathcal{R}'$, then $|X/\mathcal{R}'| \leqslant |X/\mathcal{R}|$. Therefore, the index of $\mathcal{R}'$ is at most the index of $\mathcal{R}$.

Transitive and Reflexive Closure Let $\mathcal{R}$ be a binary relation over X. The *transitive and reflexive closure* of $\mathcal{R}$ is the smallest transitive, reflexive binary relation on X containing $\mathcal{R}$. Usually, this is denoted by $\mathcal{R}^*$. Thus,

$$\mathcal{R}^* \;=\; \bigcup_{n \geqslant 0} \mathcal{R}^n$$

where $\mathcal{R}^0 = \{(x,x) \mid x \in X\}$ and $\mathcal{R}^{n+1} \;=\; \{\, (x,y) \in X \times X \mid \exists z \in X.(x,z) \in \mathcal{R} \wedge (z,y) \in \mathcal{R}^n \,\}$. If $\mathcal{R}$ is symmetric, then $\mathcal{R}^*$ is an equivalence relation.

Partitions A *partition* (or partitioning) of a set X is a set $\mathcal{B} \subseteq 2^X$ consisting of pairwise disjoint, nonempty subsets of X such that $\bigcup_{B \in \mathcal{B}} B = X$. Elements of $\mathcal{B}$ are called *blocks*, as well. In particular, each element $x \in X$ is included in exactly one block $B \in \mathcal{B}$. There is a close connection between equivalence relations on X and partitions of X. If $\mathcal{R}$ is an equivalence relation on X, then the quotient space $X/\mathcal{R}$ is a partition of X. Vice versa, if $\mathcal{B}$ is a partition of X, then

$$\{(x,y) \mid x \text{ and } y \text{ are in the same block } B \in \mathcal{B}\}$$

is an equivalence relation with quotient space $\mathcal{B}$.

Preorder A preorder $\mathcal{R}$ on X denotes a reflexive, transitive relation on X. Any preorder induces an equivalence, the so-called *kernel* of $\mathcal{R}$, which is given by $\mathcal{R} \cap \mathcal{R}^{-1}$, i.e., the relation $\{(x,y) \mid x\mathcal{R}y \text{ and } y\mathcal{R}x\}$.

A.2 Formal Languages

This section summarizes the main concepts of regular languages. Further details and the proofs of the results mentioned here can be found in any text book on formal language theory, see e.g., [214, 272, 363, 383].

Words over an Alphabet An alphabet is an arbitrary nonempty and finite set Σ. The elements of Σ are typically called symbols or letters. A word over Σ denotes a finite or infinite sequence of symbols in Σ, i.e., words have the form $w = A_1\, A_2 \ldots A_n$ where $n \in \mathbb{N}$ or $\sigma = A_1\, A_2 A_3 \ldots$ and where the A_i's are symbols in Σ. [1] The special case $n = 0$ is

[1] Formally, infinite words can be defined as functions $\sigma : \mathbb{N} \to \Sigma$ and the notation $\sigma = A_1\, A_2 A_3 \ldots$ means that $\sigma(i) = A_i$ for all $i \in \mathbb{N}$. Similarly, finite words are obtained by functions $w : \{1, \ldots, n\} \to \Sigma$.

allowed, in which case the so-called empty word, denoted ε, is obtained. The *length* of a word is the number of symbols in the given word. Thus, for $w = A_1\, A_2 \ldots A_n$, the length is n, while each infinite word has the length ω. (The Greek letter ω (omega) is typically used to denote "infinity".) Σ^* denotes the set consisting of all finite words over Σ, while Σ^ω denotes the set of all infinite words over Σ. Note that $\varepsilon \in \Sigma^*$. Thus, the set of all nonempty finite words, denoted Σ^+, is $\Sigma^* \setminus \{\varepsilon\}$. A set of finite words over the alphabet Σ is called a *language*, and is ranged over by $\mathcal{L}$ (and primed and subscripted versions thereof).

A *prefix* of a finite word $w = A_1\, A_2 \ldots A_n$, is a word v of the form $A_1 A_2 \ldots A_i$ for some i where $0 \leqslant i \leqslant n$. A *suffix* of w is a word v of the form $A_i\, A_{i+1} \ldots A_n$ where $1 \leqslant i \leqslant n{+}1$. (In particular, ε is a prefix and suffix of any finite word.) The words of the form $A_i A_{i+1} \ldots A_j$ with $1 \leqslant i \leqslant j \leqslant n$ are called subwords of w. The definition of prefixes, suffixes, and subwords of infinite words are similar. For infinite word $\sigma = A_0\, A_1\, A_2 \ldots$, the suffix $A_j\, A_{j+1}\, A_{j+2} \ldots$ is denoted $\sigma[j..]$. For technical reasons, we start with the index 0. Thus, $\sigma = \sigma[0..]$.

Operations on Words Important operations on words are concatenation and finite repetition. *Concatenation* takes two words and "glues them together" to construct a new word. It is denoted by juxtaposition. For instance, the concatenation of the words BA and AAB yields the word $BA.AAB = BAAAB$. The concatenation of a word with itself is denoted by squaring, e.g., $(AB)^2$ equals $ABAB$; this is generalized in a straightforward manner for arbitrary n. In the special cases $n = 0$ and $n = 1$, we have $w^0 = \varepsilon$ (the empty word) and $w^1 = w$.

Finite repetition, also called *Kleene star* and denoted by *, of a finite word w yields the language consisting of all finite words that arise by zero or more (but finitely many) repetitions of w. Formally, for $w \in \Sigma^*$ we have $w^* = \{w^i \mid i \in \mathbb{N}\}$. For instance, $(AB)^* = \{\, \varepsilon, AB, ABAB, ABABAB, \ldots \}$. Note that the empty word ε is included in w^* for each finite word w. This is precisely the difference with the slight variant of the Kleene star, denoted $^+$, defined by $w^+ = \{w^i \mid i \in \mathbb{N}, i \geqslant 1\}$, or, equivalently, $w^+ = w^* \setminus \{\, \varepsilon \,\}$. For instance, $(AB)^+$ denotes the set $\{\, AB, ABAB, ABABAB, \ldots \}$.

Operations on Languages Concatenation is lifted to languages in a natural way as a pointwise extension of concatenation on words. The same applies to repetition. For languages $\mathcal{L}$, $\mathcal{L}_1, \mathcal{L}_2 \subseteq \Sigma^*$ we have

$$\mathcal{L}_1.\mathcal{L}_2 = \{\, w_1.w_2 \mid w_1 \in \mathcal{L}_1, w_2 \in \mathcal{L}_2 \,\}$$

and

$$\mathcal{L}^* = \bigcup_{i=0}^{\infty} \mathcal{L}^i, \quad \text{and} \quad \mathcal{L}^+ = \bigcup_{i=1}^{\infty} \mathcal{L}^i,$$

where $\mathcal{L}^i$ denotes the concatenation of i times $\mathcal{L}$. Note that the star and plus notation for

languages over finite words is consistent with the standard notations Σ^* and Σ^+ for the set of all finite words over Σ with or without the empty word, respectively. For instance, for $\mathcal{L}_1 = \{ A, AB \}$ and $\mathcal{L}_2 = \{ \varepsilon, BBB \}$ we have that

$$\begin{array}{lcl} \mathcal{L}_1.\mathcal{L}_2 & = & \{ A, AB, ABBB, ABBBB \}, \text{and} \\ \mathcal{L}_1^2 & = & \{ AA, AAB, ABAB, ABA \}. \end{array}$$

There are several equivalent formalisms to describe regular languages. In this monograph we only use regular expressions and finite automata. We start with the former and introduce the automata approach later.

Regular Expressions Regular expressions (denoted E or F) are built from the symbols $\underline{\varnothing}$ (to denote the empty language), $\underline{\varepsilon}$ (to denote the language $\{ \varepsilon \}$ consisting of the empty word), the symbols $\underline{A}$ for $A \in \Sigma$ (for the singleton sets $\{ A \}$), and the language operators "$+$" (union), "$*$" (Kleene star, finite repetition), and "$.$" (concatenation). Formally, regular expressions can be defined inductively:

1. $\underline{\varnothing}$ and $\underline{\varepsilon}$ are regular expressions over Σ.
2. If $A \in \Sigma$, then $\underline{A}$ is a regular expression over Σ.
3. If E, E_1 and E_2 are regular expressions over Σ, then so are $\mathsf{E}_1 + \mathsf{E}_2$, $\mathsf{E}_1.\mathsf{E}_2$, and E^*.
4. Nothing else is a regular expression over Σ.

E^+ is an abbreviation for the regular expression $\mathsf{E}.\mathsf{E}^*$. The semantics of a regular expression E is a language $\mathcal{L}(\mathsf{E}) \subseteq \Sigma^*$ that is defined as follows:

$$\mathcal{L}(\underline{\varnothing}) = \varnothing, \quad \mathcal{L}(\underline{\varepsilon}) = \{ \varepsilon \}, \quad \mathcal{L}(\underline{A}) = \{ A \}$$

and

$$\mathcal{L}(\mathsf{E}_1 + \mathsf{E}_2) = \mathcal{L}(\mathsf{E}_1) \cup \mathcal{L}(\mathsf{E}_2), \quad \mathcal{L}(\mathsf{E}_1.\mathsf{E}_2) = \mathcal{L}(\mathsf{E}_1).\mathcal{L}(\mathsf{E}_2), \quad \mathcal{L}(\mathsf{E}^*) = \mathcal{L}(\mathsf{E})^*.$$

From this definition, we derive $\mathcal{L}(\mathsf{E}^+) = \mathcal{L}(\mathsf{E})^+$.

A language $\mathcal{L} \subseteq \Sigma^*$ is called *regular* if there is some regular expression E over Σ such that $\mathcal{L}(\mathsf{E}) = \mathcal{L}$. For instance, $\mathsf{E} = (\underline{A}+\underline{B})^*.\underline{B}.\underline{B}.(\underline{A}+\underline{B})$ is a regular expression over $\Sigma = \{ A, B \}$ representing the language

$$\mathcal{L}(\mathsf{E}) = \{ wBBA \mid w \in \Sigma^* \} \cup \{ wB^3 \mid w \in \Sigma^* \},$$

consisting of all finite words that end with three B's or with the word BBA. The regular expression $\mathsf{E}' = (\underline{A} + \underline{B})^*.\underline{B}.\underline{B}.(\underline{A} + \underline{B})^*$ represents the regular language consisting of all finite words over $\Sigma = \{ A, B \}$ that contain the subword BB.

It is standard to use a simplified syntax for regular expressions that does not distinguish between the atomic expression $\underline{x}$ and the symbol x for $x \in \{\varnothing, \varepsilon\} \cup \Sigma$ and skips the operator symbol ". " for concatenation. For example,

$$(A + B)^* BB(A + B)$$

stands for the regular expression $(\underline{A} + \underline{B})^*.\underline{B}.\underline{B}.(\underline{A} + \underline{B})$.

A.3 Propositional Logic

This section summarizes the basic principles of propositional logic. For more elaborate treatments we refer to the textbook [342].

Given is a finite set AP of *atomic propositions*, sometimes also called *propositional symbols*. In the following, Latin letters like a, b, and c (with or without subscripts) are used to denote elements of AP. The set of *propositional logic formulae* over AP, formulae for short, is inductively defined by the following four rules:

1. true is a formula.
2. Any atomic proposition $a \in AP$ is a formula.
3. If Φ_1, Φ_2 and Φ are formulae, then so are $(\neg\Phi)$ and $(\Phi_1 \wedge \Phi_2)$.
4. Nothing else is a formula.

Any formula stands for a proposition that might hold or not, depending on which of the atomic propositions are assumed to hold. Intuitively, the formula a stands for the propositions stating that a holds. The intuitive meaning of the symbol $\wedge$ is conjunction ("and"), i.e., $\Phi_1 \wedge \Phi_2$ holds if and only if both propositions Φ_1 and Φ_2 hold. The symbol $\neg$ denotes negation, i.e., $\neg\Phi$ holds if and only if Φ does not hold. Thus, e.g., $a \wedge \neg b$ holds if and only if a holds and b does not hold. The constant true stands for a proposition which holds in any context, independent of the interpretation of the atomic propositions a.

It is standard to use simplified notations for formulae with the help of derived operators and by declaring a precedence order on the basic and derived operators, which often allows skipping brackets. The standard precedence order assigns a higher priority to the unary negation operator $\neg$ than the binary conjunction operator $\wedge$. Thus, $\neg a \wedge b$ is short for

$((\neg a) \wedge b)$. Moreover, conjunction $\wedge$ binds stronger than the derived binary operators, such as

$\Phi_1 \vee \Phi_2$	$\stackrel{\text{def}}{=}$	$\neg(\neg\Phi_1 \wedge \neg\Phi_2)$	disjunction ("or")
$\Phi_1 \rightarrow \Phi_2$	$\stackrel{\text{def}}{=}$	$\neg\Phi_1 \vee \Phi_2$	implication
$\Phi_1 \leftrightarrow \Phi_2$	$\stackrel{\text{def}}{=}$	$(\neg\Phi_1 \wedge \neg\Phi_2) \vee (\Phi_1 \wedge \Phi_2)$	equivalence
$\Phi_1 \oplus \Phi_2$	$\stackrel{\text{def}}{=}$	$(\neg\Phi_1 \wedge \Phi_2) \vee (\Phi_1 \wedge \neg\Phi_2)$	parity (xor)

For example, $\neg a \vee \neg b \wedge c$ is a short-form notation for $(\neg a) \vee ((\neg b) \wedge c)$ which – by the definition of $\vee$ – stands for the formula $\Phi = (\neg(\neg a) \wedge \neg((\neg b)) \wedge c))$.

Abstract Syntax Throughout this monograph, we provide the definition of the syntax of logics in a more relaxed way. Skipping the syntactic rules for brackets (which can be derived from the precedence order of the operators that will be declared in words), the above inductive definition of propositional formulae over AP can be rewritten as

$$\Phi \quad ::= \quad \text{true} \;\Big|\; a \;\Big|\; \Phi_1 \wedge \Phi_2 \;\Big|\; \neg\Phi$$

where $a \in AP$. The above can be understood as a casual notation for the Backus-Naur form of a context-free grammar over the alphabet $\Sigma = \{\text{true}\} \cup AP \cup \{\neg, \wedge\}$. In this short-form notation, the symbol Φ serves simultaneously for (1) a nonterminal symbol (variable) of the grammar and (2) its derived words over Σ^* (i.e., propositional formulae). The latter explains the indices in the term $\Phi_1 \wedge \Phi_2$, which is correct on the formula level, although the correct notation would be $\Phi \wedge \Phi$ (without indices) in the grammar.

