Chapter 1

Introduction to Trace Theory

Antoni Mazurkiewicz

Institute of Computer Science, Polish Academy of Sciences
ul. Ordona 21, 01-237 Warszawa, and
Institute of Informatics, Jagiellonian University
ul. Nawojki 11, 31-072 Kraków, Poland
amaz@wars.ipipan.waw.pl

Contents

1.1 Introduction

The theory of traces has been motivated by the theory of *Petri Nets* and the theory of *formal languages and automata.*

Already in 1960's Carl Adam Petri has developed the foundations for the theory of concurrent systems. In his seminal work [226] he has presented a model which is based on the communication of interconnected sequential systems. He has also presented an entirely new set of notions and problems that arise when dealing with concurrent systems. His model (or actually a family of "net-based" models) that has been since generically referred to as *Petri Nets*, has provided both an intuitive informal framework for representing basic situations of concurrent systems, and a formal framework for the mathematical analysis of concurrent systems. The intuitive informal framework has been based on a very convenient graphical representation of net-based systems, while the formal framework has been based on the token game resulting from this graphical representation.

The notion of a *finite automaton*, seen as a restricted type of Turing Machine, has by then become a classical model of a sequential system. The strength of automata theory is based on the "simple elegance" of the underlying model which admits powerful mathematical tools in the investigation of both the structural properties expressed by the underlying graph-like model, and the behavioural properties based on the notion of the *language* of a system. The notion of language has provided a valuable link with the theory of free monoids.

The original attempt of the theory of traces [189] was to use the well developed tools of formal language theory for the analysis of concurrent systems where the notion of concurrent system is understood in very much the same way as it is done in the theory of Petri Nets. The idea was that in this way one will get a framework for reasoning about concurrent systems which, for a number of important problems, would be mathematically more convenient that some approaches based on formalizations of the token game.

In 1970's when the basic theory of traces has been formulated, the most popular approach to deal with concurrency was *interleaving*. In this approach concurrency is replaced by *non-determinism* where concurrent execution of actions is treated as non-deterministic choice of the order of executions of those actions. Although the interleaving approach is quite adequate for many problems, it has a number of serious pitfalls. We will briefly discuss here some of them, because those were important drawbacks that we wanted to avoid in trace theory.

By reducing concurrency to non-determinism one assigns two different meanings to the term "non-deterministic choice": the choice between two (or more) possible actions that exclude each other, and the lack of information about the order of two (or more) actions that are executed independently of each other. Since in the theory of concurrent systems we are interested not only in the question "*what* is computed?" but also in the question "*how* it is computed?", this identification may be very misleading.

This disadvantage of the interleaving approach is well-visible in the treatment of *refinement*, see e.g. [49]. It is widely recognized that refinement is one of the basic transformations of any calculus of concurrent systems from theoretical, methodological, and practical point of view. This transformation must preserve the basic relationship between a system and its behaviour: the behaviour of the refined system is the refined behaviour of the original system. This requirement is not respected if the behaviour of the system is represented by interleaving.

Also in considerations concerning *inevitability*, see [192], the interleaving approach leads to serious problems. In non-deterministic systems where a choice between two partners may be repeated infinite number of times, a run discriminating against one of the partners is possible; an action of an a priori chosen partner is not inevitable. On the other hand, if the two partners repeat their actions *independently* of each other, an action of any of them is inevitable. However, in the interleaving approach one does not distinguish between these two situations: both are described in the same way. Then, as a remedy against this confusion, a special notion of fairness has to be introduced, where in fact this notion is outside of the usual algebraic means for the description of system behaviour.

Finally, the identification of non-deterministic choice and concurrency becomes a real drawback when considering *serializability of transactions* [104]. To keep consistency of a database to which a number of users have concurrent access, the database manager has to allow concurrent execution of those transactions that do not interfere with each other. To this end it is necessary to distinguish transactions that are in conflict from those which are independent of each other; the identification of the non-deterministic choice (in the case of conflicting transactions) with the choice of the execution order (in case of independent transactions) leads to serious difficulties in the design of database managing systems.

Above we have sketched some of the original motivations that led to the formulation of the theory of traces in 1977. Since then this theory has been developed both in breadth and in depth, and this volume presents the state of the art of the theory of traces. Some of the developments have followed the initial motivation coming from concurrent systems, while other fall within the areas such as formal language theory, theory of partially commutative monoids, graph grammars, combinatorics of words, etc.

In our paper we discuss a number of notions and results that have played a crucial role in the initial development of the theory of traces, and which in our opinion are still quite relevant.

1.2 Preliminary Notions

Let X be a set and R be a binary relation in X; R is an ordering relation in X, or X is ordered by R, if R is reflexive (xRx for all $x \in X$), transitive ($xRyRz$ implies xRz), and antisymmetric (xRy and yRx implies $x = y$); if, moreover, R is connected (for any $x, y \in X$ either xRy or yRx), then R is said to be a *linear*, or *total* ordering. A set, together with an ordering relation, is an *ordered set*. Let X be a set ordered by relation R. Any subset Y of X is then ordered by the restriction of R to Y, i.e. by $R \cap (Y \times Y)$.

The set $\{0, 1, 2, \ldots, n, \ldots\}$ will be denoted by ω. By an *alphabet* we shall understand a finite set of *symbols* (*letters*); alphabets will be denoted by Greek capital letters, e.g. $\Sigma, \Delta, \ldots$, etc. Let Σ be an alphabet; the set of all finite sequences of elements of Σ will be denoted by Σ^*; such sequences will be referred to as *strings* over Σ. Small initial letters $a, b, \ldots$, with possible sub- or superscripts will denote strings, small final Latin letters: $w, u, v, \ldots$, etc. will denote strings. If $u = (a_1, a_2, \ldots, a_n)$ is a string, n is called the *length* of u and is denoted by $|u|$. For any strings $(a_1, a_2, \ldots, a_n), (b_1, b_2, \ldots, b_m)$, the concatenation $(a_1, a_2, \ldots, a_n) \circ (b_1, b_2, \ldots, b_m)$

is the string $(a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m)$. Usually, sequences $(a_1, a_2, \ldots, a_n)$ are written as $a_1 a_2 \ldots a_n$. Consequently, a will denote symbol a as well as the string consisting of single symbol a; the proper meaning will be always understood by a context. String of length 0 (containing no symbols) is denoted by ϵ. The set of strings over an alphabet together with the concatenation operation and the empty string as the neutral element will be referred to as the *monoid* of strings over Σ, or the *free monoid* generated by Σ. Symbol Σ^* is used to denote the monoid of strings over Σ as well the set Σ^* itself.

We say that symbol a *occurs* in string w, if $w = w_1 a w_2$ for some strings w_1, w_2. For each string w define $\mathrm{Alph}(w)$ (the *alphabet* of w) as the set of all symbols occurring in w. By $w(a)$ we shall understand the number of occurrences of symbol a in string w.

Let Σ be an alphabet and w be a string (over an arbitrary alphabet); then $\pi_\Sigma(w)$ denotes the (string) *projection* of w onto Σ defined as follows:

$$\pi_\Sigma(w) = \begin{cases} \epsilon, & \text{if } w = \epsilon, \\ \pi_\Sigma(u), & \text{if } w = ua, a \notin \Sigma, \\ \pi_\Sigma(u)a, & \text{if } w = ua, a \in \Sigma. \end{cases}$$

Roughly speaking, projection onto Σ deletes from strings all symbols not in Σ. The subscript in π_Σ is omitted if Σ is understood by a context.

The *right cancellation* of symbol a in string w is the string $w \div a$ defined as follows:

$$\epsilon \div a = \epsilon, \tag{1.1}$$

$$(wb) \div a = \begin{cases} w, & \text{if } a = b, \\ (w \div a)b, & \text{otherwise.} \end{cases} \tag{1.2}$$

for all strings w and symbols a, b. It is easy to prove that projection and cancellation commute:

$$\pi_\Sigma(w) \div a = \pi_\Sigma(w \div a), \tag{1.3}$$

for all strings w and symbols a.

In our approach a *language* is defined by an alphabet Σ and a set of strings over Σ (i.e. a subset of Σ^*). Frequently, a language will be identified with its set of strings; however, in contrast to many other approaches, two languages with the same set of strings but with different alphabets will be considered as different; in particular, two singleton languages containing only empty strings, but over different alphabets, are considered as different. Languages will be denoted by initial Latin capital letters, e.g. $A, B, \ldots$, with possible subscripts and superscripts.

If A, B are languages over a common alphabet, then their concatenation AB is the language $\{uv \mid u \in A, v \in B\}$ over the same alphabet. If w is a string, A is a language, then (omitting braces around singletons)

$$wA = \{wu \mid u \in A\}, Aw = \{uw \mid u \in A\}.$$

The power of a language A is defined recursively:

$$A^0 = \{\epsilon\}, A^{n+1} = A^n A,$$

for each $n = 0, 1, \ldots$, and the iteration A^* of A is the union:

$$\bigcup_{n=0}^{\infty} A^n.$$

Extend the projection function from strings to languages defining for each language A and any alphabet Σ the projection of A onto Σ as the language $\pi_\Sigma(A)$ defined as

$$\pi_\Sigma(A) = \{\pi_\Sigma(u) \mid u \in A\}.$$

For each string w elements of the set

$$\text{Pref}(w) = \{u \mid \exists v : uv = w\}$$

are called *prefixes* of w. Obviously, Pref(w) contains ϵ and w. For any language A over Σ define

$$\text{Pref}(A) = \bigcup_{w \in A} \text{Pref}(w).$$

Elements of Pref(A) are called prefixes of A. Obviously, $A \subseteq \text{Pref}(A)$. Language A is *prefix closed*, if $A = \text{Pref}(A)$. Clearly, for any language A the language Pref(A) is prefix closed. The *prefix relation* is a binary relation $\sqsubseteq$ in Σ^* such that $u \sqsubseteq w$ if and only $u \in \text{Pref}(w)$. It is clear that the prefix relation is an ordering relation in any set of strings.