Length of a Formula The length of a formula Φ is defined by the number of operators in Φ and is denoted by $|\Phi|$. For instance, the formula $\Phi = (\neg b) \wedge c$ has the length 2. Since in most cases we are only interested in the asymptotic length of formulae in formula sequences (Φ_n), we may also assign one cost unit to the derived operators $\vee$ and $\rightarrow$. In fact, the asymptotic formula length does not depend on whether $\vee$ and $\rightarrow$ are treated as a basic operator (with one cost unit per occurrence in a formula) or a derived one (using conjunction and negation).

Semantics of Propositional Logic To formalize the intuitive meaning of propositional formulae, we first need a precise definition of the "context" that declares which atomic propositions hold and which do not hold. This is done by means of an *evaluation* which assigns a truth value 0 ("false") or 1 ("true") to each atomic proposition. Formally, an *evaluation* for AP is a function $\mu : AP \rightarrow \{0, 1\}$. *Eval*($AP$) denotes the set of all evaluations for AP. The semantics of propositional logic is specified by a *satisfaction relation* $\models$ indicating the evaluations μ for which a formula Φ is true. Formally, $\models$ is a set of pairs (μ, Φ) where μ is an evaluation and Φ is a formula. It is written

$$\begin{array}{lll} \mu \models \text{true} & & \\ \mu \models a & \text{iff} & \mu(a) = 1 \\ \mu \models \neg\Phi & \text{iff} & \mu \not\models \Phi \\ \mu \models \Phi \wedge \Psi & \text{iff} & \mu \models \Phi \text{ and } \mu \models \Psi. \end{array}$$

Figure A.2: The satisfaction relation $\models$ of propositional logic.

$$\mu \models \Phi \quad \text{instead of} \quad (\mu, \Phi) \in \models.$$

Accordingly, $\mu \not\models \Phi$ stands for $(\mu, \Phi) \notin \models$. Intuitively, $\mu \models \Phi$ stands for the fact that Φ is true under evaluation μ. The satisfaction relation $\models$ is inductively defined by the conditions indicated in Figure A.2. If $\mu \models \Phi$, then μ is called a *satisfied condition* for Φ. In literature, the notation $\mu(\Phi) = 1$ if $\mu \models \Phi$, and $\mu(\Phi) = 0$, if $\mu \not\models \Phi$, is used, too. The value $\mu(\Phi) \in \{0, 1\}$ is called the *truth-value* of Φ under μ.

Formulae with derived operators like disjunction $\vee$ or implication $\rightarrow$ have the expected semantics. Thus,

$$\begin{array}{lll} \mu \models \Phi \vee \Psi & \text{iff} & \mu \models \Phi \text{ or } \mu \models \Psi \\ \mu \models \Phi \rightarrow \Psi & \text{iff} & \mu \not\models \Phi \text{ or } \mu \models \Psi \\ & \text{iff} & \mu \models \Phi \text{ implies } \mu \models \Psi. \end{array}$$

Set Notation for Evaluations An alternative representation of evaluations for AP is based upon representation of sets. Each evaluation μ can be represented by the set $A_\mu = \{a \in AP \mid \mu(a) = 1\}$. And conversely, an evaluation $\mu = \mu_A$ with $A = A_\mu$ can be assigned to each subset A of AP. Evaluation μ_A is the characteristic function of A_μ, that is, $\mu_A(a) = 1$ if $a \in A$ and $\mu_A(a) = 0$ if $a \notin A$. This observation suggests extending the satisfaction relation $\models$ to subsets of AP by

$$A \models \Phi \quad \text{iff} \quad \mu_A \models \Phi.$$

As an example, we look at $\Phi = (a \wedge \neg b) \vee c$. Given an evaluation μ with $\mu(a) = 0$ and $\mu(b) = \mu(c) = 1$, then $\mu \not\models a \wedge \neg b$ and $\mu \models c$, and thus, $\mu \models \Phi$. The accompanying set A_μ is $A_\mu = \{b, c\}$. Hence, $\{b, c\} \models \Phi$. The empty set induces an evaluation $\mu_\varnothing$ with $\mu_\varnothing(a) = \mu_\varnothing(b) = \mu_\varnothing(c) = 0$. Due to $\mu_\varnothing \not\models \Phi$ (where $\Phi = (a \wedge \neg b) \vee c$ as above), we get $\varnothing \not\models \Phi$. However, we have $\varnothing \models \neg a \wedge \neg b$ since $\neg a$ and $\neg b$ hold for the associated evaluation $\mu_\varnothing$.

Semantic Equivalence Two propositional logic formulae Φ, Ψ are called *(semantically) equivalent* if they have the same truth-value for each evaluation. That is, for all evaluations μ:

rule for double negation	*idempotency law*
$\neg\neg\Phi \equiv \Phi$	$\Phi \vee \Phi \equiv \Phi$ $\Phi \wedge \Phi \equiv \Phi$
absorption law	*commutativity law*
$\Phi \wedge (\Psi \vee \Phi) \equiv \Phi$ $\Phi \vee (\Psi \wedge \Phi) \equiv \Phi$	$\Phi \wedge \Psi \equiv \Psi \wedge \Phi$ $\Phi \vee \Psi \equiv \Psi \vee \Phi$
associativity law	*de Morgan's law*
$\Phi \wedge (\Psi \wedge \Xi) \equiv (\Phi \wedge \Psi) \wedge \Xi$ $\Phi \vee (\Psi \vee \Xi) \equiv (\Phi \vee \Psi) \vee \Xi$	$\neg(\Phi \wedge \Psi) \equiv \neg\Phi \vee \neg\Psi$ $\neg(\Phi \vee \Psi) \equiv \neg\Phi \wedge \neg\Psi$
distributivity law	
$\Phi \vee (\Psi_1 \wedge \Psi_2) \equiv (\Phi \vee \Psi_1) \wedge (\Phi \vee \Psi_2)$ $\Phi \wedge (\Psi_1 \vee \Psi_2) \equiv (\Phi \wedge \Psi_1) \vee (\Phi \wedge \Psi_2)$	

Figure A.3: Some equivalence rules for propositional logic.

$$\mu \models \Phi \text{ iff } \mu \models \Psi.$$

In this case we write $\Phi \equiv \Psi$. For example, the formulae $a \wedge \neg\neg b$ and $a \wedge b$ are semantically equivalent. A few of the most important equivalence rules for propositional logic and the operators $\neg$, $\wedge$ and $\vee$ are shown in Figure A.3. Here, the Greek capital letters Φ, Ψ, Ξ (with or without subscripts) serve as metasymbols for formulae of propositional logic. The associativity and commutativity law for disjunction $\vee$ and conjunction $\wedge$ justify the omission of brackets and notations like

$$\bigwedge_{1 \leqslant i \leqslant n} \Phi_i \quad \text{or} \quad \Phi_1 \wedge \ldots \wedge \Phi_n \quad .$$

Note that the length of a formula of type $\bigwedge_{1 \leqslant i \leqslant n} \Phi_i$ is equal to $n - 1$ (and not to 1). Furthermore, notations like $\bigwedge_{i \in I} \Phi_i$ or $\bigwedge\{\Phi_i \mid i \in I\}$ are often used where I is an arbitrary finite index set. If I is nonempty, then Φ stands for one of the formulae $\Phi_{i_1} \wedge \ldots \wedge \Phi_{i_k}$ where $I = \{i_1, \ldots, i_k\}$ and $i_1, \ldots, i_k$ are pairwise different. For $I = \varnothing$, the convention is

$$\bigwedge_{i \in \varnothing} \Phi_i \stackrel{\text{def}}{=} \text{true}, \quad \text{while} \quad \bigvee_{i \in \varnothing} \Phi_i \stackrel{\text{def}}{=} \text{false}.$$

Satisfiability and Validity Propositional formula Φ is called *satisfiable* if there is an evaluation μ with $\mu \models \Phi$. Φ is called *valid* (or a *tautology*) if $\mu \models \Phi$ for each evaluation μ. Φ is *unsatisfiable* if Φ is not satisfiable. For example, $a \wedge \neg a$ is unsatisfiable, while

$a \vee \neg(a \wedge b)$ is a tautology. The formulae $a \vee \neg b$ and $a \wedge \neg b$ are satisfiable, but not tautologies. Obviously:

$$\begin{array}{ll} & \Phi \text{ is unsatisfiable} \\ \text{iff} & \mu \not\models \Phi \text{ for all evaluations } \mu \\ \text{iff} & \mu \models \neg\Phi \text{ for all evaluations } \mu \\ \text{iff} & \neg\Phi \text{ is valid.} \end{array}$$

Thus, Φ is unsatisfiable if and only if $\neg\Phi$ is a tautology.

Literals and Positive Normal Form (PNF) A *literal* means a formula of the form a or $\neg a$ where $a \in AP$ is an atomic proposition. Propositional formulae in *positive normal form* (PNF for short, also sometimes called *negation normal form*) use the negation operator only on the level of literals. To ensure that the class of PNF formulae is as expressive as full propositional logic, both the conjunction and the disjunction operator serve as basic operators. Thus, the abstract syntax of PNF formulae is

$$\Phi \quad ::= \quad \text{true} \;\Big|\; \text{false} \;\Big|\; a \;\Big|\; \neg a \;\Big|\; \Phi_1 \wedge \Phi_2 \;\Big|\; \Phi_1 \vee \Phi_2$$

where $a \in AP$. Given a (non-PNF) formula Φ, de Morgan's laws and the rule for double negation allow for "pushing the negation inside" until an equivalent formula in PNF arises. That is, successive application of the transformations

$$\begin{array}{lcl} \neg(\Phi_1 \wedge \Phi_2) & \rightsquigarrow & \neg\Phi_1 \vee \neg\Phi_2 \\ \neg(\Phi_1 \vee \Phi_2) & \rightsquigarrow & \neg\Phi_1 \wedge \neg\Phi_2 \\ \neg\neg\Psi & \rightsquigarrow & \Psi \end{array}$$

to Φ's subformulae yields an equivalent formula in PNF of the same asymptotic length.

Conjunctive and Disjunctive Normal Form Special cases of PNF are the *conjunctive normal form* (CNF) and *disjunctive normal form* (DNF). A CNF formula has the form

$$\bigwedge_{i \in I} \bigvee_{j \in J_i} \mathit{lit}_{i,j}$$

where I and J_i are arbitrary finite index sets and $\mathit{lit}_{i,j}$ for $i \in I$ and $j \in J_i$ are literals. The subformulae $\bigvee_{j \in J_i} \mathit{lit}_{i,j}$ are called *clauses*. For instance, $(a_1 \vee \neg a_3 \vee a_4) \wedge (\neg a_2 \vee \neg a_3 \vee a_4) \wedge a_4$ is a CNF formula with three clauses. Note that, e.g., true and false are also representable by CNF formulae: true is obtained by $I = \varnothing$, while false is equivalent to $a \wedge \neg a$, a CNF with two clauses consisting of one literal each. Given a PNF formula Φ, an equivalent CNF formula is obtained (on the basis of the distributivity laws) by applying the transformation rules:

$$\begin{array}{lcl} \Phi_0 \vee (\Psi_1 \wedge \Psi_2) & \rightsquigarrow & (\Phi_0 \vee \Psi_1) \wedge (\Phi_0 \vee \Psi_2) \\ (\Psi_1 \wedge \Psi_2) \vee \Phi_0 & \rightsquigarrow & (\Psi_1 \vee \Phi_0) \wedge (\Psi_2 \vee \Phi_0) \end{array}$$

to Φ's subformulae as long as possible. Thus, for each propositional formula Φ there exists an equivalent CNF formula Φ'. Similarly, DNF formulae are formulae of the form

$$\bigvee_{i \in I} \bigwedge_{j \in J_i} lit_{i,j}$$

where I and J_i are arbitrary finite index sets and $lit_{i,j}$ for $i \in I$ and $j \in J_i$ are literals. E.g., $(a_1 \wedge \neg a_2) \vee (\neg a_2 \wedge \neg a_3 \wedge a_4) \vee (\neg a_1 \wedge \neg a_3)$ is a DNF formula. A transformation similar to the one for CNF can be applied to prove that any propositional formula Φ has an equivalent DNF formula Φ'.

A.4 Graphs

In most chapters of this monograph, the reader is supposed to be familiar with the basic principles of graph theory and elementary graph algorithms. These are supposed to be known from basic courses. This section is a brief summary of the terms and concepts that are central to this monograph. The details can be found in any elementary textbook on algorithms and data structures; see, e.g. [24, 100], or textbooks that provide an introduction to graph theory, see, e.g. [396].

Digraphs (Graphs, for short) A digraph (or directed graph, in the following called *graph*) is a pair $G = (V, E)$ consisting of a set V of *vertices* and an edge relation $E \subseteq V \times V$. The elements of E are called *edges*. G is called finite if the set V (and hence also E) is finite. The basic models considered in this monograph will rely on graphs where the vertices and edges are augmented with certain information. E.g., instead of defining E as a set of vertex pairs, we may also deal with labeled edges, in which case E is a subset of $V \times \Sigma \times V$ for a certain alphabet Σ. If this additional information is irrelevant, the edge labels may be omitted and rather than considering $E \subseteq V \times \Sigma \times V$ we use $E' = \{ (v, w) \mid \text{there exists an edge } (v, \sigma, w) \in E \}$ to obtain a digraph in the above sense.

The following explanations refer to the case of a digraph $G = (V, E)$ where E is a binary relation over the vertices. For $v \in V$, let $Post_G(v)$, or briefly $Post(v)$, denote the set of direct successors of v; that is, $Post(v) = \{ w \in V \mid (v, w) \in E \}$. Similarly, $Pre_G(v)$, or briefly $Pre(v)$, stands for the set of direct predecessors of v, i.e., $Pre(v) = \{ w \in V \mid (w, v) \in E \}$. A vertex v is called *terminal* if $Post(v) = \varnothing$. Edges of the form (v, v) are called *self-loops*.

Paths in Graphs A *path* in G denotes a (finite or infinite) non-empty sequence of vertices $\hat{\pi} = v_0\, v_1 \ldots v_r$ with $r \geqslant 0$ or $\pi = v_0\, v_1\, v_2 \ldots$ such that $v_{i+1} \in Post(v_i)$, $i = 0, 1, 2, \ldots$. (In

the context of transition systems, the term *path* is used in the sense of a maximal path, that is, a path is either infinite or ending in a terminal state.) A path is called *simple* if its vertices are pairwise distinct, e.g., the finite path $v_0\, v_1 \ldots v_n$ is simple if $v_i \neq v_j$ for all $0 \leqslant i < j \leqslant n$. The *length* of a path defines the number of traversed edges along the path and is denoted by $|\cdot|$. If $\hat{\pi}$ and π are as above, then $|\hat{\pi}| = r$ and $|\pi| = \omega$. The set $\mathit{Post}^*(v)$ or $\mathit{Reach}(v)$ denotes the set of all vertices reachable from v; that is, the set of all vertices $w \in V$ with a finite path $v_0\, v_1 \ldots v_r$ with $v = v_0$ and $w = v_r$. A finite path $v_0, v_1, \ldots, v_r$ is called a *cycle* if $v_0 = v_r$ and $r > 0$. Graph G is called acyclic (or cycle-free) if G does not contain any cycle; otherwise G is called cyclic.

A popular representation of finite digraphs is by *adjacency lists*. This entails a list representation for the sets $\mathit{Post}(v)$ for each vertex v. To support direct access from vertex v to its direct successors, an array for the vertices (in arbitrary order) can be used which contains pointers to the heads of the adjacency lists.

Depth- and Breadth-First Search (DFS, BFS) Depth-first search (DFS) and breadth-first search (BFS) are important graph traversal strategies. They are both based on the skeleton given in Algorithm 48 which deterimines each vertex v that is reachable from the vertex v_0. R is the set of vertices that have been already visited, while U keeps track of all vertices that still have to be explored (provided they are not contained in R). The procedure terminates since each edge is tagged as *taken* at most once. Any vertex u can thus be added to U (and taken from U) at most $|\mathit{Pre}(u)|$ times. Hence, Algorithm 48 terminates after at most $\mathcal{O}(M)$ iterations where $M = |E|$ is the number of edges. (Note that $M = \sum_{v \in V} |\mathit{Post}(v)| = \sum_{u \in V} |\mathit{Pre}(u)|$.) On termination, R contains all vertices that are reachable from v_0.

In case all vertices $v \in V$ have to be visited, the algorithm can be started with some arbitrary vertex v_0 and restarted with a new vertex v_0', say, that has not yet been visited, i.e., $v_0' \notin R$. This can be repeated until all vertices belong to R. Given an adjacency list representation of the sets $\mathit{Post}(v)$, the time complexity is $\Theta(N + M)$ where $N = |V|$ is the number of vertices and $M = \sum_{v \in V} |\mathit{Post}(v)| = |E|$ the number of edges. Here, it is assumed that R and U are represented such that access to the elements of R and U, as well as the insertion and deletion into U, takes constant time. For moderately sized graphs, the set R can be represented by a bit-vector. In the context of model checking, R is typically represented by a hash table. Using this data structure, the expected time to insert an element into R and to check membership in R is constant.

Depth-first and breadth-first search differ in the realization of the multiset U. The DFS approach organizes U as a *stack*, while BFS relies on a *queue* implementation for U. Accordingly, a stack obeys the LIFO principle (last-in, first-out), while the insertion and deletion of elements from a queue relies on the FIFO principle (first-in, first-out).

Algorithm 48 Reachability analysis

Input: finite digraph $G = (V, E)$, vertex $v_0 \in V$
Output: $\mathit{Reach}(v_0)$

```
set of  vertex R := ∅;                                   (* the set of explored vertices *)
multiset of  vertex U := {v_0};                          (* vertices still to be explored *)

(* initially none of the edges is tagged as taken *)

while U ≠ ∅ do
  let v ∈ U;                                   (* choose some vertex still to be explored *)
  if ∃u ∈ Post(v) such that (v,u) is not tagged as taken then
    let u ∈ Post(v) be such a vertex;
    tag the edge (v,u) as taken;
    if u ∉ R then
      U := U ∪ {u};                                (* ensure that all successors *)
      R := R ∪ { u }                                 (* of v will be explored *)
    fi
  else
    U := U \ { v }
  fi
od
return R.                                                (* R = Reach(v_0) *)
```

Let us briefly summarize some details of the DFS approach. As mentioned above, DFS-based graph traversal organizes the multiset of vertices that are still to be explored by a stack. Stacks support the operations $top(U)$ (which returns the first element of U), $pop(U)$ (which deletes the first element of U), and $push(U, v)$ (which inserts v into U as the first element). The empty stack will be denoted with ε. Using an appropriate array or list implementation of U, each of these operations, as well as checking emptiness, can be executed in constant time.

DFSs provide the basis for many algorithms that analyze the topological structure of a graph. An important algorithm in this monograph is a DFS-based cycle check, i.e., the problem of deciding whether a given finite digraph contains a cycle. Such an algorithm exploits the concept of *backward edges*. The edge (v, u) is a backward edge whenever vertex u is on the DFS stack U on tagging (v, u) as *taken*; see Algorithm 49.

Algorithm 49 Depth-first search

Input: finite digraph $G = (V, E)$, vertex $v_0 \in V$
Output: $Reach(v_0)$

```
set of vertex R := ∅;                                   (* the set of explored vertices *)
stack of vertex U := ε;                                 (* initialize stack U as empty *)
for all vertices v_0 ∈ V do
  if v_0 ∉ R then
    push(U, v_0);                          (* U represents the singleton (multi-)set {v_0} *)

    while U ≠ ε do
      v := top(U);                                      (* choose some vertex still to be explored *)
      if ∃u ∈ Post(v) such that (v, u) is not tagged as taken then
        let u ∈ Post(v) be such a vertex;
        tag the edge (v, u) as taken;
        if u ∉ R then
          push(U, u);
          R := R ∪ { u }                                                    (* u is visited *)
        else
          if u is contained in U then
            tag (v, u) as backward_edge                                     (* cycle found *)
          fi
        fi
      else
        U := U \ { v }
      fi
    od
  fi
od
                                                        (* any vertex v in G has been visited *)
```

Any vertex w which is visited (i.e., inserted into R) when u is on the stack U is reachable from u. Thus, any backward edge "closes" a cycle. Vice versa, if $v_0\, v_1 \ldots v_n$ is a cycle in G and v_0 is visited during the DFS after visiting $v_1, \ldots, v_{n-1}$, then—on removing v_0 as top element from U tagging the edge (v_0, v_1) as *taken*—the vertices v_2 through v_{n-1} are on the stack U (since v_0 is reachable from each of these vertices). As v_1 has been visited before v_0, the edge (v_0, v_1) is a backward edge. Hence, for any cycle $v_0\, v_1 \ldots v_n$ in G, the DFS classifies at least one edge (v_{i-1}, v_i) as a backward edge. This yields that G has a cycle if and only if the DFS detects a backward edge. Moreover, if we consider a fixed starting vertex v_0 and apply the DFS approach to visit (exactly) the vertices that are reachable from v_0, then v_0 belongs to a cycle in G if and only if the DFS finds a backward edge of the form (w, v_0) for some vertex w.

Strongly Connected Components (SCCs) Let $G = (V, E)$ be a finite digraph and $C \subseteq V$. C is *strongly connected* if for every pair of vertices v, $w \in C$, the vertices v and w are mutually reachable, i.e., $v \in \mathit{Post}^*(w)$ and $w \in \mathit{Post}^*(v)$. A *strongly connected component* (SCC, for short) of G is a maximally strongly connected set of vertices. That is, C is an SCC if C is strongly connected and C is not contained in another strongly connected vertex set $D \subseteq V$ with $C \neq D$. SCC C is called *trivial* if $C = \{ v \}$ and $(v, v) \notin E$. SCC C is called *terminal* if there is no SCC $D \neq C$ such that $(v, w) \in E$ for some $v \in C$ and $w \in D$. Finite digraph G is cyclic if and only if G contains a nontrivial SCC. The time complexity of determining the SCCs in finite digraph G is $\Theta(N + M)$ where $N = |V|$ and $M = |E|$. This complexity is obtained using a variant of DFS.

Trees The digraph $T = (V, E)$ is a (directed) tree if there exists a vertex v_0 with $\mathit{Pre}(v_0) = \varnothing$ such that each vertex $v \in V$ is reachable from v_0 via a unique path. The vertex v_0 is the unique vertex with this property and is called the *root* of T. Any vertex v with $\mathit{Post}(v) = \varnothing$ is called a *leaf*. Any vertex $v \in V \setminus \{ v_0 \}$ has exactly one direct predecessor, i.e., $|\mathit{Pre}(v)| = 1$. This unique direct predecessor w of v is the *father* of v; v is called a *son* of w. Tree T is called *finitely branching* if the maximal outdegree of all its vertices is finite, i.e., $\sup_{v \in V} |\mathit{Post}(v)| \in \mathbb{N}$. A binary tree is a tree with $|\mathit{Post}(v)| \leqslant 2$ for each vertex v.

Hamiltonian Paths A Hamiltonian path in a finite digraph $G = (V, E)$ is a path $v_1\, v_2 \ldots v_n$ where each vertex $v \in V$ appears exactly once, i.e., $V = \{ v_1, \ldots, v_n \}$ and $i \neq j$ implies $v_i \neq v_j$. The (directed) Hamiltonian path problem questions whether a given finite digraph G has a Hamiltonian path.

Undirected Graphs An undirected graph is a digraph in which the orientation of the edges is ignored. Formally, an undirected graph is a pair (V, E) where V is a set of vertices and E a set consisting of subsets of V with exactly two elements. Then, $\{ v, w \} \in E$ means that there is an edge connecting v and w. Paths in undirected graphs are defined as in

the directed case. For cycles it is required that they are composed of simple paths, i.e., paths $v_0\, v_1 \ldots v_n$ where $n > 1$ and $v_0, \ldots, v_n$ are pairwise distinct. (This constraint avoids that a path of the form $v\, w\, v$ with $\{ v, w \} \in E$ is classified as cycle.)

A.5 Computational Complexity

In this monograph, some lower bounds on the time complexity of verification problems are presented. These lower bounds are based on showing the hardness for a certain complexity class. This section summarizes the main basic concepts at a rather intuitive level. For the precise definitions and technical details we refer to textbooks on complexity theory, such as [160, 320].

The complexity class we consider here refers to decision problems, i.e., problems that take a certain (finite) input and are required to return either "yes" or "no". Example decision problems are:

SAT (satisfiability problem for propositional logic)
Given a propositional formula Φ. Is Φ satisfiable?

(Directed) Hamiltonion path problem
Given a finite digraph G. Does G have a Hamiltonion path?

Cycle problem
Given a finite digraph G. Is G cyclic?

Pattern matching
Given finite words $w = A_1 \ldots A_n$ (a text), $v = B_1 \ldots B_m$ (a pattern) over some alphabet. Is v a subword of w?

Complexity classes are classes of decision problems, classified by the resources (i.e., time and space) that are required to solve them algorithmically.

Deterministic and Nondeterministic Algorithms The precise definition of the complexity classes uses *Turing machines* as a formalization of algorithms. The input is encoded by a finite word of the input alphabet of the Turing machine. The answers "yes" and "no" correspond to the requirement that the Turing machine halts in an accept state ("yes") or in a nonaccept state ("no"). These details are omitted here. We describe informally the complexity classes by means of an intuitive notion of what an algorithm is. For the purpose of this monograph we need both *deterministic* and *nondeterministic* algorithms. Although one typically aims at a deterministic algorithm to solve a certain problem (since

real programs do not work nondeterministically) the concept of nondeterministism plays a crucial role in various theoretical considerations.

A *deterministic algorithm* is an algorithm where for any configuration it is uniquely determined whether the algorithm halts or not, which output (if any) is provided, or what the next step (and the successor configuration) is. A deterministic algorithm solves a given decision problem P if for each input w (1) the unique computation of the algorithm started with w terminates, and (2) returns the correct answer. An input w (for P), of course, needs to satisfy all criteria as given by the decision problem P, e.g., for the SAT problem, the input w is a propositional formula Φ. A deterministic algorithm solves the SAT problem if it provides the answer "yes" if Φ is satisfiable and "no" if Φ is not satisfiable.

The time complexity of a deterministic algorithm A is typically measured by a function $T_A : \mathbb{N} \to \mathbb{N}$ where $T_A(n)$ is the maximal number of "steps" that A performs on an input of size n. As the maximal number of steps is considered, T_A reflects the worst-case running time. Similarly, the space complexity is formalized by a function $S_A : \mathbb{N} \to \mathbb{N}$ where $S_A(n)$ is the maximal number of atomic memory units that are used for inputs of length n.[2]

Nondeterministic algorithms differ from deterministic algorithms in the sense that they may provide finitely many alternatives for the next step of a configuration. Thus, there might be several possible computations of a nondeterministic algorithm on a given input w. A nondeterministic algorithm is said to solve a decision problem P if for each input w:

(1) all computations for w terminate,

(2a) if the correct answer for w is "yes", then there is at least one computation for w that returns the answer "yes", and

(2b) if the correct answer for w is "no", then all computations for w return the answer "no".

An example of a nondeterministic algorithm for SAT is shown in Algorithm 50. Here, $sat(\mu, \Phi)$ is a subprocedure which computes deterministically the truth value of Φ under the evaluation μ (in time $\Theta(|\Phi|)$). It returns the Boolean value true if $\mu \models \Phi$ and false otherwise. The details of this algorithm *sat* are not of importance here. Algorithm 50 relies on the so-called "guess and check" principle which is the basis for many nondeterministic algorithms. After nondeterministically guessing an evaluation μ it is checked whether or not $\mu \models \Phi$. If the guess is correct, the evaluation is a witness for "$\mu \models \Phi$", and the answer

[2] For some complexity classes, $S_A(n)$ only takes into account the memory requirement that is needed in addition to the memory units for storing the input. As these complexity classes are not used in this monograph, this detail is ignored here.