1.3 Dependency and Traces

By a *dependency* we shall mean any finite, reflexive and symmetric relation, i.e. a finite set of ordered pairs D such that if (a, b) is in D, then (b, a) and (a, a) are in D. Let D be a dependency; the the domain of D will be denoted by Σ_D and called the *alphabet* of D. Given dependency D, call the relation $I_D = (\Sigma_D \times \Sigma_D) - D$ *independency* induced by D. Clearly, independency is a symmetric and irreflexive relation. In particular, the empty relation, identity relation in Σ and the full relation in Σ (the relation $\Sigma \times \Sigma$) are dependencies; the first has empty alphabet, the second is the least dependency in Σ, the third is the greatest dependency in Σ. Clearly, the union and intersection of a finite number of dependencies is a dependency. It is also clear that each dependency is the union of a finite number of full dependencies (since any symmetric and reflexive relation is the union of its cliques).

Example 1.3.1 The relation $D = \{a, b\}^2 \cup \{a, c\}^2$ is a dependency; $\Sigma_D = \{a, b, c\}$, $I_D = \{(b, c), (c, b)\}$.

In this section we choose dependency as a primary notion of the trace theory; however, for other purposes it may be more convenient to take as a basic notion *concurrent alphabet*, i.e any pair (Σ, D) where Σ is an alphabet and D is a dependency, or any pair (Σ, I), where Σ is an alphabet and I is an independency, or *reliance alphabet*, i.e. any triple (Σ, D, I), where Σ is an alphabet, D is a dependency and I is independency induced by D.

Let D be a dependency; define the *trace equivalence* for D as the least congruence $\equiv_D$ in the monoid Σ_D^* such that for all a, b

$$(a,b) \in I_D \Rightarrow ab \equiv_D ba. \tag{1.4}$$

Equivalence classes of $\equiv_D$ are called *traces* over D; the trace represented by string w is denoted by $[w]_D$. By $[\Sigma^*]_D$ we shall denote the set $\{[w]_D \mid w \in \Sigma_D^*\}$, and by $[\Sigma]_D$ the set $\{[a]_D \mid a \in \Sigma_D\}$.

Example 1.3.2 For dependency $D = \{a,b\}^2 \cup \{a,c\}^2$ the trace over D represented by string $abbca$ is $[abbca]_D = \{abbca, abcba, acbba\}$.

By definition, a single trace arises by identifying all strings which differ only in the ordering of adjacent independent symbols. The quotient monoid $\mathbb{M}(D) = \Sigma_D^*/_{\equiv_D}$ is called the *trace monoid* over D and its elements the *traces* over D. Clearly, $\mathbb{M}(D)$ is generated by $[\Sigma]_D$. In the monoid $\mathbb{M}(D)$ some symbols from Σ commute (in contrast to the monoid of strings); for that reason $\mathbb{M}(D)$ is also called a *free partially commutative monoid* over D. As in the case of the monoid of strings, we use the symbol $\mathbb{M}(D)$ to denote the monoid itself as well as the set of all traces over D. It is clear that in case of full dependency, i.e. if D is a single clique, traces reduce to strings and $\mathbb{M}(D)$ is isomorphic with the free monoid of strings over Σ_D. We are going to develop the algebra of traces along the same lines as it has been done in case of of strings. Let us recall that the mapping $\varphi_D : \Sigma^* \longrightarrow [\Sigma^*]_D$ such that

$$\varphi_D(w) = [w]_D$$

is a homomorphism of Σ^* onto $\mathbb{M}(D)$, called the *natural homomorphism* generated by the equivalence $\equiv_D$.

Now, we shall give some simple facts about the trace equivalence and traces. Let D be fixed from now on; subscripts D will be omitted if it causes no ambiguity, I will be the independency induced by D, Σ will be the domain of D, all symbols will be symbols in Σ_D, all strings will be strings over Σ_D, and all traces will be traces over D, unless explicitly stated otherwise.

It is clear that $u \equiv v$ implies $\text{Alph}(u) = \text{Alph}(v)$; thus, for all strings w, we can define $\text{Alph}([w])$ as $\text{Alph}(w)$. Denote by $\sim$ a binary relation in Σ^* such that $u \sim v$ if and only if there are $x, y \in \Sigma^*$, and $(a,b) \in I$ such that $u = xaby, v = xbay$; it is not difficult to prove that $\equiv$ is the symmetric, reflexive, and transitive closure of $\sim$. In other words, $u \equiv v$ if and only if there exists a sequence $(w_0, w_1, \ldots, w_n), n \geq 0$, such that $w_0 = u, w_n = v$, and for each $i, 0 < i \leq n, w_{i-1} \sim w_i$.

Permutation is the least congruence $\simeq$ in Σ^* such that for all symbols a, b

$$ab \simeq ba. \tag{1.5}$$

If $u \simeq v$, we say that u is a permutation of v. Comparing (1.4) with (1.5) we see at once that $u \equiv v$ implies $u \simeq v$.

Define the *mirror image* w^R of string w as follows:

$$\epsilon^R = \epsilon, \quad (ua)^R = a(u^R),$$

for any string u and symbol a. It is clear that the following implication (the mirror rule) holds:

$$u \equiv v \Rightarrow u^R \equiv v^R; \tag{1.6}$$

thus, we can define $[w]^R$ as equal to $[w^R]$.

Since obviously $u \sim v$ implies $(u \div a) \sim (v \div a)$, and $\equiv$ is the transitive and reflexive closure of $\sim$, we have the following property (the cancellation property) of traces:

$$u \equiv v \Rightarrow (u \div a) \equiv (v \div a); \tag{1.7}$$

thus, we can define $[w] \div a$ as $[w \div a]$, for all strings w and symbols a.

We have also the following *projection rule*:

$$u \equiv_v \Rightarrow \pi_\Sigma(u) \equiv \pi_\Sigma(v) \tag{1.8}$$

for any alphabet Σ, being a consequence of the obvious implication $u \sim_v \Rightarrow \pi_\Sigma(u) \equiv \pi_\Sigma(v)$.

Let u, v be strings, a, b be symbols, $a \neq b$. If $ua \equiv vb$, then applying twice rule (1.7) we get $u \div b \equiv v \div a$; denoting $u \div b$ by w, by the cancellation rule again we get $u = (ua) \div a \equiv (vb) \div a = (v \div a)b \equiv wb$. Similarly, we get $v \equiv wa$. Since $ua \equiv vb, wba \equiv wab$ and by the definition of traces $ab \equiv ba$. Thus, we have the following implication for all strings u, v and symbols a, b:

$$ua \equiv vb \wedge a \neq b \Rightarrow (a,b) \in I \wedge \ \exists w : u \equiv wb \wedge v \equiv wa. \tag{1.9}$$

Obviously, $u \equiv v \Rightarrow xuy \equiv xvy$. If $xuy \equiv xvy$, by the cancellation rule $xu \equiv xv$; by the mirror rule $u^R x^R \equiv v^R x^R$; again by the cancellation rule $u^R \equiv v^R$, and again by the mirror rule $u \equiv v$. Hence, from the mirror property and the cancellation property it follows the following implication:

$$xuy \equiv xvy \Rightarrow u \equiv v. \tag{1.10}$$

Extend independency relation from symbols to strings, defining strings u, v to be independent, $(u, v) \in I$, if $\text{Alph}(u) \times \text{Alph}(v) \subseteq I$. Notice that the empty string is independent of any other: $(\epsilon, w) \in I$ trivially holds for any string w.

Proposition 1.3.3 *For all strings w, u, v and symbol $a \notin \text{Alph}(v)$: $uav \equiv wa \Rightarrow (a, v) \in I$.*

Proof: By induction. If $v = \epsilon$ the implication is clear. Otherwise, there is symbol $b \neq a$ and string x such that $v = xb$. By Proposition 1.9 $(a, b) \in I$ and $uaxb \equiv wa$. By cancellation rule, since $b \neq a$, we get $uax \equiv (w \div b)a$. By induction hypothesis $(a, x) \in I$. Since $(a, b) \in I$, $(a, xb) \in I$ which proves the proposition. □

The next theorem is a trace generalization of the Levi Lemma for strings. which reads: for all strings u, v, x, y, if $uv = xy$, then there exists a string w such that either $uw = x, wy = v$, or $xw = u, wv = y$. In case of traces this useful lemma admits more symmetrical form:

Theorem 1.3.4 (Levi Lemma for traces) *For any strings u, v, x, y such that $uv \equiv xy$ there exist strings z_1, z_2, z_3, z_4 such that $(z_2, z_3) \in I$ and*

$$u \equiv z_1z_2, v \equiv z_3z_4, x \equiv z_1z_3, y \equiv z_2z_4. \tag{1.11}$$

Proof: Let u, v, x, y be strings such that $uv \equiv xy$. If $y = \epsilon$, the proposition is proved by setting $z_1 = u, z_2 = z_4 = \epsilon, z_3 = v$. Let w be a string and e be a symbol such that $y = we$. Then there can be two cases: either (1) there are strings v', v'' such that $v = v'ev''$ and $e \notin \text{Alph}(v'')$, or (2) there are strings u', u'' such that $u = u'eu''$, and $e \notin \text{Alph}(u''v)$.

In case (1) by the cancellation rule we have $uv'v'' \equiv xw$; by induction hypothesis there are z_1', z_2', z_3', z_4' such that

$$uv' \equiv z_1'z_2', v'' \equiv z_3'z_4', x \equiv z_1'z_3', w \equiv z_2'z_4',$$

and $(z_2', z_3') \in I$. Set $z_1 = z_1', z_2 = z_2'e, z_3 = z_3', z_4 = z_4'$. By Proposition 1.3.3 $(e, v'') \in I$, hence $(e, z_3') \in I$ and $(e, z_4') \in I$. Since $(e, z_4') \in I$, $ez_4' \equiv z_4'e$; and it is easy to check that equivalences (1.11) are satisfied. We have also $(z_2', z_3') \in I$ and $(e, z_3') \in I$; it yields $(z_2'e, z_3') \in I$ which proves $(z_2, z_3) \in I$.

In case(2) by cancellation rule we have $u'u''v \equiv xw$ and by induction hypothesis there are z_1', z_2', z_3', z_4' such that

$$u' \equiv z_1'z_2', u''v \equiv z_3'z_4', x \equiv z_1'z_3', w \equiv z_2'z_4'.$$

and $(z_2', z_3') \in I$. Set, as above, $z_1 = z_1', z_2 = z_2'e, z_3 = z_3', z_4 = z_4'$. By Proposition 1.3.3 $(e, u''v) \in I$, hence $(e, z_3') inI$ and $(e, z_4') \in I$. Since $ue \equiv z_2'z_4'e \equiv z_2'ez_4' = z_2z_4)$, equivalences (1.11) are satisfied. By induction hypothesis $(z_2', z_3') \in I$; together with $(e, z_3') inI$ it implies $(z_2, z_3) \in I$. □

Let $[u], [v]$ be traces over the same dependency. Trace $[u]$ is a *prefix* of trace $[v]$ (and $[v]$ is a *dominant* of $[u]$), if there exists trace $[x]$ such that $[ux] = [v]$. Similarly to the case of strings, the relation "to be a prefix" in the set of all traces over a fixed dependency is an ordering relation; however, in contrast to the prefix relation in the set of strings, the prefix relation for traces orders sets of traces (in general) only partially.

The structure of prefixes of the trace $[abbca]$ for dependency $\{a, b\}^2 \cup \{a, c\}^2$ is given in the Figure 1.2.

Proposition 1.3.5 *Let $[w]$ be a trace and $[u], [v]$ be prefixes of $[w]$. Then there exist the greatest common prefix and the least common dominant of $[u]$ and $[v]$.*

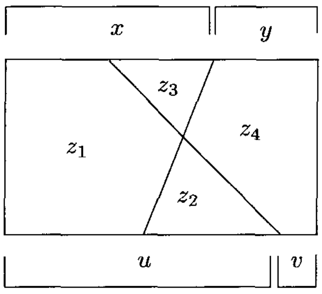

Figure 1.1: Graphical interpretation of Levi Lemma for traces.

$$\begin{tikzcd}
 & & {[\epsilon]} \arrow[d] & \\
 & & {[a]} \arrow[dl] \arrow[dr] & \\
 & {[ab]} \arrow[dl] \arrow[dr] & & {[ac]} \arrow[dl] \\
{[abb]} \arrow[dr] & & {[abc]} \arrow[dl] & \\
 & {[abbc]} \arrow[d] & & \\
 & {[abbca]} & &
\end{tikzcd}$$

Figure 1.2: Structure of prefixes of $[abbca]_D$ for $D = \{a, b\}^2 \cup \{a, c\}^2$.

Proof: Since $[u], [v]$ are prefixes of $[w]$, there are traces $[x], [y]$ such that $ux \equiv vy$. Then, by Levi Lemma for traces, there are strings z_1, z_2, z_3, z_4 such that $u \equiv z_1 z_2, x \equiv z_3 z_4, v \equiv z_1 z_3, y \equiv z_2 z_4$, and z_2, z_3 are independent. Then $[z_1]$ is the greatest common prefix and $[z_1 z_2 z_3]$ is the least common dominant of $[u], [v]$. Indeed, $[z_1]$ is a common prefix of both $[u], [v]$; it is the greatest common prefix, since any symbol in z_2 does not occur in z_3 and any symbol in z_3 does not occur in z_2; hence any extension of $[z_1]$ is not a prefix either of $[z_1 z_2] = [u]$, or of $[z_1 z_3] = [v]$. Thus, any trace being a prefix of $[u]$ and of $[v]$ must be a prefix of $[z_1]$. Similarly, $[z_1 z_2 z_3]$ is a common dominant of $[u], [v]$; but any proper prefix of $[z_1 z_2 z_3]$ is either not dominating $[u]$, if it does not contain a symbol from z_2, or not dominating $[v]$, if it does not contain a symbol from z_3. □

Let us close this section with a characterization of trace monoids by homomorphisms (so-called dependency morphisms).

A *dependency* morphism w.r.t. D is any homomorphism ϕ from the monoid of strings over Σ_D onto another monoid such that

$$\begin{array}{ll}
\text{A1}: & \phi(w) = \phi(\epsilon) \Rightarrow w = \epsilon, \\
\text{A2}: & (a,b) \in I \Rightarrow \phi(ab) = \phi(ba), \\
\text{A3}: & \phi(ua) = \phi(v) \Rightarrow \phi(u) = \phi(v \div a), \\
\text{A4}: & \phi(ua) = \phi(vb) \wedge a \neq b \Rightarrow (a,b) \in I.
\end{array}$$

Lemma 1.3.6 *Let ϕ be a dependency morphism, $u, v \in \Sigma^*, a, b \in \Sigma$. If $\phi(ua) = \phi(vb)$ and $a \neq b$, then there exists $w \in \Sigma^*$ such that $\phi(u) = \phi(wb)$ and $\phi(v) = \phi(wa)$.*

Proof: Let u, v, a, b be such as assumed in the above Lemma and let

$$\phi(ua) = \phi(vb) \tag{1.12}$$

Applying twice A3 to (1.12) we get

$$\phi(u \div b) = \phi(v \div a) \tag{1.13}$$

Set $w = u \div b$. Then

$$\begin{array}{rcl}
\phi(wa) & = & \phi((u \div b)a) \\
 & = & \phi((ua) \div b), \text{ since } a \neq b, \\
 & = & \phi(v), \text{ from (1.12) by A3,}
\end{array}$$

and

$$\begin{array}{rcl}
\phi(wb) & = & \phi((u \div b)b) \\
 & = & \phi((v \div a)b), \text{ from (1.13),} \\
 & = & \phi((vb) \div a), \text{ since } a \neq b, \\
 & = & \phi(u), \text{ from (1.12) by A3.}
\end{array}$$

□

Lemma 1.3.7 *Let ϕ, ψ be dependency morphisms w.r.t. the same dependency. Then $\phi(x) = \phi(y) \Rightarrow \psi(x) = \psi(y)$ for all x, y.*

Proof: Let D be a dependency and let ϕ, ψ be two concurrency mappings w.r.t. dependency D, $x, y \in \Sigma^*$, and $\phi(x) = \phi(y)$. If $x = \epsilon$, then by A1 $y = \epsilon$ and clearly $\psi(x) = \psi(y)$. If $x \neq \epsilon$, then $y \neq \epsilon$. Thus, $x = ua, y = vb$ for some $u, v \in \Sigma^*, a, b \in \Sigma$ and we have

$$\phi(ua) = \phi(vb). \tag{1.14}$$

There can be two cases. In the first case we have $a = b$, then by A3 $\phi(u) = \phi(v)$ and by induction hypothesis $\psi(u) = \psi(v)$; Thus $\psi(ua) = \psi(vb)$, which proves $\psi(x) = \psi(y)$. In the second case, $a \neq b$. By A4 $(a, b) \in I$. By Lemma 1.3.6 we get $\phi(u) = \phi(wb), \phi(v) = \phi(wa)$, for some $w \in \Sigma^*$. By induction hypothesis

$$\psi(u) = \psi(wb), \psi(v) = \psi(wa) \tag{1.15}$$

hence

$$\psi(ua) = \psi(wba), \psi(vb) = \psi(wab). \tag{1.16}$$

By A3, since $(a, b) \in I$,

$$\psi(ba) = \psi(ab), \tag{1.17}$$

hence $\psi(wba) = \psi(wab)$, which proves $\psi(x) = \psi(y)$. □

Theorem 1.3.8 *If ϕ, ψ are dependency morphisms w.r.t. the same dependency onto monoids M, N, then M is isomorphic with N.*

Proof: By Lemma 1.3.7 the mapping defined by the equality

$$\theta(\phi(w)) = \psi(w),$$

for all $w \in \Sigma^*$ is a homomorphism from M onto N. Since θ has its inverse, namely

$$\theta^{-1}(\psi(w)) = \phi(w),$$

which is a homomorphism also, θ is a homomorphism from N onto M, θ is an isomorphism. □

Theorem 1.3.9 *The natural homomorphism of the monoid of strings over Σ_D onto the monoid of traces over D is a dependency morphism.*

Proof: We have to check conditions A1 – A4 for $\phi(w) = [w]_D$. Condition A1 is obviously satisfied; condition A2 follows directly from the definition of trace monoids. Condition A3 follows from the cancellation property (1.7); condition A4 follows from (1.9). □

Corollary. If ϕ is a dependency morphism w.r.t. D onto M, then M is isomorphic with the monoid of traces over D and the isomorphism is induced by the mapping θ such that $\theta([a]_D) = \phi_D(a)$ for all $a \in \Sigma_D$.

Proof: It follows from Theorem 1.3.8 and Theorem 1.3.9. □

1.4 Dependence Graphs

Dependence graphs are thought as graphical representations of traces which make explicit the ordering of symbol occurrences within traces. It turns out that for a given dependency the algebra of traces is isomorphic with the algebra of dependency graphs, as defined below. Therefore, it is only a matter of taste which objects are chosen for representing concurrent processes: equivalence classes of strings or labelled graphs.

Let D be a dependency relation. Dependency graphs over D (or d-graphs for short) are finite, oriented, acyclic graphs with nodes labelled with symbols from Σ_D in such a way that two nodes of a d-graph are connected with an arc if and only if they are different and labelled with dependent symbols. Formally, a triple

$$\gamma = (V, R, \varphi)$$

is a dependence graph (d-graph) over D, if

$$V \text{ is a finite set (of nodes of } \gamma\text{)}, \tag{1.18}$$
$$R \subseteq V \times V \text{ (the set of arcs of } \gamma\text{)}, \tag{1.19}$$
$$\varphi : V \longrightarrow \Sigma_D \text{ (the labelling of } \gamma\text{)}, \tag{1.20}$$

such that

$$R^+ \cap id_V = \emptyset, \quad \text{(acyclicity)} \tag{1.21}$$
$$(v_1, v_2) \in R \cup R^{-1} \cup id_V \Leftrightarrow (\varphi(v_1), \varphi(v_2)) \in D \quad \text{(D-connectivity)} \tag{1.22}$$

Two d-graphs γ', γ'' are isomorphic, $\gamma' \simeq \gamma''$, if there exists a bijection between their nodes preserving labelling and arc connections. As usual, two isomorphic graphs are identified; all subsequent properties of d-graphs are formulated up to isomorphism. The empty d-graph $(\emptyset, \emptyset, \emptyset)$ will be denoted by λ and the set of all isomorphism classes of d-graphs over D by Γ_D.