Algorithm 50 Nondeterministic algorithm for SAT

Input: propositional logic formula Φ over $AP = \{a_1, \ldots, a_n\}$
Output: "yes" or "no"

```
                                          (* Guess an evalution μ for AP *)
for i = 1, ..., n do
  choose nondeterministically a truth value t ∈ {0, 1};
  μ(a_i) := t
od
                                          (* Check whether μ ⊨ Φ *)
if sat(μ, Φ) then
  return "yes"                            (* Φ is satisfiable *)
else
  return "no"                             (* Φ might or might not be satisfiable *)
fi
```

"yes" is correct. If, however, $\mu \not\models \Phi$, then the given answer "no" might be wrong, as there might be another evaluation μ' with $\mu' \models \Phi$. However, Algorithm 50 solves SAT since (1) it terminates for all input formulae Φ. The fact that it also fulfills conditions (2a) and (2b) can be seen as follows. Condition (2a) holds since whenever Φ is satisfiable, then there exists a computation which chooses an evalution under which Φ holds, and hence, the algorithm returns "yes". Condition (2b) holds since whenever Φ is not satisfiable, then for each choice of μ in the guess phase we have $\mu \not\models \Phi$ and the returned answer is "no".

The time complexity of a nondeterministic algorithm A that solves a decision problem (i.e., all computations terminate) is given by the function $T_A : \mathbb{N} \to \mathbb{N}$ where $T_A(n)$ is the maximum number of steps that A may perform for inputs of length n. Similarly, the space complexity of A is given by the function $S_A : \mathbb{N} \to \mathbb{N}$ where $S_A(n)$ is the maximum number of memory units that A may access on an input of length n.

Algorithm A (deterministic or nondeterministic) is called *polynomially time-bounded* (or briefly, a *polytime algorithm*) if $T_A(n) = \mathcal{O}(\text{poly}(n))$. Similarly, algorithm A is a polyspace algorithm if $S_A(n) = \mathcal{O}(\text{poly}(n))$.

Complexity Classes Using the above intuitive explanations of deterministic and nondeterministic algorithms, we are in a position to introduce the complexity classes used in this monograph:

PTIME (or briefly P) denotes the class of all decision problems that can be solved by a deterministic polytime algorithm.

NP denotes the class of all decision problems that can be solved by a nondeterministic polytime algorithm.

PSPACE denotes the class of all decision problems that can be solved by a deterministic polyspace algorithm.

The complexity class NPSPACE is the class of all decision problems that can be solved by a nondeterministic polyspace algorithm. It is known that PSPACE = NPSPACE. Since any deterministic algorithm can be viewed as a nondeterministic algorithm (where the number of alternative steps per configuration is at most one) and since in N steps at most N memory units can be used, it follows that:

$$\text{PTIME} \subseteq \text{NP} \subseteq \text{PSPACE} = \text{NPSPACE}.$$

One of the most important open questions in theoretical computer science is whether PTIME agrees with NP. It is also unknown so far whether NP agrees with PSPACE.

The next complexity class considered is coNP which is somehow "complementary" to NP. Let P be a decision problem. Then, the complementary problem of P, denoted $\overline{P}$, is the problem which takes the same inputs as P but requires the answer "no" if P requires the answer "yes", and vice versa. For example, the complementary problem of SAT takes as input a propositional logic formula Φ and asks whether Φ is not satisfiable. The complementary problem of the (directed) Hamiltonian path problem asks whether a given finite digraph G does not have a Hamiltonian path.

coNP denotes the class of all decision problems P where the complementary problem $\overline{P}$ belongs to NP.

coNP subsumes PTIME and is contained in PSPACE, while it is unknown whether these inclusions are strict. The complexity classes coPTIME and coPSPACE can be defined as the classes of the complementary problems of PTIME and PSPACE, respectively. However, the symmetry of the treatment of "yes" and "no" by deterministic algorithms allows swapping the outputs of a deterministic polytime (or polyspace) algorithm that solves problem P to obtain a deterministic polytime (polyspace) algorithm for $\overline{P}$. Hence, PTIME = coPTIME and PSPACE = coPSPACE. Note that this argument does not apply to nondeterministic complexity classes due to the asymmetry of "yes" and "no" answers according to (2a) and (2b). The question whether NP and coNP agree is an open problem.

Further complexity classes that will be needed in this monograph are EXPTIME and 2EXPTIME. EXPTIME denotes the class of decision problems P that are solvable by a deterministic algorithm where the worst-case time complexity is exponentially bounded, i.e., bounded above by a function $n \mapsto 2^{p(n)}$ where p is a polynomial. The complexity

class 2EXPTIME consists of all decision problems that can be solved by a deterministic algorithm for which the worst-case time complexity is bounded by some function $n \mapsto 2^{2^{p(n)}}$ where p is a polynomial. It holds that

$$\text{PSPACE} \subseteq \text{EXPTIME} \subseteq \text{2EXPTIME}.$$

Completeness To specify the "hardest" problem in NP, Cook [99] introduced in the early seventies the notion of NP-completeness. Decision problem $P \in \text{NP}$ is NP-complete if the existence of a deterministic polytime algorithm A for P, implies that any other problem $Q \in \text{NP}$ can be solved deterministically with the help of A plus a polytime transformation from Q's inputs to P's inputs. (The transformation of Q's inputs into P's inputs relies on the *reduction* principle.) Thus, if one NP-complete problem is shown to be in PTIME, then NP = PTIME.

Let us briefly consider these concepts in more detail. Let P and Q be decision problems. Problem Q is called *polynomially reducible* to P if there exists a polytime deterministic algorithm which transforms a given input w_Q for Q into an input w_P for P such that the correct answer for Q on w_Q is "yes" if and only if the correct answer for P on w_P is "yes". Decision problem P is said to be NP-*hard* if each problem $Q \in \text{NP}$ is polynomially reducible to P. P is called NP-*complete* if (1) P belongs to NP and (2) P is NP-hard. There is a wide range of NP-complete problems. For many cases, the proof of membership of P in NP is quite simple—one provides a polytime nondeterministic algorithm for P on the basis of the guess-and-check paradigm. The NP-hardness is often harder to prove. The first NP-completeness result was by Cook [99] who showed that the SAT problem is NP-complete. His proof for the NP-hardness of SAT is generic: it starts with a formalization of a polytime nondeterministic algorithm for some NP-problem Q by a nondeterministic Turing machine $\mathcal{M}$ and constructs for a given input word w for $\mathcal{M}$ a propositional formula $\Phi_{\mathcal{M},w}$ such that $\Phi_{\mathcal{M},w}$ is satisfiable if and only if $\mathcal{M}$ has an accepting computation for w. For proving the NP-hardness of a given problem P, it is often simpler to use the *reduction principle*:

if Q is NP-hard and Q is polynomially reducible to P, then P is NP-hard.

For instance, a polynomial reduction from SAT to 3SAT (a variant of SAT which takes as input a CNF formula Φ with at most three literals per clause and asks for the satisfiability of Φ) is not very involved and yields the NP-hardness of 3SAT. The NP-hardness of the Hamiltonian path problem can be proven by a polynomial reduction from 3SAT to the Hamiltonian path problem. The latter problem is again polynomially reducible to the three-coloring problem, a decision problem on finite undirected graphs that asks whether the vertices can be colored with three colors such that for no edge $\{ v, w \}$ the vertices v and w are equally colored. Hence, the three-coloring problem is NP-hard too. In this way, a wide range of NP-hardness results have been obtained.

Hardness or completeness can be defined in an analogous way for other complexity classes such as PSPACE and coNP. P is called PSPACE-*complete* if (1) P belongs to PSPACE and (2) P is PSPACE-hard, i.e., all problems in PSPACE are polynomially reducible to P. An example of a PSPACE-complete problem is the universality problem: does the language described by a regular expression over the alphabet Σ equal Σ^*? Similarly, P is called coNP-*complete* if (1) P belongs to coNP and (2) P is coNP-hard, i.e., all problems in coNP are polynomially reducible to P. The symmetry of NP and coNP yields that a problem P is NP-hard if and only if its complementary problem $\overline{P}$ is coNP-hard. Thus, P is NP-complete if and only if $\overline{P}$ is coNP-complete. For instance, the problem whether a given propositional logic formula Φ is valid is coNP-complete, since $\Phi \mapsto \neg\Phi$ yields a polynomial reduction from SAT to the validity problem—recall that Φ is not satisfiable if and only if $\neg\Phi$ is valid (see page 919)—and $\overline{\text{SAT}}$ is coNP-complete (as SAT is NP-complete). Since coNP and NP are contained in PSPACE, any problem that is hard for PSPACE is also coNP- and NP-hard.

Bibliography

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, **82**(2):253–284, 1991.

[2] Y. Abarbanel-Vinov and N. Aizenbud-Reshef and I. Beer and C. Eisner and D. Geist and T. Heyman and I. Reuveni and E. Rippel and I. Shitsevalov and Y. Wolfsthal and T. Yatzkar-Haham. On the effective deployment of functional formal verification. *Formal Methods in System Design*, **19**:35–44, 2001.

[3] P. A. Abdulla and B. Jonsson and M. Kindahl and D. Peled. A general approach to partial order reductions in symbolic verification (extended abstract). In *10th International Conference on Computer Aided Verification (CAV)*, volume 1427 of *Lecture Notes in Computer Science*, pages 379–390. Springer-Verlag, 1998.

[4] S. Abramsky. A domain equation for bisimulation. *Information and Computation*, **92**(2):161–218, 1991.

[5] B. Alpern and F. Schneider. Defining liveness. *Information Processing Letters*, **21**(4):181–185, 1985.

[6] B. Alpern and F. Schneider. Recognizing safety and liveness. *Distributed Computing*, **2**(3):117–126, 1987.

[7] B. Alpern and F. Schneider. Verifying temporal properties without temporal logic. *ACM Transactions on Programming Languages and Systems*, **11**(1):147–167, 1989.

[8] R. Alur and R. K. Brayton and T. Henzinger and S. Qadeer and S. K. Rajamani. Partial order reduction in symbolic state-space exploration. *Formal Methods in System Design*, **18**(2):97–116, 2001.

[9] R. Alur and C. Courcoubetis and D. Dill. Model-checking in dense real time. *Information and Computation*, **104**(2):2–34, 1993.

[10] R. Alur and D. Dill. Automata for modeling real-time systems. In *17th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer-Verlag, 1990.

[11] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, **126**(2):183–235, 1994.

[12] R. Alur and D. Dill. Automata-theoretic verification of real-time systems. In C. Heitmeyer and D. Mandrioli, editors, *Formal Methods for Real-Time Computing*, pages 55–82. John Wiley & Sons, 1996.

[13] R. Alur and L. Fix and T. A. Henzinger. Event-clock automata: a determinizable class of timed automata. *Theoretical Computer Science*, **211**(1–2):253–273, 1999.

[14] S. Andova and H. Hermanns and J.-P. Katoen. Discrete-time rewards model checked. In *1st International Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS)*, volume 2791 of *Lecture Notes in Computer Science*, pages 88–104. Springer-Verlag, 2003.

[15] K. R. Apt. Correctness proofs of distributed termination algorithms. *ACM Transactions on Programming Languages and Systems*, **8**(3):388–405, 1986.

[16] K. R. Apt and N. Francez and W.-P. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, **2**(3):359–385, 1980.

[17] K. R. Apt and D. Kozen. Limits for the automatic verification of finite-state concurrent systems. *Information Processing Letters*, **22**(6):307–309, 1986.

[18] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 1997.

[19] A. Arnold. *Finite Transition Systems*. Prentice-Hall, 1994.

[20] E. Asarin and P. Caspi and O. Maler. Timed regular expressions. *Journal of the ACM*, **49**(2):172–206, 2002.

[21] R. B. Ash and C. A. Doléans-Dade. *Probability and Measure Theory*. Academic Press, 2000.

[22] A. Aziz and K. Sanwal and V. Singhal and R. K. Brayton. Model-checking continous-time Markov chains. *ACM Transactions on Computer Logic*, **1**(1):162–170, 2000.

[23] A. Aziz and V. Singhal and F. Balarin and R. K. Brayton and A. L. Sangiovanni-Vincentelli. It usually works: The temporal logic of stochastic systems. In *7th International Conference on Computer Aided Verification (CAV)*, volume 939 of *Lecture Notes in Computer Science*, pages 155–165. Springer-Verlag, 1995.

[24] S. Baase and A. van Gelder. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, 2000.

[25] C. Baier and F. Ciesinski and M. Grösser. Probmela: a modeling language for communicating probabilistic systems. In *2nd ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 57–66. IEEE Computer Society Press, 2004.

[26] C. Baier and E. Clarke and V. Hartonas-Garmhausen and M. Kwiatkowska and M. Ryan. Symbolic model checking for probabilistic processes. In *24th International Colloqium on Automata, Languages and Programming (ICALP)*, volume 1256 of *Lecture Notes in Computer Science*, pages 430–440. Springer-Verlag, 1997.

[27] C. Baier and B. Engelen and M. E. Majster-Cederbaum. Deciding bisimilarity and similarity for probabilistic processes. *Journal of Computer and System Sciences*, **60**(1):187–231, 2000.

[28] C. Baier and M. Grösser and F. Ciesinski. Partial order reduction for probabilistic systems. In *1st International Conference on Quantitative Evaluation of Systems (QEST)*, pages 230–239. IEEE Computer Society Press, 2004.

[29] C. Baier and B. R. Haverkort and H. Hermanns and J.-P. Katoen. Model checking algorithms for continuous time Markov chains. *IEEE Transactions on Software Engineering*, **29**(6):524–541, 2003.

[30] C. Baier and J.-P. Katoen and H. Hermanns and V. Wolf. Comparative branching time semantics for Markov chains. *Information and Computation*, **200**(2):149–214, 2005.

[31] C. Baier and M. Kwiatkowska. Model checking for a probabilistic branching time logic with fairness. *Distributed Computing*, **11**(3):125–155, 1998.

[32] C. Baier and M. Kwiatkowska. On the verification of qualitative properties of probabilistic processes under fairness constraints. *Information Processing Letters*, **66**(2):71–79, 1998.

[33] T. Ball and A. Podelski and S. Rajamani. Boolean and Cartesian abstraction for model checking C programs. In *7th International Conference on Tools and*

Algorithms for the Construction and Analysis of Systems (TACAS), volume 2031 of *Lecture Notes in Computer Science*, pages 268–283. Springer-Verlag, 2001.