Example 1.4.1 Let $D = \{a, b\}^2 \cup \{a, c\}^2$. Then the node labelled graph (V, R, φ) with

$$V = \{1, 2, 3, 4, 5\},$$
$$R = \{(1,2), (1,3), (1,4), (1,5), (2,4), (2,5), (3,5), (4,5)\},$$
$$\varphi(1) = a, \varphi(2) = b, \varphi(3) = c, \varphi(4) = b, \varphi(5) = a,$$

is a d-graph. It is (isomorphic to) the graph in Fig.1.3

The following fact follows immediately from the definition:

Proposition 1.4.2 *Let D', D'' be dependencies, (V, R', φ) be a d-graph over D', (V, R'', φ) be a d-graph over D''. If $D' \subseteq D''$, then $R' \subseteq R''$.*

A vertex of a d-graph with no arcs leaving it (leading to it) is said to be a *maximum*(*minimum*, resp.) vertex. Clearly, a d-graph has at least one maximum

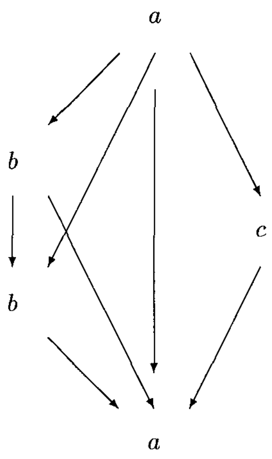

Figure 1.3: A dependence graph over $D = \{a,b\}^2 \cup \{a,c\}^2$.

and one minimum vertex. Since d-graphs are acyclic, the transitive and reflexive closure of d-graph arc relation is an ordering. Thus, each d-graph uniquely determines an ordering of symbol occurrences. This ordering will be discussed later on; for the time being we only mention that considering d-graphs as descriptions of non-sequential processes and dependency relation as a model of causal dependency, this ordering may be viewed as causal ordering of process events.

Observe that in case of full dependency D (i.e. if $D = S_D \times S_D$) the arc relation of any d-graph g over D is the linear order of vertices of g; in case of minimum dependency D (i.e. when D is an identity relation) any d-graph over D consists of a number of connected components, each of them being a linearly ordered set of vertices labelled with a common symbol.

Define the composition of d-graphs as follows: for all graphs γ_1, γ_2 in Γ_D the composition $(\gamma_1 \circ \gamma_2)$ with γ_1 with γ_2 is a graph arising from the disjoint union of γ_1 and γ_2 by adding to it new arcs leading from each node of γ_1 to each node of γ_2, provided they are labelled with dependent symbols. Formally, $(V, R, \varphi) \simeq (\gamma_1 \circ \gamma_2)$ iff there are instances $(V_1, R_1, \varphi_1), (V_2, R_2, \varphi_2)$ of γ_1, γ_2, respectively, such that

$$V = V_1 \cup V_2, V_1 \cap V_2 = \emptyset, \tag{1.23}$$
$$R = R_1 \cup R_2 \cup R_{12}, \tag{1.24}$$
$$\varphi = \varphi_1 \cup \varphi_2, \tag{1.25}$$

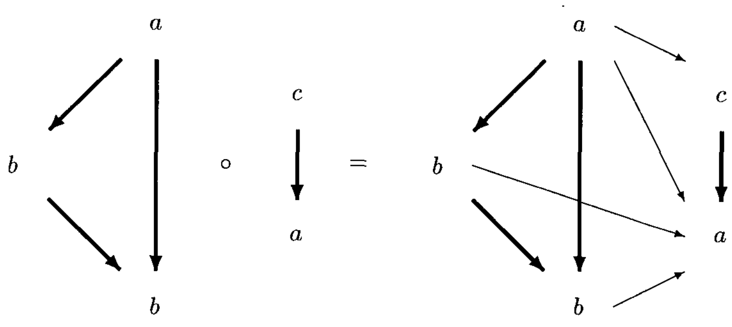

Figure 1.4: D-graphs composition

where R_{12} denotes a binary relation in V such that

$$(v_1, v_2) \in R_{12} \Leftrightarrow v_1 \in V_1 \wedge v_2 \in V_2 \wedge (\varphi(v_1), \varphi(v_2)) \in D. \qquad (1.26)$$

Example 1.4.3 In Fig. 1.4 the composition of two d-graphs is presented; thin arrows represent arcs added in the composition.

Proposition 1.4.4 *The composition of d-graphs is a d-graph.*

Proof: In view of (1.23) and (1.25) it is clear that the composition γ of two d-graphs γ_1, γ_2 is a finite graph, with nodes labelled with symbols from Σ_D. It is acyclic, since γ_1 and γ_2 are acyclic and by (1.26), in γ there is no arcs leading from nodes of γ_2 to nodes of γ_1. Let v_1, v_2 be nodes of γ with $(\varphi(v_1), \varphi(v_2)) \in D$. If both of them are nodes of γ_1 or of γ_2, then by D-connectivity of components and by (1.24) they are also joined in γ. If v_1 is a node of γ_1 and v_2 is a node of γ_2, then by (1.24) and (1.26) they are joined in γ, which proves D-connectivity of γ. □

Composition of d-graphs can be viewed as "sequential" as well as "parallel": composing two independent d-graphs (i.e. d-graphs such that any node of one of them is independent of any node of the other) we get the result intuitively understood as "parallel" composition, while composing two dependent d-graphs, i.e. such that any node of one of them is dependent of any label of the other, the result can be viewed as the "sequential" or "serial" composition. In general, d-graph composition is a mixture of "parallel" and "sequential" compositions, where some (but not all) nodes of one d-graph are dependent on some nodes of the other, and some of them are not. The nature of composition of d-graphs depends on the nature of the underlying dependency relation.

Denote the empty graph (with no nodes) by e. The set of all d-graphs over a dependency D with composition $\circ$ defined as above and with the empty graph as a distinguished element forms an algebra denoted by $G(D)$.

Theorem 1.4.5 *The $G(D)$ is a monoid.*

Proof: Since the empty graph is obviously the neutral (left and right) element w.r.t. to the composition, it suffices to show that the composition of d-graphs is associative. Let, for $i = 1, 2, 3, (V_i, R_i, \varphi_i)$ be a representative of d-graph γ_i, such that $V_i \cap V_j = \emptyset$ for $i \neq j$. By simple calculation we prove that $((\gamma_1 \circ \gamma_2) \circ \gamma_3)$ is (isomorphic to) the d-graph (V, R, φ) with

$$V = V_1 \cup V_2 \cup V_3, \tag{1.27}$$
$$R = R_1 \cup R_2 \cup R_3 \cup R_{12} \cup R_{13} \cup R_{23} \tag{1.28}$$
$$\varphi = \varphi_1 \cup \varphi_2 \cup \varphi_3, \tag{1.29}$$

where R_{ij} denotes a binary relation in V such that

$$(v_1, v_2) \in R_{ij} \Leftrightarrow v_1 \in V_i \wedge v_2 \in V_j \wedge (\varphi(v_1), \varphi(v_2)) \in D,$$

and the same result we obtain for $(\gamma_1 \circ (\gamma_2 \circ \gamma_3))$. □

Let D be a dependency. For each string w over S_D denote by $\langle w \rangle_D$ the d-graph defined recursively:

$$\langle \epsilon \rangle_D = e, \quad \langle wa \rangle_D = \langle w \rangle_D \circ (a, \emptyset, \{(a, a)\}) \tag{1.30}$$

for all strings w and symbols a. In other words, $\langle wa \rangle$ arises from the graph $\langle w \rangle$ by adding to it a new node labelled with symbol a and new arcs leading to it from all vertices of $\langle w \rangle$ labelled with symbols dependent of a.

D-graph $\langle abbca \rangle_D$ with dependency $D = \{a, b\}^2 \cup \{a, c\}^2$ is presented in Fig. 1.3.

Let dependency D be fixed from now on and let $\Sigma, I, \langle w \rangle$ denote $\Sigma_D, I_D, \langle w \rangle_D$, respectively.

Proposition 1.4.6 *For each d-graph γ over D there is a string w such that $\langle w \rangle = \gamma$.*

Proof: It is clear for the empty d-graph. By induction, let γ be a non-empty d-graph; remove from γ one of its maximum vertices. It is clear that the resulting graph is a d-graph. By induction hypothesis there is a string w such that $\langle w \rangle$ is isomorphic with this restricted d-graph; then $\langle wa \rangle$, where a is the label of the removed vertex, is isomorphic with the original graph. □

Therefore, the mapping $\langle \rangle$ is a surjection.

Proposition 1.4.7 *Mapping $\phi : \Sigma^* \longrightarrow G(D)$ defined by $\phi(w) = \langle w \rangle$ is a dependency morphism w.r.t. D.*

Proof: By Proposition 1.4.6 mapping ϕ is a surjection. By an easy induction we prove that for each strings u, v

$$\phi(uv) = \langle uv \rangle = \langle u \rangle \circ \langle v \rangle = \phi(u) \circ \phi(v),$$

and clearly $\phi(\epsilon) = \langle \epsilon \rangle = e$. Thus, ϕ is a homomorphism onto $G(D)$. Condition A1 is obviously satisfied. If $(a, b) \in I$, d-graph $\langle ab \rangle$ has no arcs, hence it is isomorphic with $\langle ba \rangle$. It proves A2. By definition, d-graph $\langle ua \rangle$, for string u and symbol a, has a maximum vertex labelled with a; if $\langle ua \rangle = \langle v \rangle$, then $\langle v \rangle$ has also a maximum vertex labelled with a; removing these vertices from both graphs results in isomorphic graphs; it is easy to see that removing such a vertex from $\langle v \rangle$ results in $\langle v \div a \rangle$. Hence, $\langle u \rangle = \langle v \div a \rangle$, which proves A3. If $\gamma = \langle ua \rangle = \langle vb \rangle$ and $a \neq b$, then γ has at least two maximum vertices, one of them labelled with a and the second labelled with b. Since both of them are maximum vertices, there is no arc joining them. It proves $(a, b) \in I$ and A4 is satisfied. □

Theorem 1.4.8 *The trace monoid $\mathbb{M}(D)$ is isomorphic with the monoid $G(D)$ of d-graphs over dependency D; the isomorphism is induced by the bijection θ such that $\theta([a]_D) = \langle a \rangle_D$ for all $a \in \Sigma_D$.*

Proof: It follows from Corollary to Theorem 1.3.9. □

The above theorem claims that traces and d-graphs over the same dependency can be viewed as two faces of the same coin; the same concepts can be expressed in two different ways: speaking about traces the algebraic character of the concept is stressed upon, while speaking about d-graphs its causality (or ordering) features are emphasized. We can consider some graph - theoretical features of traces (e.g. connectivity of traces) as well as some algebraic properties of dependence graphs (e.g. composition of d-graphs). Using this isomorphism, one can prove facts about traces using graphical methods and the other way around, prove some graph properties using algebraic methods. In fact, dual nature of traces, algebraic and graph-theoretical, was the principal motivation to introduce them for representing concurrent processes.

An illustration of graph-theoretical properties of traces consider the notion (useful in the other part of this book) of connected trace: a trace is connected, if the d-graph corresponding to it is connected. The algebraic definition of this notion could be the following: trace t is connected, if there is no non-empty traces t_1, t_2 with $t = t_1 t_2$ and such that $\text{Alph}(t_1) \times \text{Alph}(t_2) \subseteq I$. A *connected component* of trace t is the trace corresponding to a connected component of the d-graph corresponding to t; the algebraic definition of this notion is much more complex.

As the second illustration of graph representation of traces let us represent in graphical form projection π_1 of trace $[abcd]$ with dependency $\{a, b, c\}^2 \cup \{b, c, d\}^2$ onto dependency $\{a, c\}^2 \cup \{c, d\}^2$ and projection π_2 of the same trace onto dependency $\{a, b\}^2 \cup \{a, c\}^2 \cup \{c, d\}^2$ (Fig. 1.5). Projection π_1 is onto dependency with smaller alphabet than the original (symbol b is deleted under projection); projection π_2 preserves all symbols, but delete some dependencies, namely dependency (b, c) and (b, d).

$\{a,c\}^2 \cup \{c,d\}^2$ $\{a,b,c\}^2 \cup \{b,c,d\}^2$ $\{a,b\}^2 \cup \{a,c\}^2 \cup \{c,d\}^2$

$$\begin{tikzcd}
 & a \arrow[dd] & & & & d \arrow[dl] \arrow[dd] & & & & a \arrow[dl] \arrow[dd] \\
 & & & & b \arrow[dr] \arrow[dd] & & & & b & \\
 & c \arrow[dl] & {} & {} \arrow[l, "\pi_1"'] & & c \arrow[dl] & {} \arrow[r, "\pi_2"] & {} & & c \arrow[dl] \\
d & & & & d & & & & d &
\end{tikzcd}$$

Figure 1.5: Trace projections

1.5 Histories

The concepts presented in this section originate in papers of M.W. Shields [253, 254, 255]. The main idea is to represent non-sequential processes by a collection of individual histories of concurrently running components; an individual history is a string of events concerning only one component, and the global history is a collection of individual ones. This approach, appealing directly to the intuitive meaning of parallel processing, is particularly well suited to CSP-like systems [138] where individual components run independently of each other, with one exception: an event concerning a number of (in CSP at most two) components can occur only coincidently in all these components ("handshaking" or "rendez-vous" synchronization principle). The presentation and the terminology used here have been adjusted to the present purposes and differ from those of the authors.

Let $\mathbf{\Sigma} = (\Sigma_1, \Sigma_2, \ldots, \Sigma_n)$ be a n-tuple of finite alphabets. Denote by $P(\mathbf{\Sigma})$ the product monoid

$$\Sigma_1^* \times \Sigma_2^* \times \ldots \times \Sigma_n^*. \tag{1.31}$$

The composition in $P(\mathbf{\Sigma})$ is component-wise: if $\mathbf{u} = (u_1, u_2, \ldots, u_n), \mathbf{v} = (v_1, v_2, \ldots, v_n)$, then

$$\mathbf{uv} = (u_1v_1, u_2v_2, \ldots, u_nv_n), \tag{1.32}$$

for all $\mathbf{u}, \mathbf{v} \in \Sigma_1^* \times \Sigma_2^* \times \ldots \times \Sigma_n^*$.

Let $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \cdots \cup \Sigma_n$ and let π_i be the projection from Σ onto Σ_i, for $i = 1, 2, \ldots, n$. By the *distribution* in $\mathbf{\Sigma}$ we understand here the mapping $\pi : \Sigma^* \longrightarrow \Sigma_1^* \times \Sigma_2^* \times \ldots \Sigma_n^*$ defined by the equality:

$$\pi(w) = (\pi_1(w), \pi_2(w), \ldots, \pi_n(w)). \tag{1.33}$$

For each $a \in \Sigma$ the tuple $\pi(a)$ will be called the *elementary history* of a. Thus, the elementary history of a is the n-tuple $(a_1, a_2, \ldots, a_n)$ such that

$$a_i = \begin{cases} a, & \text{if } a \in \Sigma_i, \\ \epsilon, & \text{otherwise.} \end{cases}$$

Thus, elementary history $\pi(a)$ is n-tuple consisting of one symbol strings a that occur in positions corresponding to components containing symbol a and of empty strings that occur in the remaining positions.

Let $H(\Sigma)$ be the submonoid of $P(\Sigma)$ generated by the set of all elementary histories in $P(\Sigma)$. Elements of $H(\Sigma)$ will be called *global histories* (or simply, *histories*), and components of global histories - *individual histories* in $\mathbf{\Sigma}$.

Example 1.5.1 Let $\Sigma_1 = \{a, b\}, \Sigma_2 = \{a, c\}, \mathbf{\Sigma} = (\Sigma_1, \Sigma_2)$. Then $\Sigma = \{a, b, c\}$ and the elementary histories of $\mathbf{\Sigma}$ are: $(a, a), (b, \epsilon), (\epsilon, c)$. The pair:

$$\pi(w) = (abba, aca)$$

is a global history in Σ, since

$$\begin{aligned} \pi(w) &= (a, a)(b, \epsilon)(b, \epsilon)(\epsilon, c)(a, a) \\ &= \pi(abbca). \end{aligned}$$

The pair $(abba, cca)$ is not a history, since it cannot be obtained by composition of elementary histories.

From the point of view of concurrent processes the subalgebra of histories can be interpreted as follows. There is n sequential components of a concurrent system, each of them is capable to execute (sequentially) some elementary actions, creating in this way a sequential history of the component run. All of components can act independently of each other, but an action common to a number of components can be executed by all of them coincidently. There is no other synchronization mechanisms provided in the system. Then the join action of all components is a n-tuple of individual histories of components; such individual histories are consistent with each other, i.e. – roughly speaking – they can be combined into one global history. The following theorem offers a formal criterion for such a consistency.

Proposition 1.5.2 *The distribution in $\mathbf{\Sigma}$ is a homomorphism from the monoid of strings over Σ onto the monoid of histories $H(\Sigma)$.*

Proof: It is clear that the distribution of any string over Σ is a history in $H(\Sigma)$. It is also clear that any history in $H(\Sigma)$ can be obtained as the distribution from a string over Σ, since any history is a composition of elementary histories: $\pi(a)_1\pi(a)_2 \cdots \pi(a)_m = \pi(a_1a_2 \cdots a_m)$. By properties of projection it follows the congruence of the distribution: $\pi(uv) = \pi(u)\pi(v)$. $\square$

Theorem 1.5.3 *Let $\mathbf{\Sigma} = (\Sigma_1, \Sigma_2, \ldots, \Sigma_n), D = \Sigma_1^2 \cup \Sigma_2^2 \cup \cdots \cup \Sigma_n^2$. The distribution in $\mathbf{\Sigma}$ is a dependency morphism w.r.t. D onto the monoid of histories $H(\mathbf{\Sigma})$.*

Proof: Denote the assumed dependency by D and the induced independency by I. By Proposition 1.5.2 the distribution is a homomorphism onto the monoid of histories. By its definition we have at once $\pi(w) = \pi(\epsilon) \Rightarrow w = \epsilon$, which proves A1. By the definition of dependency, two symbols a, b are dependent iff there is index i such that a as well as b belongs to Σ_i, i.e. a, b are independent iff there is no index i such that Σ_i contains both of them. It proves that if $(a, b) \in I$, then $\pi(ab) = \pi(ba)$, since for all $i = 1, 2, \ldots, n$

$$\pi_i(ab) = \pi_i(ba) = \begin{cases} a, & \text{if } a \in \Sigma_i, \\ b, & \text{if } b \in \Sigma_i, \\ \epsilon, & \text{in the remaining cases.} \end{cases}$$

Thus, A2 holds. To prove A3, assume $\pi(ua) = \pi(v)$; then we have $\pi_i(ua) = \pi_i(v)$ for all $i = 1, 2, \ldots, n$; hence $\pi_i(ua) \div a = \pi_i(v) \div a$; since projection and cancellation commute, $\pi_i(ua \div a) = \pi_i(v \div a)$, i.e. $\pi_i(u) = \pi_i(v \div a)$ for all i, which implies $\pi(u) = \pi(v \div a)$. It proves A3. Suppose now $\pi(ua) = \pi(vb)$; if it were i with $a \in \Sigma_i, b \in \Sigma_i$, it would be $\pi_i(u)a = \pi_i(v)b$, which is impossible. Therefore, there is no such i, i.e. $(a, b) \in I$. □

Theorem 1.5.4 *Monoid of histories $H(\Sigma_1, \Sigma_2, \ldots, \Sigma_n)$ is isomorphic with the monoid of traces over dependency $\Sigma_1^2 \cup \Sigma_2^2 \cup \cdots \cup \Sigma_n^2$.*

Proof: It follows from the Theorem 1.5.3 and Corollary to Theorem 1.3.9. □

Therefore, similarly to the case of d-graphs, histories can be viewed as yet another, "third face" of traces; to represent a trace by a history, the underlying dependency should be converted to a suitable tuple of alphabets. One possibility, but not unique, is to take cliques of dependency as individual alphabets and then to make use of Theorem 1.5.4 for constructing histories. As an example consider trace $[abbca]$ over dependency $\{a, b\}^2 \cup \{a, c\}^2$ as in the previous sections; then the system components are $(\{a, b\}, \{a, c\})$ and the history corresponding to trace $[abbca]$ is the history $(abba, aca)$.