[34] K. A. Bartlett and R. A. Scantlebury and P. T. Wilkinson. A note on reliable full duplex transmission over half duplex links. *Communications of the ACM*, **12**(5):260–261, 1969.

[35] G. Behrmann and A. David and K. G. Larsen. A tutorial on Uppaal. In *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer-Verlag, 2004.

[36] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1990.

[37] F. Belina and D. Hogrefe and A. Sarma. *SDL with Applications from Protocol Specification*. Prentice-Hall, 1991.

[38] R. Bellman. A Markovian decision process. *Journal of Mathematics and Mechanics*, **38**:679–684, 1957.

[39] R. Bellman. Markovian decision processes. *Journal of Mathematics and Mechanics*, **38**:716–719, 1957.

[40] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, **16**(1):87–90, 1958.

[41] M. Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, **6**(3):333–344, 1984.

[42] M. Ben-Ari and Z. Manna and A. Pnueli. The temporal logic of branching time. *Acta Informatica*, **20**:207–226, 1983.

[43] J. Bengtsson and W. Yi. Timed automata: semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer-Verlag, 2003.

[44] B. Bérard and M. Bidoit and A. Finkel and F. Laroussinie and A. Petit and L. Petrucci and Ph. Schnoebelen. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer-Verlag, 2001.

[45] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, **37**:77–121, 1985.

[46] J. A. Bergstra and A. Ponse and S. A. Smolka (editors). *Handbook of Process Algebra*. Elsevier Publishers B.V., 2001.

[47] P. Berman and J. A. Garay. Asymptotically optimal distributed consensus (extended abstract). In *Automata, Languages and Programming (ICALP)*, volume 372 of *Lecture Notes in Computer Science*, pages 80–94. Springer-Verlag, 1989.

[48] B. Berthomieu and M. Menasche. An enumerative approach for analyzing time Petri nets. In *IFIP 9th World Computer Congress*, pages 41–46. North Holland, 1983.

[49] D. P. Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall, 1987.

[50] G. Bhat and R. Cleaveland and O. Grumberg. Efficient on-the-fly model checking for CTL*. In *10th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 388–397. IEEE Computer Society Press, 1995.

[51] A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In *15th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 1026 of *Lecture Notes in Computer Science*, pages 499–513. Springer-Verlag, 1995.

[52] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[53] B. W. Boehm and V. R. Basili. Software defect reduction top 10 list. *IEEE Computer*, **34**(1):135–137, 2001.

[54] H. Bohnenkamp and P. van der Stok and H. Hermanns and F.W. Vaandrager. Cost optimisation of the ipv4 zeroconf protocol. In *International Conference on Dependable Systems and Networks (DSN)*, pages 626–638. IEEE Computer Society Press, 2003.

[55] G. Bolch and S. Greiner and H. de Meer and K. S. Trivedi. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. John Wiley & Sons, 2006.

[56] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, **45**(9):993–1002, 1996.

[57] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, **14**(1):25–59, 1987.

[58] S. Bornot and J. Sifakis. An algebraic framework for urgency. *Information and Computation*, **163**(1):172–202, 2000.

[59] D. Bosnacki and G. Holzmann. Improving SPIN's partial-order reduction for breadth-first search. In *12th International SPIN Workshop on Model Checking of Software*, volume 3639 of *Lecture Notes in Computer Science*, pages 91–105. Springer-Verlag, 2005.

[60] A. Bouajjani and J.-C. Fernandez and N. Halbwachs. Minimal model generation. In *2nd International Workshop on Computer-Aided Verification (CAV)*, volume 531 of *Lecture Notes in Computer Science*, pages 197–203. Springer-Verlag, 1990.

[61] P. Bouyer. Untameable timed automata! In *20th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 2607 of *Lecture Notes in Computer Science*, pages 620–631. Springer-Verlag, 2003.

[62] R. K. Brayton and G. D. Hachtel and A. L. Sangiovanni-Vincentelli and F. Somenzi and A. Aziz and S.-T. Cheng and S. A. Edwards and S. P. Khatri and Y. Kukimoto and A. Pardo and S. Qadeer and R. K. Ranjan and S. Sarwary and T. R. Shiple and G. Swamy and T. Villa. VIS: a system for verification and synthesis. In *8th International Conference on Computer Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer-Verlag, 1996.

[63] P. Bremaud. *Markov Chains, Gibbs Fields, Monte Carlo Simulation and Queues*. Springer-Verlag, 1999.

[64] L. Brim and I. Černá and M. Nečesal. Randomization helps in LTL model checking. In *1st Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM-PROBMIV)*, volume 2165 of *Lecture Notes in Computer Science*, pages 105–119. Springer-Verlag, 2001.

[65] S. D. Brookes and C. A. R. Hoare and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, **31**(3):560–599, 1984.

[66] M. C. Browne and E. M. Clarke and D. L. Dill and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, **35**(12):1035–1044, 1986.

[67] M. C. Browne and E. M. Clarke and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, **59**(1–2):115–131, 1988.

[68] S. D. Bruda. Preorder relations. In M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, chapter 5, pages 115–148. Springer-Verlag, 2005.

[69] J. Brunekreef and J.-P. Katoen and R. Koymans and S. Mauw. Design and analysis of dynamic leader election protocols in broadcast networks. *Distributed Computing*, **9**(4):157–171, 1996.

[70] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, **35**(8):677–691, 1986.

[71] R. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, **40**(2):205–213, 1991.

[72] P. Buchholz. Exact and ordinary lumpability in Markov chains. *Journal of Applied Probability*, **31**:59–75, 1994.

[73] J. R. Büchi. On a decision method in restricted second order arithmetic. In *International Congress on Logic, Methodology and Philosophy of Science*, pages 1–11. Stanford University Press, 1962.

[74] J. Burch and E. Clarke and K. L. McMillan and D. L. Dill and L. Hwang. Symbolic model checking 10^{20} states and beyond. *Information and Computation*, **98**(2):142–170, 1992.

[75] J. Burch and E. M. Clarke and K. L. McMillan and D. L. Dill. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE Conference on Design Automation (DAC)*, pages 46–51. IEEE Computer Society Press, 1990.

[76] D. Bustan and O. Grumberg. Simulation-based minimization. *ACM Transactions on Computational Logic*, **4**(2):181–206, 2003.

[77] D. Bustan and S. Rubin and M. Y. Vardi. Verifying ω-regular properties of Markov chains. In *16th International Conference on Computer Aided Verification (CAV)*, volume 3114 of *Lecture Notes in Computer Science*, pages 189–201. Springer-Verlag, 2004.

[78] K. Cerans. Decidability of bisimulation equivalences for parallel timer processes. In *4th International Workshop on Computer Aided Verification (CAV)*, volume 663 of *Lecture Notes in Computer Science*, pages 302–315. Springer-Verlag, 1992.

[79] W. Chan and R. J. Anderson and P. Beame and S. Burns and F. Modugno and D. Notkin and J. D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, **24**(7):498–520, 1998.

[80] E. Chang and Z. Manna and A. Pnueli. The safety-progress classification. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, volume 94 of *NATO ASI Series F: Computer and Systems Sciences*, pages 143–202. Springer-Verlag, 1992.

[81] Y. Choueka. Theories of automata on ω-tapes. *Journal of Computer and System Sciences*, **8**:117–141, 1974.

[82] F. Ciesinski and C. Baier. LiQuor: a tool for qualititative and quantitative linear time analysis of reactive systems. In *3rd Conference on Quantitative Evaluation of Systems (QEST)*, pages 131–132. IEEE Computer Society Press, 2006.

[83] A. Cimatti and E. M. Clarke and F. Giunchiglia and M. Roveri. NuSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, **2**(4):410–425, 2000.

[84] E. M. Clarke and A. Biere and R. Raimi and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, **19**(1):7–34, 2001.

[85] E. M. Clarke and I. A. Draghicescu. Expressibility results for linear time and branching time logics. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time, and Partial Order in Logics and Model for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 428–437. Springer-Verlag, 1988.

[86] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[87] E. M. Clarke and E. A. Emerson and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, **8**(2):244–263, 1986.

[88] E. M. Clarke and O. Grumberg and K. Hamaguchi. Another look at LTL model checking. In *6th International Conference on Computer Aided Verification (CAV)*, volume 818 of *Lecture Notes in Computer Science*, pages 415–427. Springer-Verlag, 1994.

[89] E. M. Clarke and O. Grumberg and H. Hiraishi and S. Jha and D. E. Long and K. L. McMillan and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In *11th International Symposium on Computer Hardware Description Languages and their Applications*, pages 5–20. Kluwer Academic Publishers, 1993.

[90] E. M. Clarke and O. Grumberg and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, **16**(5):1512–1542, 1994.

[91] E. M. Clarke and O. Grumberg and K. L. McMillan and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *32nd ACM/IEEE Conference on Design Automation (DAC)*, pages 427–432. IEEE Computer Society Press, 1995.

[92] E. M. Clarke and O. Grumberg and D. Peled. *Model Checking*. MIT Press, 1999.

[93] E. M. Clarke and S. Jha and Y. Lu and H. Veith. Tree-like counterexamples in model checking. In *17th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 19–29. IEEE Computer Society Press, 2002.

[94] E. M. Clarke and R. Kurshan. Computer-aided verification. *IEEE Spectrum*, **33**(6):61–67, 1996.

[95] E. M. Clarke and H. Schlingloff. Model checking. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning (Volume II)*, chapter 24, pages 1635–1790. Elsevier Publishers B.V., 2000.

[96] E. M. Clarke and J. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, **28**(4):626–643, 1996.

[97] R. Cleaveland and J. Parrow and B. Steffen. The concurrency workbench: a semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, **15**(1):36–72, 1993.

[98] R. Cleaveland and O. Sokolsky. Equivalence and preorder checking for finite-state systems. In J. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 6, pages 391–424. Elsevier Publishers B.V., 2001.

[99] S. Cook. The complexity of theorem-proving procedures. In *3rd Annual ACM Symposium on Theory of Computing*, pages 151–158. ACM Press, 1971.

[100] T. H. Cormen and C. E. Leiserson and R. L. Rivest and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[101] F. Corradini and R. De Nicola and A. Labella. An equational axiomatization of bisimulation over regular expressions. *Journal of Logic and Computation*, **12**(2):301–320, 2002.

[102] C. Courcoubetis and M. Y. Vardi and P. Wolper and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, **1**(2–3):275–288, 1992.

[103] C. Courcoubetis and M. Yannakakis. Markov decision processes and regular events (extended abstract). In *17th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 443 of *Lecture Notes in Computer Science*, pages 336–349. Springer-Verlag, 1990.

[104] C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *Journal of the ACM*, **42**(4):857–907, 1995.

[105] P. Cousot and R. Cousot. On abstraction in software verification. In *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 37–56. Springer-Verlag, 2002.

[106] J.-M. Couvreur. On-the-fly verification of linear temporal logic. In *World Congress on Formal Methods (FM)*, volume 1708 of *Lecture Notes in Computer Science*, pages 253–271. Springer-Verlag, 1999.

[107] J.-M. Couvreur and A. Duret-Lutz and D. Poitrenaud. On-the-fly emptiness checks for generalized Büchi automata. In *12th International SPIN Workshop on Model Checking of Software*, volume 3639 of *Lecture Notes in Computer Science*, pages 143–158. Springer-Verlag, 2005.

[108] J.-M. Couvreur and N. Saheb and G. Sutre. An optimal automata approach to LTL model checking of probabilistic systems. In *10th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 2850 of *Lecture Notes in Computer Science*, pages 361–375. Springer-Verlag, 2003.

[109] D. Dams and R. Gerth and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, **19**(2):253–291, 1997.

[110] M. Daniele and F. Giunchiglia and M. Y. Vardi. Improved automata generation for linear temporal logic. In *11th International Conference on Computer Aided Verification (CAV)*, volume 1633 of *Lecture Notes in Computer Science*, pages 249–260. Springer-Verlag, 1999.

[111] P. R. D'Argenio and E. Brinksma. A calculus for timed automata. In *4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, volume 1135 of *Lecture Notes in Computer Science*, pages 110–129. Springer-Verlag, 1996.

[112] P. R. D'Argenio and B. Jeannet and H. Jensen and K. Larsen. Reachability analysis of probabilistic systems by successive renements. In *Proc. 1st Joint Int. Workshop Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM-PROBMIV)*, volume 2399 of *Lecture Notes in Computer Science*, pages 39–56, 2001.

[113] P. R. D'Argenio and P. Niebert. Partial order reduction on concurrent probabilistic programs. In *1st International Conference on Quantitative Evaluation of Systems (QEST)*, pages 240–249. IEEE Computer Society Press, 2004.

[114] L. de Alfaro. Temporal logics for the specification of performance and reliability. In *14th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 1200 of *Lecture Notes in Computer Science*, pages 165–176. Springer-Verlag, 1997.

[115] L. de Alfaro. *Formal Verification of Probabilistic Systems*. PhD thesis, Stanford University, Department of Computer Science, 1998.

[116] L. de Alfaro. How to specify and verify the long-run average behavior of probabilistic systems. In *Thirteenth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 454–465. IEEE Computer Society Press, 1998.

[117] L. de Alfaro. Computing minimum and maximum reachability times in probabilistic systems. In *10th Conference on Concurrency Theory (CONCUR)*, volume 1664 of *Lecture Notes in Computer Science*, pages 66–81. Springer-Verlag, 1999.

[118] W.-P. de Roever and F. S. de Boer and U. Hannemann and J. Hooman and Y. Lakhnech and M. Poel and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001.

[119] F. Dederichs and R. Weber. Safety and liveness from a methodological point of view. *Information Processing Letters*, **36**(1):25–30, 1990.

[120] S. Derisavi and H. Hermanns and W. H. Sanders. Optimal state-space lumping in Markov chains. *Information Processing Letters*, **87**(6):309–315, 2003.

[121] J. Desharnais and A. Edalat and P. Panangaden. Bisimulation for labelled Markov processes. *Information and Computation*, **179**(2):163–193, 2002.

[122] J. Desharnais and V. Gupta and R. Jagadeesan and P. Panangaden. Weak bisimulation is sound and complete for PCTL*. In *Thirteenth International Conference on Concurrency Theory (CONCUR)*, volume 2421 of *Lecture Notes in Computer Science*, pages 355–370. Springer-Verlag, 2002.