1.6 Ordering of Symbol Occurrences

Traces, dependence graphs and histories are the same objects from the algebraic point of view; since monoids of them are isomorphic, they behave in the same way under composition. However, actually they are different objects; their difference is visible when considering how symbols are ordered in corresponding objects: traces, graphs, and histories.

There are two kinds of ordering defined by strings: the ordering of symbols occurring in a string, called the occurrence ordering, and the ordering of prefixes of a strings, the prefix ordering. Looking at strings as at models of sequential processes, the first determines the order of event occurrences within a process. The second is the order of process states, since each prefix of a string can be interpreted as a partial execution of the process interpreting by this string, and such a partial execution determines uniquely an intermediate state of the process. Both orderings, although different, are closely related: given one of them, the second can be reconstructed.

Traces are intended to be generalizations of strings, hence the ordering given by traces should be a generalization of that given by strings. Actually, it is the case; as we could expect, the ordering resulting from a trace is partial. In this chapter we shall discuss in details the ordering resulting from trace approach.

We start from string ordering, since it gives us some tools and notions used next for defining the trace ordering. We shall consider both kinds of ordering: first the prefix ordering and next the occurrence ordering. Next, we consider ordering defined within dependence graphs; finally, ordering of symbols supplied by histories will be discussed.

In the whole section D will denote a fixed dependency relation, Σ be its alphabet, I the independency relation induced by D. All strings are assumed to be strings over Σ. The set of all traces over D will be denoted by the same symbol as the monoid of traces over D, i.e. by $\mathbb{M}(D)$.

Thus, in contrast to linearly ordered prefixes of a string, the set $Pref(t)$ is ordered by $\lfloor$ only partially. Interpreting a trace as a single run of a system, its prefix structure can be viewed as the (partially) ordered set of all intermediate (global) system states reached during this run. In this interpretation $[\epsilon]$ represents the initial state, incomparable prefixes represent states arising in effect of events occurring concurrently, and the ordering of prefix represents the temporal ordering of system states.

Occurrence ordering in strings. At the beginning of this section we give a couple of recursive definition of auxiliary notions. All of them define some functions on Σ_D^*; in these definitions w denotes always a string over Σ_D, and $e, e', e"$ symbols in Σ_D.

Let a be a symbol, w be a string; the number of occurrences of a in w is here denoted by $w(a)$. Thus, $\epsilon(a) = 0, wa(a) = w(a) + 1, wb(a) = w(a)$ for all strings w and symbols a, b. The *occurrence set* of w is a subset of $\Sigma \times \omega$:

$$\mathrm{Occ}\,(w) = \{(a, n) \mid a \in \Sigma \wedge 1 \leq n \leq w(a)\} \tag{1.34}$$

It is clear that a string and its permutation have the same occurrence set, since they have the same occurrence number of symbols; hence, all representatives of a trace over an arbitrary dependency have the same occurrence sets.

Example 1.6.1 -The occurrence set of string *abbca* is the set

$$\{(a, 1), (a, 2), (b, 1), (b, 2), (c, 1)\}.$$

Let $R \subseteq \Sigma \times \omega$ and A be an alphabet; define the projection $\pi_A(R)$ of R onto A as follows:

$$\pi_A(R) = \{(a, n) \in R \mid a \in A\}. \tag{1.35}$$

It is easy to show that

$$\pi_A(\mathrm{Occ}\,(w)) = \mathrm{Occ}\,(\pi_A(w)). \tag{1.36}$$

Thus, $\pi_A(\mathrm{Occ}\,(w)) \subseteq \mathrm{Occ}\,(w)$.

The *occurrence ordering* in a string w is an ordering $\mathrm{Ord}\,(w)$ in the occurrence set of w defined recursively as follows:

$$\mathrm{Ord}\,(\epsilon) = \emptyset, \quad \mathrm{Ord}\,(wa) = \mathrm{Ord}\,(w) \cup (\mathrm{Occ}\,(wa) \times \{(a, w(a)\}). \tag{1.37}$$

Thus, $\mathrm{Ord}\,(w) \subseteq \mathrm{Occ}\,(w) \times \mathrm{Occ}\,(w)$. It is obvious that $\mathrm{Ord}\,(w)$ is a linear ordering for any string w.

Occurrence ordering in traces. In this section we generalize the notion of occurrence ordering from strings to traces. Let D be a dependency. As it has been already noticed, $u \equiv v$ implies $\mathrm{Occ}\,(u) = \mathrm{Occ}\,(v)$, for all strings u, v. Thus, the occurrence set for a trace can be set as the occurrence set of its arbitrary representative. Define now the occurrence ordering of a trace $[w]$ as the intersection of occurrence orderings of all representatives of $[w]$:

$$\mathrm{Ord}\,([w]) = \bigcap_{u \equiv w} \mathrm{Ord}\,(u). \tag{1.38}$$

This definition is correct, since all instances of a trace have the common occurrence set and the intersection of different linear ordering in a common set is a (partial) ordering of this set. Thus, according to this definition, ordering of all representatives of a single trace form all possible linearizations (extensions to the linear ordering) of the trace occurrence ordering. From this definition it follows that the occurrence ordering of a trace is the greatest ordering common for all its representatives. In the rest of this section we give some properties of this ordering and its alternative definitions.

We can interpret the ordering $\mathrm{Ord}\,([w])$ from the point of view of concurrency as follows. Suppose there is a number of observers looking at the same concurrent process represented by a trace. Observation made by each of them is sequential; each of them sees only one representative of this trace. If two observers discover an opposite ordering of the same events in the observed process, it means that these events are actually not ordered in the process and that the difference noticed by observers results only because their specific points of view. Thus, such an ordering should not be taken into account, and consequently, two events can be considered as really ordered in the process if and only if they are ordered in the same way in all possible observations of the process (all observers agree on the same ordering).

D-graph ordering. Consider now d-graphs over D. Let $\gamma = (V, R, \varphi)$ be a dependence graph. Let, as in case of strings, $\gamma(a)$ denotes the number of vertices of γ labelled with a. Define the occurrence set of γ as the set

$$\mathrm{Occ}\,(\gamma) = \{(a, n) \mid 1 \leq n \leq \gamma(a)\}. \tag{1.39}$$

We say that a vertex v *precedes* vertex u in γ, if there is an arc leading from v to u in γ. Let $v(a, n)$ denotes a vertex of γ labelled with a which is preceded in γ by

$$\begin{tikzcd}
 & (a,1) \arrow[dl] \arrow[ddl] \arrow[ddd] \arrow[dr] & \\
(b,1) \arrow[d] \arrow[ddr] & & (c,1) \arrow[ddl] \\
(b,2) \arrow[dr] & & \\
 & (a,2) &
\end{tikzcd}$$

Figure 1.6. The occurrence relation for d-graph $\langle abbca\rangle_D$ over $D = \{a,b\}^2 \cup \{a,c\}^2$.

precisely $n-1$ vertices labelled with a. Observe that for each element (a,n) in the occurrence set of γ there exists precisely one vertex $v(a,n)$.

Occurrence relation for γ is the binary relation $Q(\gamma)$ in $\mathrm{Occ}\,(\gamma)$ such that

$$((a,n),(b,m)) \in Q(\gamma) \Leftrightarrow (v(a,n),v(b,m)) \in R). \tag{1.40}$$

Example 1.6.2 The diagram of the occurrence relation for the d-graph $\langle abbca\rangle$ over $\{a,b\}^2 \cup \{a,c\}^2$ is given in Fig.1.6

Since $Q(\gamma)$ is acyclic, its transitive and reflexive closure of $Q(\gamma)$ is an ordering relation in the set $\mathrm{Occ}\,(\gamma)$; this ordering will be called the *occurrence ordering* of γ and will be denoted by $\mathrm{Ord}\,(\gamma)$.

Proposition 1.6.3 *For each string w, the ordering $Ord(w)$ is an extension of $Ord(\langle w\rangle)$ to a linear ordering.*

Proof: It follows easily from the definition of arc connections in d-graphs by comparing it with the definition of $\mathrm{Ord}\,(w)$ given by (1.37) $\square$

Proposition 1.6.4 *Let γ be a d-graph, and let S be an extension of $Ord(\gamma)$ to a linear ordering. Then there is a string u such that $S = Ord(u)$ and $g = \langle u\rangle$.*

Proof: By induction. If γ is the empty graph, then set $u = \epsilon$. Let γ be not empty and let S be the extension of Ord (g) to a linear ordering. Let (a, n) be the maximum element of S. Then γ has a maximum vertex labelled with a. Delete from γ this vertex and denote the resulting graph by β; delete also the maximum element from S and denote the resulting ordering by R. Clearly, R is the linear extension of β; by induction hypothesis there is a string, say w, such that $R = \text{Ord}\,(w)$ and $\beta = \langle w \rangle$. But then $S = \text{Ord}\,(wa)$ and $\gamma = \langle wa \rangle$. Thus, $u = wa$ meets the requirement of the proposition. □

Theorem 1.6.5 *For all strings w: $Ord([w]) = Ord(\langle w \rangle)$.*

Proof: Obvious in view of the definition of the trace ordering and propositions 1.6.3 and 1.6.4. □

This theorem states that the intersection of linear orderings of representatives of a trace is the same ordering as the transitive and reflexive closure of the ordering of the d-graph corresponding to that trace.

History ordering. Now, let $(\Sigma_1, \Sigma_2, \ldots, \Sigma_n)$ be n-tuple of finite alphabets, $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \cdots \cup \Sigma_n, D = \Sigma_1^2 \cup \Sigma_2^2 \cup \cdots \cup \Sigma_n^2$, π_i be the projection on Σ_i, and let π be the distribution function.

Fact 1.6.6 *For any string w over Σ and each $i \leq n$:*

$$\pi_i(Ord(w)) = Ord(\pi_i(w)). \tag{1.41}$$

Proof: Equality (1.41) holds for $w = \epsilon$. Let $w = ue$, for string u and symbol e. If $e \notin \Sigma_i$, equality (1.41) holds by induction hypothesis, since $\pi_i(w) = \pi_i(u)$. Let then $a \in \Sigma_i$. By definition, $\text{Ord}\,(ua) = \text{Ord}\,(u) \cup (\text{Occ}\,(ua) \times \{(a, u(a)+1)\})$. By induction hypothesis $\pi_i(\text{Ord}\,(u)) = \text{Ord}\,(\pi_i(u))$; by (1.36) $\pi_i(\text{Occ}\,(ua)) = \text{Occ}\,(\pi_i(ua))$; $a \in \Sigma_i$ implies $\pi_i(u)(a) = u(a)$, hence $\pi(\text{Ord}\,(ua)) = \text{Occ}\,(\pi_i(ua)) \cup \{(a, \pi_i(u)(a) + 1\} = \text{Ord}\,(\pi_i(ua))$. It ends the proof. □

Fact 1.6.7 *For each string w over Σ:*

$$Occ(w) = \bigcup_{i=1}^{n} Occ(\pi_i(w)).$$

Proof: Since, by definition, $\pi_i(\text{Occ}\,(w)) \subseteq \text{Occ}\,(w)$, and $\pi_i(\text{Occ}\,(w)) = \text{Occ}\,(\pi_i(w))$ for each $i \leq n$, hence $\bigcup_{i=1}^{n} \text{Occ}\,(\pi_i(w)) \subseteq \text{Occ}\,(w)$. Let $(e, k) \in \text{Occ}\,(w)$; hence $(a, k) \in \pi_i(\text{Occ}\,(w))$ for i such that $a \in \Sigma_i$; thus, $(a, k) \in \text{Occ}\,(\pi_i(w))$ for this i; but it means that $(a, k) \in \bigcup_{i=1}^{n} \text{Occ}\,(\pi_i(w))$, which completes the proof. □

Let $(w_1, w_2, \ldots, w_n)$ be a global history; define

$$\text{Ord}\,(w_1, w_2, \ldots, w_n) = (\bigcup_{i=1}^{n} \text{Ord}\,(w_i))^* \tag{1.42}$$

The following theorem close the comparison of different types of defining trace ordering.

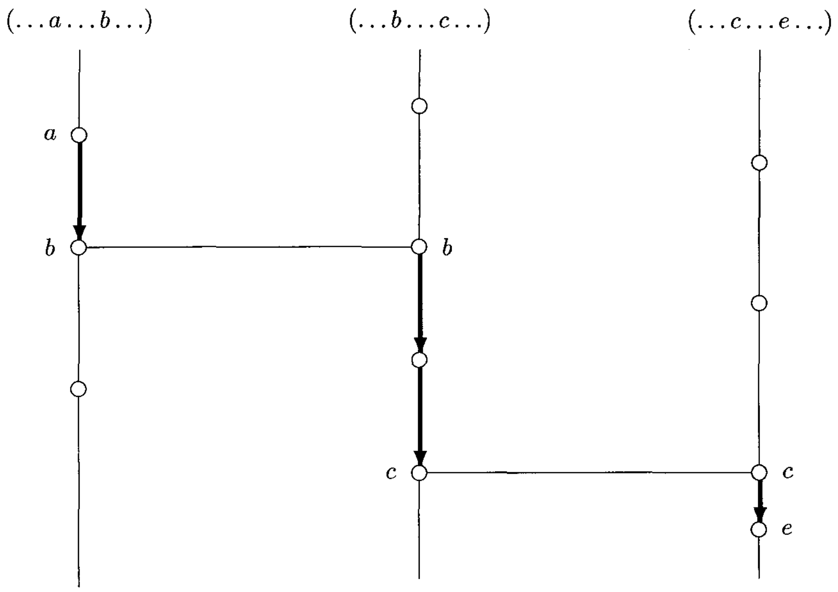

Figure 1.7: History ordering: e follows a.

Theorem 1.6.8 *Let $D = \bigcup \Sigma_i^2$, let $\pi(w)$ be distribution function in $(\Sigma_1, \Sigma_2, \ldots, \Sigma_n)$, and let $\langle w \rangle$ be the d-graph $\langle w \rangle_D$. Then for each string w $Ord(\pi(w)) = Ord(\langle w \rangle)$.*

Proof: Let D be such as assumed above, let $\langle w \rangle$ denotes $\langle w \rangle_D$, and let $\pi(w) = (w_1, w_2, \ldots, w_n)$. Thus, for each $i : w_i = \pi_i(w)$. Since $\mathrm{Occ}\,(w) = \mathrm{Occ}\,(\langle w \rangle)$, and $\mathrm{Occ}\,(w_i) \subseteq \mathrm{Occ}\,(w)$, we have $\mathrm{Occ}\,(w_i) \subseteq \mathrm{Occ}\,(\langle w \rangle)$. Any two symbols in Σ_i are dependent, hence $\mathrm{Ord}\,(w_i) \subseteq \mathrm{Ord}\,(\langle w \rangle)$. Therefore, $\bigcup_{i=1}^n \mathrm{Ord}\,(w_i) \subseteq \mathrm{Ord}\,(\langle w \rangle)$ and since $\mathrm{Ord}\,(\langle w \rangle)$ is an ordering, $(\bigcup_{i=1}^n \mathrm{Ord}\,(w_i))^* \subseteq \mathrm{Ord}\,(\langle w \rangle)$. To prove the inverse inclusion, observe that if there is an arc in an occurrence graph from (e', k') to (e'', k''), then $(e', e'') \in D$; it means that there is i such that $e', e'' \in \Sigma_i$ and that $((e', k'), (e'', k'')) \in \mathrm{Ord}\,(w_i)$, hence $((e', k'), (e'', k'')) \in \mathrm{Ord}\,(\pi(w))$. It ends the proof. □

The history ordering, defined by (1.42), is illustrated in Fig. 1.7.

It explains the principle of history ordering. Let us interpret ordering $\mathrm{Ord}\,(\pi w)$

in similar way as we have done it in case of traces. Each individual history of a component can be viewed as the result of a local observation of a process limited to events belonging to the repertoire of the component. From such a viewpoint, an observer can notice only events local to the component he is situated at; remaining actions are invisible for him. Thus, being localized at i-th component, he notices the ordering $\mathrm{Ord}\,(\pi_i(w))$ only. To discover the global ordering of events in the whole history, all such individual histories have to be put together; then events observed coincidently from different components form a sort of links between individual observations and make possible to discover an ordering between events local to separate and remote components. The rule is: an event occurrence (a'', n'') follows another event occurrence (a', n'), if there is a sequence of event occurrences beginning with (a', n') and ending with (a'', n'') in which every element follows its predecessor according to the individual history containing both of them. Such a principle is used by historians to establish a chronology of events concerning remote and separate historical objects.

Let us comment and compare all three ways of defining ordering within graphs, traces, and histories. They have been defined in the following way:

$\mathrm{Ord}\,(\langle w\rangle)$:	as the transitive and reflexive closure of arc relation of $\langle w\rangle$;
$\mathrm{Ord}\,([w])$:	as intersection of linear orderings of all representatives of $[w]$;
$\mathrm{Ord}\,(\pi(w))$:	as the transitive closure of the union of linear orderings of all $\pi_i(w)$ components.

All of them describe processes as partially ordered sets of event occurrences. They have been defined in different way: in natural way for graphs, as the intersection of individually observed sequential orderings in case of traces, and as the union of individually observed sequential orderings in case of histories. In case of traces, observations were complete, containing every action, but sometimes discovering different orderings of some events; in case of histories, observations were incomplete, limited only to local events, but discovering always the proper ordering of noticed events. In the case of traces the way of obtaining the global ordering is by comparing all individual observations and rejecting a subjective and irrelevant information (by intersection of individual orderings); in case of histories, all individual observations are collected together to gain complementary and relevant information (by unifying individual observations). While the second method has been compared to the method of historians, the first one can be compared to that of physicists. In all three cases the results are the same; it gives an additional argument for the above mentioned definitions and indicates their soundness.

1.7 Trace Languages

Let D be a dependency; by a *trace language* over D we shall mean any set of traces over D. The set of all trace languages over D will be denoted by $\mathbf{T}_D$). For a given set L of strings over Σ_D denote by $[L]_D$ the set $\{[w]_D \mid w \in L\}$; and for a given set

T of traces over D denote by $\bigcup T$ the set $\{w \mid [w]_D \in T\}$. Clearly,

$$L \subseteq \bigcup [L]_D \text{ and } T = [\bigcup T]_D \tag{1.43}$$

for all (string) languages L and (trace) languages T over D. String language L such that $L = \bigcup [L]_D$ is called to be *(existentially) consistent* with dependency D [3]. The composition $T_1 T_2$ of trace languages T_1, T_2 over the same dependency D is the set $\{t_1 t_2 \mid t_1 \in T_2 \wedge t_2 \wedge T_2\}$. The iteration of trace language T over D is defined in the same way as the iteration of string language:

$$T^* = \bigcup_{n=0}^{\infty} T^n$$

where

$$T^0 = [\epsilon]_D \text{ and } T^{n+1} = T^n T.$$

The following proposition results directly from the definitions:

Proposition 1.7.1 *For any dependency D, any string languages L, L_1, L_2 over Σ_D, and for any family $\{L_i\}_{i \in I}$ of string languages over Σ_D:*

$$[\emptyset]_D = \emptyset, \tag{1.44}$$
$$[L_1]_D [L_2]_D = [L_1 L_2]_D, \tag{1.45}$$
$$L_1 \subseteq L_2 \Rightarrow [L_1]_D \subseteq [L_2]_D, \tag{1.46}$$
$$[L_1]_D \cup [L_2]_D = [L_1 \cup L_2]_D, \tag{1.47}$$
$$\bigcup_{i \in I} [L_i]_D = [\bigcup_{i \in I} L_i]_D, \tag{1.48}$$
$$[L]^*{}_D = [L^*]_D. \tag{1.49}$$

Before defining the synchronization of languages (which is the most important operation on languages for the present purpose) let us introduce the notion of projection adjusted for traces and dependencies.