[123] J. Desharnais and V. Gupta and R. Jagadeesan and P. Panangaden. Approximating labelled Markov processes. *Information and Computation*, **184**(1):160–200, 2003.

[124] J. Desharnais and P. Panangaden. Continuous stochastic logic characterizes bisimulation of continuous-time Markov processes. *Journal of Algebraic and Logic Programming*, **56**(1–2):99–115, 2003.

[125] V. Diekert and Y. Métivier. Partial commutation and traces. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 457–533. Springer-Verlag, 1997.

[126] E. W. Dijkstra. Solutions of a problem in concurrent programming control. *Communications of the ACM*, **8**(9):569, 1965.

[127] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968.

[128] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, **1**:115–138, 1971.

[129] E. W. Dijkstra. Information streams sharing a finite buffer. *Information Processing Letters*, **1**(5):179–180, 1972.

[130] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[131] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *International Workshop on Automatic Verification Methods for Finite-State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer-Verlag, 1989.

[132] D. L. Dill. The Murφ verifier. In *8th International Conference on Computer Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393. Springer-Verlag, 1996.

[133] J. Dingel and T. Filkorn. Model checking for infinite state systems using data abstraction, assumption commitment style reasoning and theorem proving. In *7th International Conference on Computer Aided Verification (CAV)*, volume 939 of *Lecture Notes in Computer Science*, pages 54–69. Springer-Verlag, 1995.

[134] R. Drechsler and B. Becker. *Binary Decision Diagrams: Theory and Implementation*. Kluwer Academic Publishers, 1998.

[135] S. Edelkamp and A. Lluch Lafuente and S. Leue. Directed explicit model checking with HSF-SPIN. In *8th International SPIN Workshop on Model Checking of Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 57–79. Springer-Verlag, 2001.

[136] C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Series on Integrated Circuits and Systems. Springer, 2006.

[137] T. Elrad and N. Francez. Decomposition of distributed programs into communication-closed layers. *Science of Computer Programming*, **2**(3):155–173, 1982.

[138] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, vol B: Formal Models and Semantics*. Elsevier Publishers B.V., 1990.

[139] E. A. Emerson and J. Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, **30**(1):1–24, 1985.

[140] E. A. Emerson and J. Y. Halpern. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM*, **33**(1):151–178, 1986.

[141] E. A. Emerson and C. S. Jutla. The complexity of tree automata and logics of programs (extended abstract). In *29th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 328–337. IEEE Computer Society Press, 1988.

[142] E. A. Emerson and C.-L. Lei. Temporal reasoning under generalized fairness constraints. In *3rd Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 210 of *Lecture Notes in Computer Science*, pages 21–36. Springer-Verlag, 1986.

[143] E. A. Emerson and C.-L. Lei. Modalities for model checking: branching time logic strikes back. *Science of Computer Programming*, **8**(3):275–306, 1987.

[144] J. Engelfriet. Branching processes of Petri nets. *Acta Informatica*, **28**(6):575–591, 1991.

[145] J. Esparza. Model checking using net unfoldings. *Science of Computer Programming*, **23**(2–3):151–195, 1994.

[146] K. Etessami. Stutter-invariant languages, omega-automata, and temporal logic. In *11th International Conference on Computer Aided Verification (CAV)*, volume 1633 of *Lecture Notes in Computer Science*, pages 236–248. Springer-Verlag, 1999.

[147] K. Etessami. A note on a question of Peled and Wilke regarding stutter-invariant LTL. *Information Processing Letters*, **75**(6):261–263, 2000.

[148] K. Etessami and G. Holzmann. Optimizing Büchi automata. In *11th International Conference on Concurrency Theory (CONCUR)*, volume 1877 of *Lecture Notes in Computer Science*, pages 153–165. Springer-Verlag, 2000.

[149] K. Etessami and T. Wilke and R. Schuller. Fair simulation relations, parity games, and state space reduction for Büchi automata. *SIAM Journal of Computing*, **34**(5):1159–1175, 2005.

[150] W. Feller. *An Introduction to Probability Theory and Its Applications,* volumes 1 and 2. John Wiley & Sons, 2001.

[151] C. Fencott. *Formal Methods for Concurrency.* Thomson Computer Press, 1995.

[152] K. Fisler and M. Y. Vardi. Bisimulation minimization in an automata-theoretic verification framework. In *2nd International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, volume 1522 of *Lecture Notes in Computer Science*, pages 115–132. Springer-Verlag, 1998.

[153] K. Fisler and M. Y. Vardi. Bisimulation and model checking. In *10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 1703 of *Lecture Notes in Computer Science*, pages 338–341. Springer-Verlag, 1999.

[154] K. Fisler and M. Y. Vardi. Bisimulation minimization and symbolic model checking. *Formal Methods in System Design*, **21**(1):39–78, 2002.

[155] N. Francez. *Fairness*. Springer-Verlag, 1986.

[156] L.-A. Fredlund. The timing and probability workbench: a tool for analysing timed processes. Technical Report 49, Uppsala University, 1994.

[157] C. Fritz and T. Wilke. State space reductions for alternating Büchi automata. In *22th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 2556 of *Lecture Notes in Computer Science*, pages 157–168. Springer-Verlag, 2002.

[158] D. Gabbay and I. Hodkinson and M. Reynolds. *Temporal Logic: Mathematical Foundations and Computational Aspects*, volume 1. Oxford University Press, 1994.

[159] D. Gabbay and A. Pnueli and S. Shelah and J. Stavi. On the temporal basis of fairness. In *7th Symposium on Principles of Programming Languages (POPL)*, pages 163–173. ACM Press, 1980.

[160] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[161] P. Gastin and P. Moro and M. Zeitoun. Minimization of counterexamples in SPIN. In *11th International SPIN Workshop on Model Checking of Software*, volume 2989 of *Lecture Notes in Computer Science*, pages 92–108. Springer-Verlag, 2004.

[162] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *Thirteenth International Conference on Computer Aided Verification (CAV)*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer-Verlag, 2001.

[163] J. Geldenhuys and A. Valmari. More efficient on-the-fly LTL verification with Tarjan's algorithm. *Theoretical Computer Science*, **345**(1):60–82, 2005.

[164] R. Gerth. Transition logic: how to reason about temporal properties in a compositional way. In *16th Annual ACM Symposium on Theory of Computing (STOC)*, pages 39–50. ACM Press, 1984.

[165] R. Gerth and R. Kuiper and D. Peled and W. Penczek. A partial order approach to branching time logic model checking. In *3rd Israel Symposium on the Theory of Computing Systems (ISTCS)*, pages 130–139. IEEE Computer Society Press, 1995.

[166] R. Gerth and D. Peled and M. Y. Vardi and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18. Chapman & Hall, 1995.

[167] D. Giannakopoulou and F. Lerda. From states to transitions: improving translation of LTL formulae to Büchi automata. In *22nd IFIP International Conference on Formal Techniques for Networked and Distributed Systems*, volume 2529 of *Lecture Notes in Computer Science*, pages 308–326. Springer-Verlag, 2002.

[168] P. Godefroid. Using partial orders to improve automatic verification methods. In *2nd International Workshop on Computer Aided Verification (CAV)*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185. Springer-Verlag, 1990.

[169] P. Godefroid. *Partial Order Methods for the Verification of Concurrent Systems: An Approach to the State Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[170] P. Godefroid. Model checking for programming languages using Verisoft. In *24th Annual Symposium on Principles of Programming Languages (POPL)*, pages 174–186. ACM Press, 1997.

[171] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In *5nd International Workshop on Computer Aided Verification (CAV)*, volume 697 of *Lecture Notes in Computer Science*, pages 438–449. Springer-Verlag, 1993.

[172] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in Systems Design*, **2**(2):149–164, 1993.

[173] R. Gotzhein. Temporal logic and applications: a tutorial. *Computer Networks and ISDN Systems*, **24**(3):203–218, 1992.

[174] E. Grädel and W. Thomas and T. Wilke (editors). *Automata Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[175] W. D. Griffioen and F. Vaandrager. A theory of normed simulations. *ACM Transactions on Computational Logic*, **5**(4):577–610, 2004.

[176] J. F. Groote and F. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In *17th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 443 of *Lecture Notes in Computer Science*, pages 531–540. Springer-Verlag, 1990.

[177] J. F. Groote and J. van de Pol. State space reduction using partial tau-confluence. In *25th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 1893 of *Lecture Notes in Computer Science*, pages 383–393. Springer-Verlag, 2000.

[178] H. Gumm. Another glance at the Alpern-Schneider characterization of safety and liveness in concurrent executions. *Information Processing Letters*, **47**(6):291–294, 1993.

[179] A. Gupta. Formal hardware verification methods: a survey. *Formal Methods in System Design*, **1**(2–3):151–238, 1992.

[180] G. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, 1996.

[181] G. D. Hachtel and E. Macii and A. Pardo and F. Somenzi. Markovian analysis of large finite-state machines. *IEEE Transactions on CAD of Integrated Circuits and Systems*, **15**(12):1479–1493, 1996.

[182] J. Hajek. Automatically verified data transfer protocols. In *4th International Conference on Computer Communication (ICCC)*, pages 749–756. IEEE Computer Society Press, 1978.

[183] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1992.

[184] M. Hammer and A. Knapp and S. Merz. Truly on-the-fly LTL model checking. In *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *Lecture Notes in Computer Science*, pages 191–205. Springer-Verlag, 2005.

[185] T. Han and J.-P. Katoen. Counterexamples in probabilistic model checking. In *Thirteenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *Lecture Notes in Computer Science*, pages 72–86. Springer-Verlag, 2007.

[186] H. Hansson. *Time and Probability in Formal Design of Distributed Systems*. Series in Real-Time Safety Critical Systems. Elsevier Publishers B.V., 1994.

[187] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, **6**(5):512–535, 1994.

[188] F. Harary. *Graph Theory*. Addison-Wesley, 1969.

[189] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, **8**(3):231–274, 1987.

[190] S. Hart and M. Sharir. Probabilistic propositional temporal logics. *Information and Control*, **70**(2–3):97–155, 1986.

[191] S. Hart and M. Sharir and A. Pnueli. Termination of probabilistic concurrent programs. *ACM Transactions on Programming Languages and Systems*, **5**(3):356–380, 1983.

[192] V. Hartonas-Garmhausen and S. Campos and E. M. Clarke. ProbVerus: probabilistic symbolic model checking. In *5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems (ARTS)*, volume 1601 of *Lecture Notes in Computer Science*, pages 96–110. Springer-Verlag, 1999.

[193] J. Hatcliff and M. Dwyer. Using the Bandera tool set to model-check properties of concurrent Java software. In *12th International Conference on Concurrency Theory (CONCUR)*, volume 2154 of *Lecture Notes in Computer Science*, pages 39–58. Springer-Verlag, 2001.

[194] K. Havelund and M. Lowry and J. Penix. Formal analysis of a space-craft controller using SPIN. *IEEE Transactions on Software Engineering*, **27**(8):749–765, 2001.

[195] K. Havelund and T. Pressburger. Model checking Java programs using Java Pathfinder. *International Journal on Software Tools for Technology Transfer*, **2**(4):366–381, 2000.

[196] B. R. Haverkort. *Performance of Computer Communication Systems: A Model-Based Approach*. John Wiley & Sons, 1998.

[197] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, **32**(1):137–161, 1985.

[198] M. R. Henzinger and T. A. Henzinger and P. W. Kopke. Computing simulations on finite and infinite graphs. In *36th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 453–462. IEEE Computer Society Press, 1995.

[199] T. Henzinger and R. Majumdar and J.-F. Raskin. A classification of symbolic transition systems. *ACM Transactions on Computational Logic*, **6**(1):1–32, 2005.

[200] T. A. Henzinger and X. Nicollin and J. Sifakis and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, **111**(2):193–244, 1994.

[201] H. Hermanns and J.-P. Katoen and J. Meyer-Kayser and M. Siegle. A tool for model-checking Markov chains. *International Journal on Software Tools for Technology Transfer*, **4**(2):153–172, 2003.

[202] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, **21**(8):666–677, 1978.

[203] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[204] R. Hojati and R. K. Brayton and R. P. Kurshan. BDD-based debugging of designs using language containment and fair CTL. In *5th International Conference on Computer Aided Verification (CAV)*, volume 697 of *Lecture Notes in Computer Science*, pages 41–58. Springer-Verlag, 1993.

[205] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1990.

[206] G. J. Holzmann. Design and validation of protocols: a tutorial. *Computer Networks and ISDN Systems*, **25**(9):981–1017, 1993.

[207] G. J. Holzmann. The theory and practice of a formal method: NewCoRe. In *IFIP World Congress*, pages 35–44. North Holland, 1994.

[208] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, **23**(5):279–295, 1997.

[209] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[210] G. J. Holzmann and E. Najm and A. Serhrouchini. SPIN model checking: an introduction. *International Journal on Software Tools for Technology Transfer*, **2**(4):321–327, 2000.

[211] G. J. Holzmann and D. Peled. An improvement in formal verification. In *7th IFIP WG6.1 International Conference on Formal Description Techniques (FORTE)*, pages 197–211. Chapman & Hall, 1994.

[212] G. J. Holzmann and D. Peled and M. Yannakakis. On nested depth-first search. In *2nd International SPIN workshop on Model Checking of Software*, pages 23–32. AMS Press, 1996.

[213] J. E. Hopcroft. An $n \log n$ algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[214] J. E. Hopcroft and R. Motwani and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 2001.

[215] R. A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, 1960.

[216] R. A. Howard. *Dynamic Probabilistic Systems,* volume 2*: Semi-Markov and Decision Processes*. John Wiley & Sons, 1972.

[217] D. A. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, **257**(3–4):161–190, 275–303, 1954.

[218] M. Huhn and P. Niebert and H. Wehrheim. Partial order reductions for bisimulation checking. In *18th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 1530 of *Lecture Notes in Computer Science*, pages 271–282. Springer-Verlag, 1998.

[219] M. Huth and M. D. Ryan. *Logic in Computer Science – Modelling and Reasoning about Systems*. Cambridge University Press, 1999.

[220] T. Huynh and L. Tian. On some equivalence relations for probabilistic processes. *Fundamenta Informaticae*, **17**(3):211–234, 1992.

[221] H. Hyman. Comments on a problem in concurrent programming control. *Communications of the ACM*, **9**(1):45, 1966.

[222] ISO/ITU-T. *Formal Methods in Conformance Testing*. International Standard, 1996.

[223] A. Itai and M. Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, **88**(1):60–87, 1990.

[224] H. Iwashita and T. Nakata and F. Hirose. CTL model checking based on forward state traversal. In *International Conference on Computer-Aided Design (ICCAD)*, pages 82–87. IEEE Computer Society Press, 1996.

[225] W. Janssen and J. Zwiers. Specifying and proving communication closedness in protocols. In *Thirteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification*, pages 323–339. North Holland, 1993.

[226] B. Jeannet and P. R. D'Argenio and K. G. Larsen. RAPTURE: a tool for verifying Markov decision processes. In *Tools Day, International Conference on Concurrency Theory (CONCUR)*, 2002.

[227] B. Jonsson and K. G. Larsen. Specification and refinement of probabilistic processes. In *6th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 266–277. IEEE Computer Society Press, 1991.

[228] B. Jonsson and W. Yi and K. G. Larsen. Probabilistic extensions of process algebras. In J. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 11, pages 685–711. Elsevier Publishers B.V., 2001.

[229] M. Kaminski. A classification of omega-regular languages. *Theoretical Computer Science*, **36**:217–229, 1985.

[230] J. A. W. Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California, Los Angeles, 1968.

[231] P. Kanellakis and S. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, **86**(1):43–68, 1990.

[232] J.-P. Katoen and T. Kemna and I. S. Zapreev and D. N. Jansen. Bisimulation minimisation mostly speeds up probabilistic model checking. In *Thirteenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *Lecture Notes in Computer Science*, pages 87–102. Springer-Verlag, 2007.

[233] J.-P. Katoen and M. Khattri and I. S. Zapreev. A Markov reward model checker. In *2nd International Conference on Quantitative Evaluation of Systems (QEST)*, pages 243–244. IEEE Computer Society Press, 2005.

[234] S. Katz and D. Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, **101**(2):337–359, 1992.

[235] S. Katz and D. Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, **6**(2):107–120, 1992.

[236] R. M. Keller. Formal verification of parallel programs. *Communications of the ACM*, **19**(7):371–384, 1976.

[237] J. Kemeny and J. Snell. *Finite Markov Chains*. D. Van Nostrand, 1960.

[238] J. Kemeny and J. Snell. *Denumerable Markov Chains*. D. Van Nostrand, 1976.

[239] E. Kindler. Safety and liveness properties: a survey. *Bulletin of the European Association for Theoretical Computer Science*, **53**:268–272, 1994.

[240] S. C. Kleene. Representation of events in nerve nets and finite automata. In C. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–42. Princeton University Press, 1956.

[241] J. Klein and C. Baier. Experiments with deterministic ω-automata for formulas of linear temporal logic. *Theoretical Computer Science*, **363**(2):182–195, 2006.

[242] D. E. Knuth and A. C. Yao. The complexity of nonuniform random number generation. In J.E. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pages 357–428. Academic Press, New York, 1976.

[243] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, **27**:333–354, 1983.

[244] S. A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, **16**:83–94, 1963.

[245] F. Kröger. *Temporal Logic of Programs*, volume 8 of *Springer Monographs on Theoretical Computer Science*. Springer-Verlag, 1987.

[246] T. Kropf. *Introduction to Formal Hardware Verification*. Springer-Verlag, 1999.

[247] A. Kucera and P. Schnoebelen. A general approach to comparing infinite-state systems with their finite-state specifications. *Theoretical Computer Science*, **358**(2-3):315–333, 2006.

[248] V. Kulkarni. *Modeling and Analysis of Stochastic Systems*. Chapman & Hall, 1995.

[249] O. Kupferman and M.Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, **19**(3):291–314, 2001.

[250] R. Kurshan. *Computer-aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[251] R. Kurshan and V. Levin and M. Minea and D. Peled and H. Yenigün. Combining software and hardware verification techniques. *Formal Methods in System Design*, **21**(3):251–280, 2002.

[252] M. Kwiatkowska. Survey of fairness notions. *Information and Software Technology*, **31**(7):371–386, 1989.

[253] M. Kwiatkowska. A metric for traces. *Information Processing Letters*, **35**(3):129–135, 1990.

[254] M. Kwiatkowska and G. Norman and D. Parker. Modelling and verification of probabilistic systems. In P. Panangaden and F. van Breugel, editors, *Part 2 of Mathematical Techniques for Analyzing Concurrent and Probabilistic Systems*, volume 23 of *CRM Monograph Series*. AMS Press, 2004.

[255] M. Kwiatkowska and G. Norman and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. *International Journal on Software Tools for Technology Transfer*, **6**(2):128–142, 2004.

[256] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, **17**(8):453–455, 1974.

[257] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, **3**(2):125–143, 1977.

[258] L. Lamport. Time, clocks and the ordering of events in distributed systems. *Communication of the ACM*, **21**(7):558–565, 1978.

[259] L. Lamport. "Sometime" is sometimes "not never" – on the temporal logic of programs. In *7th Annual Symposium on Principles of Programming Languages (POPL)*, pages 174–185. ACM Press, 1980.

[260] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, **16**(3):872–923, 1994.

[261] L. H. Landweber. Decision problems for omega-automata. *Mathematical Systems Theory*, **3**(4):376–384, 1969.

[262] F. Laroussinie and N. Markay and Ph. Schnoebelen. Temporal logic with forgettable past. In *17th IEEE Symposium on Logic in Computer Science (LICS)*, pages 383–392. IEEE Computer Society Press, 2002.

[263] K. G. Larsen and J. Pearson and C. Weise and W. Yi. Clock difference diagrams. *Nordic Journal of Computing*, **6**(3):271–298, 1999.

[264] K. G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, **94**(1):1–28, 1991.

[265] K. G. Larsen and W. Yi. Time-abstracted bisimulation: implicit specification and decidability. In *9th International Conference on the Mathematical Foundations of Programming Semantics (MFPS)*, volume 802 of *Lecture Notes in Computer Science*, pages 160–176. Springer-Verlag, 1993.

[266] D. Lee and M. Yannakakis. Online minimization of transition systems. In *24th Annual ACM Symposium on Theory of Computing (STOC)*, pages 264–274. ACM Press, 1992.

[267] D. Lehmann and A. Pnueli and J. Stavi. Impartiality, justice and fairness: the ethics of concurrent termination. In *8th Colloquium on Automata, Languages and Programming (ICALP)*, volume 115 of *Lecture Notes in Computer Science*, pages 264–277. Springer-Verlag, 1981.

[268] D. Lehmann and M. Rabin. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In *8th ACM Symposium on Principles of Programming Languages (POPL)*, pages 133–138. ACM Press, 1981.

[269] N. Leveson. *Safeware: System Safety and Computers*. ACM Press, 1995.

[270] C. Lewis. Implication and the algebra of logic. *Mind,* **N. S.**, **12**(84):522–531, 1912.

[271] H. R. Lewis. A logic of concrete time intervals (extended abstract). In *5th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 380–389. IEEE Computer Society Press, 1990.

[272] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, 1997.

[273] O. Lichtenstein and A. Pnueli. Checking that finite-state concurrent programs satisfy their linear specification. In *12th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 97–107. ACM Press, 1985.

[274] O. Lichtenstein and A. Pnueli and L. Zuck. The glory of the past. In *Conference on Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer-Verlag, 1985.

[275] P. Liggesmeyer and M. Rothfelder and M. Rettelbach and T. Ackermann. Qualitätssicherung Software-basierter technischer Systeme. *Informatik Spektrum*, **21**(5):249–258, 1998.

[276] R. Lipton. Reduction: a method of proving properties of parallel programs. *Communications of the ACM*, **18**(12):717–721, 1975.

[277] C. Loiseaux and S. Graf and J. Sifakis and A. Bouajjani and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, **6**(1):11–44, 1995.

[278] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. *Software Concepts and Tools*, **17**(3):93–102, 1996.

[279] N. Lynch and F. Vaandrager. Forward and backward simulations – part I: untimed systems. *Information and Computation*, **121**(2):214–233, 1993.

[280] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

[281] O. Maler and Z. Manna and A. Pnueli. From timed to hybrid systems. In *Real-Time: Theory in Practice, REX Workshop*, volume 600 of *Lecture Notes in Computer Science*, pages 447–484. Springer-Verlag, 1992.

[282] Z. Manna and A. Pnueli. Completing the temporal picture. *Theoretical Computer Science*, **83**(1):97–130, 1991.

[283] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1992.

[284] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Safety*. Springer-Verlag, 1995.

[285] P. Manolios and R. Trefler. Safety and liveness in branching time. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 366–372. IEEE Computer Society Press, 2001.

[286] P. Manolios and R. J. Trefler. A lattice-theoretic characterization of safety and liveness. In *22nd Annual Symposium on Principles of Distributed Computing (PODC)*, pages 325–333. IEEE Computer Society Press, 2003.

[287] A. Mazurkiewicz. Trace theory. In *Advances in Petri Nets*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer-Verlag, 1987.

[288] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[289] K. L. McMillan. A technique of state space search based on unfoldings. *Formal Methods in System Design*, **6**(1):45–65, 1995.

[290] R. McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, **9**(5):521–530, 1966.

[291] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, **34**:1045–1079, 1955.

[292] C. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design*. Springer-Verlag, 1998.

[293] S. Merz. Model checking: a tutorial. In F. Cassez, C. Jard, B. Rozoy, and M.D. Ryan, editors, *Modelling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 3–38. Springer-Verlag, 2001.

[294] S. Merz and N. Navet (editors). *Modeling and Verification of Real-Time Systems: Formalisms and Software Tools*. ISTE Ltd, 2008.

[295] R. Milner. An algebraic definition of simulation between programs. In *2nd International Joint Conference on Artificial Intelligence*, pages 481–489. William Kaufmann, 1971.

[296] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[297] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, **25**(3):267–310, 1983.

[298] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[299] R. Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999.

[300] S. Minato. *Binary Decision Diagrams and Applications for VLSI CAD*. Kluwer Academic Publishers, 1996.

[301] S. Minato and N. Ishiura and S. Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *27th ACM/IEEE Conference on Design Automation (DAC)*, pages 52–57. ACM Press, 1991.

[302] F. Moller and S. A. Smolka. On the computational complexity of bisimulation. *ACM Computing Surveys*, **27**(2):287–289, 1995.

[303] E. F. Moore. Gedanken-experiments on sequential machines. *Automata Studies*, **34**:129–153, 1956.

[304] C. Morgan and A. McIver. pGCL: Formal reasoning for random algorithms. *South African Computer Journal*, **22**:14–27, 1999.

[305] A. W. Mostowski. Regular expressions for infinite trees and a standard form of automata. In *5th Symposium on Computational Theory*, volume 208 of *Lecture Notes in Computer Science*, pages 157–168. Springer-Verlag, 1984.

[306] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[307] D. E. Muller. Infinite sequences and finite machines. In *4th IEEE Symposium on Switching Circuit Theory and Logical Design*, pages 3–16. IEEE, 1963.

[308] G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, 1979.

[309] J. Myhill. Finite automata and the representation of events. Technical Report WADD TR-57-624, Wright Patterson Air Force Base, OH, 1957.

[310] R. Nalumasu and G. Gopalakrishnan. A new partial order reduction algorithm for concurrent systems. In *Thirteenth IFIP International Conference on Hardware Description Languages and their Applications (CHDL)*, pages 305–314. Chapman & Hall, 1997.

[311] K. S. Namjoshi. A simple characterization of stuttering bisimulation. In *17th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 1346 of *Lecture Notes in Computer Science*, pages 284–296. Springer-Verlag, 1997.

[312] G. Naumovich and L. A. Clarke. Classifying properties: an alternative to the safety-liveness classification. *ACM SIGSOFT Software Engineering Notes*, **25**(6):159–168, 2000.

[313] A. Nerode. Linear automaton transformations. In *Proceedings of the American Mathematical Society*, volume 9, pages 541–544, 1958.

[314] R. De Nicola and F. Vaandrager. Three logics for branching bisimulation (extended abstract). In *5th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 118–129. IEEE Computer Society Press, Springer-Verlag, 1990.

[315] X. Nicollin and J.-L. Richier and J. Sifakis and J. Voiron. ATP: an algebra for timed processes. In *IFIP TC2 Working Conference on Programming Concepts and Methods*, pages 402–427. North Holland, 1990.

[316] A. Olivero and J. Sifakis and S. Yovine. Using abstractions for the verification of linear hybrid systems. In *6th International Conference on Computer Aided Verification (CAV)*, volume 818 of *Lecture Notes in Computer Science*, pages 81–94. Springer-Verlag, 1994.

[317] S. Owicki. Verifying concurrent programs with shared data classes. In *IFIP Working Conference on Formal Description of Programming Concepts*, pages 279–298. North Holland, 1978.

[318] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, **16**(6):973–989, 1987.

[319] P. Panangaden. Measure and probability for concurrency theorists. *Theoretical Computer Science*, **253**(2):287–309, 2001.

[320] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[321] D. Park. On the semantics of fair parallelism. In *Abstract Software Specification*, volume 86 of *Lecture Notes in Computer Science*, pages 504–526. Springer-Verlag, 1979.

[322] D. Park. Concurrency and automata on infinite sequences. In *5th GI-Conference on Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1981.

[323] D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, UK, 2002.

[324] D. Peled. All from one, one for all: On model checking using representatives. In *5th International Conference on Computer Aided Verification (CAV)*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag, 1993.

[325] D. Peled. Combining partial order reductions with on-the-fly model checking. *Formal Methods in System Design*, **8**(1):39–64, 1996.

[326] D. Peled. Partial order reduction: Linear and branching temporal logics and process algebras. In *Partial Order Methods in Verification* [328], pages 79–88.

[327] D. Peled. *Software Reliability Methods*. Springer-Verlag, 2001.

[328] D. Peled and V. Pratt and G. J. Holzmann (editors). *Partial Order Methods in Verification*, volume 29 (10) of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. AMS Press, 1997.

[329] D. Peled and T. Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters*, **63**(5):243–246, 1997.

[330] W. Penczek and R. Gerth and R. Kuiper and M. Szreter. Partial order reductions preserving simulations. In *8th Workshop on Concurrency, Specification and Programming (CS&P)*, pages 153–172. Warsaw University Press, 1999.

[331] G. Della Penna and B. Intrigila and I. Melatti and E. Tronci and M. Venturini Zilli. Finite horizon analysis of Markov chains with the Murphi verifier. *Journal on Software Tools and Technology Transfer*, **8**(4-5):397–409, 2006.

[332] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, **12**(3):15–116, 1981.

[333] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981.

[334] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[335] G. D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, **60–61**:3–15, 2005.

[336] G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, **60–61**:17–139, 2005.

[337] A. Pnueli. The temporal logic of programs. In *18th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 46–67. IEEE Computer Society Press, 1977.

[338] A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In *12th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 194 of *Lecture Notes in Computer Science*, pages 15–32. Springer-Verlag, 1985.

[339] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In *Advanced School on Current Trends in Concurrency Theorey*, volume 244 of *Lecture Notes in Computer Science*, pages 510–584. Springer-Verlag, 1986.

[340] A. Pnueli and L. Zuck. Probabilistic verification by tableaux. In *1st Annual Symposium on Logic in Computer Science (LICS)*, pages 322–331. IEEE Computer Society Press, 1986.

[341] A. Pnueli and L. Zuck. Probabilistic verification. *Information and Computation*, **103**(1):1–29, 1993.

[342] H. Pospesel. *Introduction to Logic: Propositional Logic*. Prentice-Hall, 1979.

[343] V. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, **15**(1):33–71, 1986.

[344] W. Press and S. A. Teukolsky and W. T. Vetterling and B. P. Flannery. *Numerical Recipes in C++. The Art of Scientific Computing*. Cambridge University Press, 2002.

[345] A. Prior. *Time and Modality*. Oxford University Press, 1957.

[346] M. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 1994.

[347] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *5th International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1982.

[348] J.-P. Queille and J. Sifakis. Fairness and related properties in transition systems. a temporal logic to deal with fairness. *Acta Informatica*, **19**(3):195–220, 1983.

[349] M. O. Rabin. Probabilistic algorithms. In J. F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pages 21–39. Academic Press, 1976.

[350] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, **3**(2):114–125, 1959.

[351] M.O. Rabin. Decidability of second order theories and automata on infinite trees. *Transactions of the AMS*, **141**:1–35, 1969.

[352] Y. Ramakrishna and S. Smolka. Partial-order reduction in the weak modal mu-calculus. In *8th International Conference on Concurrency Theory (CONCUR)*, volume 1243 of *Lecture Notes in Computer Science*, pages 5–24. Springer-Verlag, 1997.

[353] J. I. Rasmussen and K. G. Larsen and K. Subramani. On using priced timed automata to achieve optimal scheduling. *Formal Methods in System Design*, **29**(1):97–114, 2006.

[354] M. Rem. Trace theory and systolic computations. In *Parallel Architectures and Languages Europe (PARLE),* volume 1, volume 258 of *Lecture Notes in Computer Science*, pages 14–33. Springer-Verlag, 1987.

[355] M. Rem. A personal perspective of the Alpern-Schneider characterization of safety and liveness. In W. H. J. Feijen, A. J. M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is Our Business: A Birthday Salute to Edsger W. Dijkstra*, chapter 43, pages 365–372. Springer-Verlag, 1990.

[356] A. W. Roscoe. Model-checking CSP. In A. W. Roscoe, editor, *A Classical Mind: Essays in Honour of C. A. R. Hoare*, pages 353–378. Prentice-Hall, 1994.

[357] G. Rozenberg and V. Diekert. *The Book of Traces*. World Scientific Publishing Co., Inc., 1995.

[358] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *International Conference on Computer-Aided Design (ICCAD)*, pages 42–47. IEEE Computer Society Press, 1993.

[359] J. Rushby. Formal methods and the certification of critical systems. Technical Report SRI-CSL-93-7, SRI International, 1993. (also issued as *Formal Methods and Digital System Validation*, NASA CR 4551).

[360] T. C. Ruys and E. Brinksma. Managing the verification trajectory. *International Journal on Software Tools for Technology Transfer*, **4**(2):246–259, 2003.

[361] S. Safra. On the complexity of ω-automata. In *29th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 319–327. IEEE Computer Society Press, 1988.

[362] A. L. Sangiovanni-Vincentelli and P. C. McGeer and A. Saldanha. Verification of electronic systems. In *33rd Annual Conference on Design Automation (DAC)*, pages 106–111. ACM Press, 1996.

[363] J. E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley, 1998.

[364] T. Schlipf and T. Buechner and R. Fritz and M. Helms and J. Koehl. Formal verification made easy. *IBM Journal of Research and Development*, **41**(4–5):567–576, 1997.

[365] K. Schneider. *Verification of Reactive Systems: Formal Methods and Algorithms*. Springer-Verlag, 2004.

[366] S. Schneider. *Specifying Real-Time Systems in Timed CSP*. Prentice-Hall, 2000.

[367] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Springer, 2003.

[368] S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. In *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *Lecture Notes in Computer Science*, pages 174–190. Springer-Verlag, 2005.

[369] R. Sebastiani and S. Tonetta. "More deterministic" vs. "smaller" Büchi automata for efficient LTL model checking. In *12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 2860 of *Lecture Notes in Computer Science*, pages 126–140. Springer-Verlag, 2003.

[370] R. Segala and N. Lynch. Probabilistic simulations for probabilistic processes. *Nordic Journal of Computing*, **2**(2):250–273, 1995.

[371] A. P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, **6**(5):495–512, 1994.

[372] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logic. *Journal of the ACM*, **32**(3):733–749, 1985.

[373] A. P. Sistla and M. Y. Vardi and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, **49**:217–237, 1987.

[374] F. Somenzi. Binary decision diagrams. In M. Broy and R. Steinbruggen, editors, *Calculational System Design*, volume 173 of *NATO Science Series F: Computer and Systems Sciences*, pages 303–366. IOS Press, 1999.

[375] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *12th International Conference on Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 248–263. Springer-Verlag, 2000.

[376] L. Staiger. Research in the theory of omega-languages. *Elektronische Informationsverarbeitung und Kybernetik*, **23**(8–9):415–439, 1987.

[377] J. Staunstrup and H. R. Andersen and H. Hulgaard and J. Lind-Nielsen and K. G. Larsen and G. Behrmann and K. Kristoffersen and A. Skou and H. Leerberg and N. B. Theilgaard. Practical verification of embedded software. *IEEE Computer*, **33**(5):68–75, 2000.

[378] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.

[379] C. Stirling. *Modal and Temporal Properties of Processes*. Texts in Computer Science. Springer-Verlag, New York, 2001.

[380] F. A. Stomp and W.-P. de Roever. A principle for sequential reasoning about distributed algorithms. *Formal Aspects of Computing*, **6**(6):716–737, 1994.

[381] N. Storey. *Safety-Critical Computer Systems*. Addison-Wesley, 1996.

[382] R. S. Streett. Propositional dynamic logic of looping and converse is elementarily decidable. *Information and Control*, **54**(1–2):121–141, 1982.

[383] T. A. Sudkamp. *Languages and Machines, 3rd edition*. Addison-Wesley, 2005.

[384] B.K. Szymanski. A simple solution to Lamport's concurrent programming problem with linear wait. In *International Conference on Supercomputing Systems*, pages 621–626, 1988.

[385] L. Tan and R. Cleaveland. Simulation revisited. In *7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, pages 480–495. Springer-Verlag, 2001.

[386] S. Tani and K. Hamaguchi and S. Yajima. The complexity of the optimal variable ordering problems of shared binary decision diagrams. In *4th International Symposium on Algorithms and Computation*, volume 762 of *Lecture Notes in Computer Science*, pages 389–398. Springer-Verlag, 1993.

[387] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, **1**(2):146–160, 1972.

[388] H. Tauriainen. Nested emptiness search for generalized Büchi automata. Research Report A79, Helsinki University of Technology, Laboratory for Theoretical Computer Science, 2003.

[389] X. Thirioux. Simple and efficient translation from LTL formulas to Büchi automata. *Electronic Notes in Theoretical Computer Science*, **66**(2), 2002.

[390] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 4, pages 133–191. Elsevier Publishers B.V., 1990.

[391] W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 389–455. Springer-Verlag, 1997.

[392] B. A. Trakhtenbrot. Finite automata and the logic of one-place predicates. *Siberian Mathematical Journal*, **3**:103–131, 1962.

[393] G. J. Tretmans and K. Wijbrans and M. Chaudron. Software engineering with formal methods: the development of a storm surge barrier control system. *Formal Methods in System Design*, **19**(2):195–215, 2001.

[394] S. Tripakis and S. Yovine. Analysis of timed systems based on time-abstracting bisimulations. In *8th International Conference on Computer Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 232–243. Springer-Verlag, 1996.

[395] S. Tripakis and S. Yovine. Analysis of timed systems using time-abstracting bisimulations. *Formal Methods in System Design*, **18**(1):25–68, 2001.

[396] R. Trudeau. *Introduction to Graph Theory*. Dover Publications Inc., 1994.

[397] D. Turi and J. J. M. M. Rutten. On the foundations of final coalgebra semantics. *Mathematical Structures in Computer Science*, **8**(5):481–540, 1998.

[398] A. Valmari. Stubborn sets for reduced state space generation. In *10th International Conference on Applications and Theory of Petri Nets (ICATPN)*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer-Verlag, 1989.

[399] A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, **1**(4):297–322, 1992.

[400] A. Valmari. On-the-fly verification with stubborn sets. In *5th International Conference on Computer Aided Verification (CAV)*, volume 697 of *Lecture Notes in Computer Science*, pages 397–408. Springer-Verlag, 1993.

[401] A. Valmari. Stubborn set methods for process algebras. In *Partial Order Methods in Verification* [328], pages 213–231.

[402] H. van der Schoot and H. Ural. An improvement of partial order verification. *Software Testing, Verification and Reliability*, **8**(2):83–102, 1998.

[403] J.L.A. van der Snepscheut. *Trace Theory and VLSI Design*, volume 200 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[404] R. J. van Glabbeek. The linear time – branching time spectrum (extended abstract). In *1st International Conference on Concurrency Theory (CONCUR)*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer-Verlag, 1990.

[405] R. J. van Glabbeek. The linear time – branching time spectrum II. In *4th International Conference on Concurrency Theory (CONCUR)*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer-Verlag, 1993.

[406] R. J. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, **43**(3):555–600, 1996.

[407] M. Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *26th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 327–338. IEEE Computer Society Press, 1985.

[408] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *8th Banff Higher Order Workshop Conference on Logics for Concurrency: Structure versus Automata*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer-Verlag, 1996.

[409] M. Y. Vardi. Probabilistic linear-time model checking: An overview of the automata-theoretic approach. In *5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems (ARTS)*, volume 1601, pages 265–276. Springer-Verlag, 1999.

[410] M. Y. Vardi. Branching versus linear time: Final showdown. In *7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22. Springer-Verlag, 2001.

[411] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *1st Annual Symposium on Logic in Computer Science (LICS)*, pages 332–344. IEEE Computer Society Press, 1986.

[412] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, **115**(1):1–37, 1994.

[413] K. Varpaaniemi. On stubborn sets in the verification of linear time temporal properties. In *19th International Conference on Application and Theory of Petri Nets (ICATPN)*, volume 1420 of *Lecture Notes in Computer Science*, pages 124–143. Springer-Verlag, 1998.

[414] W. Visser and H. Barringer. Practical CTL* model checking: should SPIN be extended? *International Journal on Software Tools for Technology Transfer*, **2**(4):350–365, 2000.

[415] H. Völzer and D. Varacca and E. Kindler. Defining fairness. In *16th International Conference on Concurrency Theory (CONCUR)*, volume 3653 of *Lecture Notes in Computer Science*, pages 458–472. Springer-Verlag, 2005.

[416] F. Wallner. Model checking LTL using net unfoldings. In *10th International Conference on Computer Aided Verification (CAV)*, volume 1427 of *Lecture Notes in Computer Science*, pages 207–218. Springer-Verlag, 1998.

[417] F. Wang. Efficient verification of timed automata with BDD-like data structures. *Journal on Software Tools and Technology Transfer*, **6**(1):77–97, 2004.

[418] I. Wegener. *Branching Programs and Binary Decision Diagrams: Theory and Applications*. SIAM Monographs on Discrete Mathematics and Applications. Society for Industrial and Applied Mathematics, 2000.

[419] C. H. West. An automated technique for communications protocol validation. *IEEE Transactions on Communications*, **26**(8):1271–1275, 1978.

[420] C. H. West. Protocol validation in complex systems. In *Symposium on Communications Architectures and Protocols*, pages 303–312. ACM Press, 1989.

[421] J. A. Whittaker. What is software testing? Why is it so hard? *IEEE Software*, **17**(1):70–79, 2000.

[422] B. Willems and P. Wolper. Partial-order methods for model checking: from linear time to branching time. In *11th IEEE Symposium on Logic in Computer Science (LICS)*, page 294. IEEE Computer Society Press, 1996.

[423] G. Winskel. Event structures. In *Petri Nets: Central Models and Their Properties, Advances in Petri Nets*, volume 255 of *Lecture Notes in Computer Science*, pages 325–392. Springer-Verlag, 1986.

[424] P. Wolper. Specification and synthesis of communicating processes using an extended temporal logic. In *9th Symposium on Principles of Programming Languages (POPL)*, pages 20–33. ACM Press, 1982.

[425] P. Wolper. Temporal logic can be more expressive. *Information and Control*, **56**(1–2):72–99, 1983.

[426] P. Wolper. An introduction to model checking. Position statement for panel discussion at the Software Quality workshop, 1995.

[427] W. Yi. CCS + time = an interleaving model for real-time systems. In *18th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 510 of *Lecture Notes in Computer Science*, pages 217–228. Springer-Verlag, 1991.

[428] M. Yoeli. *Formal Verification of Hardware Design*. IEEE Computer Society Press, 1990.

[429] S. Yovine. KRONOS: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, **1**(1-2):123–133, 1997.

[430] S. Yovine. Model checking timed automata. In G. Rozenberg and F. Vaandrager, editors, *Lectures on Embedded Systems*, volume 1494 of *Lecture Notes in Computer Science*, pages 114–152. Springer-Verlag, 1998.

Index

F

G

V

W

Z