Let C be a dependency, $C \subseteq D$, and let t be a trace over D; then the *trace projection* $\pi_C(t)$ of t onto C is a trace over C defined as follows:

$$\pi_C(t) = \begin{cases} [\epsilon]_C, & \text{if } t = [\epsilon]_D, \\ \pi_C(t_1), & \text{if } t = t_1[e]_D, e \notin \Sigma_C, \\ \pi_C(t_1)[e]_C, & \text{if } t = t_1[e]_D, e \in \Sigma_C. \end{cases}$$

Intuitively, trace projection onto C deletes from traces all symbols not in Σ_C and weakens the dependency within traces. Thus, the projection of a trace over dependency D onto a dependency C which is "smaller", but has the same alphabet as D, is a "weaker" trace, containing more representatives the the original one.

The following proposition is easy to prove:

Proposition 1.7.2 *Let C, D be dependencies, $D \subseteq C$. Then $u \equiv_C v \Rightarrow \pi_D(u) \equiv_D \pi_D(v)$.*

Define the *synchronization* of the string language L_1 over Σ_1 with the string language L_2 over Σ_2 as the string language $(L_1 \parallel L_2)$ over $(\Sigma_1 \cup \Sigma_2)$ such that

$$w \in (L_1 \parallel L_2) \Leftrightarrow \pi_{\Sigma_1}(w) \in L_1 \wedge \pi_{\Sigma_2}(w) \in L_2.$$

Proposition 1.7.3 *Let L, L_1, L_2, L_3 be string languages, Σ_1 be the alphabet of L_1, Σ_2 be the alphabet of L_2, let $\{L_i\}_{i \in I}$ be a family of string languages over a common alphabet, and let w be a string over $\Sigma_1 \cup \Sigma_2$. Then:*

$$\begin{aligned}
&L \parallel L = L,\\
&L_1 \parallel L_2 = L_2 \parallel L_1,\\
&L_1 \parallel (L_2 \parallel L_3) = (L_1 \parallel L_2) \parallel L_3,\\
&\{[\epsilon]_{\Sigma_1}\} \parallel \{[\epsilon]_{\Sigma_2}\} = \{[\epsilon]_{\Sigma_1 \cup \Sigma_2}\},\\
&(\bigcup_{i \in I} L_i) \parallel L = \bigcup_{i \in I}(L_i \parallel L)\\
&(L_1 \parallel L_2)\{w\} = (L_1\{\pi_{\Sigma_1}(w)\}) \parallel (L_2\{\pi_{\Sigma_2}(w)\}),\\
&\{w\}(L_1 \parallel L_2) = (\{\pi_{\Sigma_1}(w)\}L_1) \parallel (\{\pi_{\Sigma_2}(w)\}L_2).
\end{aligned}$$

Proof: Equalities (1.50) – (1.50) are obvious. Let $w \in (\bigcup_{i \in I} L_i) \parallel L$; it means that $\pi_{\Sigma'}(w) \in (\bigcup_{i \in I} L_i)$ and $\pi_{\Sigma}(w) \in (L)$; then $\exists i \in I : \pi_{\Sigma'}(w) \in L_i$ and $\pi_{\Sigma}(w) \in L$, i.e. $w \in (\bigcup_{i \in I}(L_i \parallel L)$, which proves (1.50). To prove (1.50), let $u \in (L_1 \parallel L_2)\{w\}$; it means that there exists v such that $v \in (L_1 \parallel L_2)$ and $u = vw$; by definition of synchronization, $\pi_{\Sigma_1}(v) \in L_1$ and $\pi_{\Sigma_2}(v) \in L_2$; because w is a string over $\Sigma_1 \cup \Sigma_2$, this is equivalent to $\pi_{\Sigma_1}(vw) \in L_1\{\pi_{\Sigma_1}(w)\}$ and $\pi_{\Sigma_2}(vw) \in L_2\{\pi_{\Sigma_2}(w)\}$; it means that $u \in (L_1\{\pi_{\Sigma_1}(w)\}) \parallel (L_2\{\pi_{\Sigma_2}(w)\})$. Proof of (1.50) is similar. □

Define the *synchronization* of the trace language T_1 over D_1 with the trace language T_2 over D_2 as the trace language $(T_1 \parallel T_2)$ over $(D_1 \cup D_2)$ such that

$$t \in (T_1 \parallel T_2) \Leftrightarrow \pi_{D_1}(t) \in T_1 \wedge \pi_{D_2}(t) \in T_2.$$

Proposition 1.7.4 *For any dependencies D_1, D_2 and any string languages L_1 over Σ_{D_1}, L_2 over Σ_{D_2}:*

$$[L_1]_{D_1} \parallel [L_2]_{D_2} = [L_1 \parallel L_2]_{D_1 \cup D_2}$$

Proof:

$$\begin{aligned}
[L_1]_{D_1} \parallel [L_2]_{D_2} &= \{t \mid \pi_{D_1}(t) \in [L_1]_{D_1} \wedge \pi_{D_2}(t) \in [L_2]_{D_2}\}\\
&= \{[w]_{D_1 \cup D_2} \mid \pi_{S_{D_1}}(w) \in L_1 \wedge \pi_{S_{D_2}}(w) \in L_2\}\\
&= \{[w]_{D_1 \cup D_2} \mid w \in L_1 \parallel L_2\}\\
&= [\{w \mid w \in (L_1 \parallel L_2)\}]_{D_1 \cup D_2}\\
&= [L_1 \parallel L_2]_{D_1 \cup D_2}.
\end{aligned}$$

□

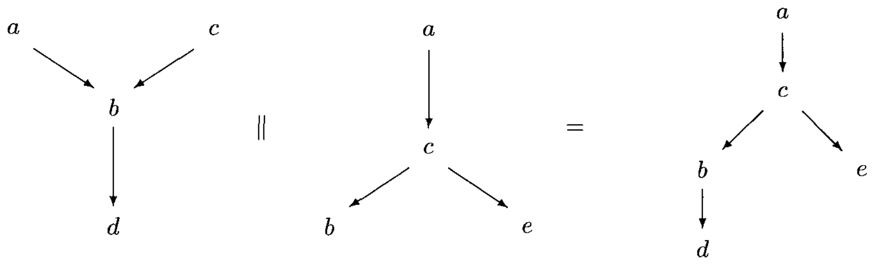

Figure 1.8: Synchronization of two singleton trace languages.

Proposition 1.7.5 *Let T, T_1, T_2, T_3 be trace languages, let D_1 be the dependency of T_1, D_2 be the dependency of T_2, let $\{T_i\}_{i \in I}$ be a family of trace languages over a common dependency, and let t be a trace over $D_1 \cup D_2$. Then:*

$$T \parallel T = T, \tag{1.50}$$

$$T_1 \parallel T_2 = T_2 \parallel T_1, \tag{1.51}$$

$$T_1 \parallel (T_2 \parallel T_3) = (T_1 \parallel T_2) \parallel T_3, \tag{1.52}$$

$$\{[\epsilon]_{D_1}\} \parallel \{[\epsilon]_{D_2}\} = \{[\epsilon]_{D_1 \cup D_2}\}, \tag{1.53}$$

$$(\bigcup_{i \in I} T_i) \parallel T = \bigcup_{i \in I} (T_i \parallel T) \tag{1.54}$$

$$(T_1 \parallel T_2)\{t\} = (T_1\{\pi_{D_1}(t)\}) \parallel (T_2\{\pi_{D_2}(t)\}), \tag{1.55}$$

$$\{t\}(T_1 \parallel T_2) = (\{\pi_{D_1}(t)\}T_1) \parallel (\{\pi_{D_2}(t)\}T_2). \tag{1.56}$$

Proof: It is similar to that of Proposition 1.7.3. □

Example 1.7.6 Let $D_1 = \{a, b, d\}^2 \cup \{b, c, d\}^2, D_2 = \{a, b, c\}^2 \cup \{a, c, e\}^2, T_1 = \{[cabd]_{D_1}\}, T_2 = \{[acbe]_{D_2}\}$. The synchronization $T_1 \parallel T_2$ is the language $\{[acbde]_D\}$ over dependency $D = D_1 \cup D_2 = \{a, b, c, d\}^2 \cup \{a, c, e\}^2$. Synchronization of these two trace languages is illustrated in Fig. 1.8 where traces are represented by corresponding d-graphs (without arcs resulting by transitivity from others).

Observe that in a sense the synchronization operation is inverse w. r. to projection, namely we have for any trace t over $D_1 \cup D_2$ the equality:

$$\{t\} = \{\pi_{D_1}(t)\} \parallel \{\pi_{D_2}(t)\}.$$

Observe also that for any traces t_1 over D_1, t_2 over D_2, the synchronization $\{t_1\} \parallel \{t_2\}$ is either empty, or a singleton set. Thus the synchronization can be viewed as a partial operation on traces. In general, it is not so in case of strings: synchronization of two singleton string languages may not be a singleton. E.g., the synchronization of the string language $\{ab\}$ over $\{a, b\}$ with the string language $\{cd\}$ over $\{c, d\}$ is the string language $\{abcd, acbd, cabd, acdb, cadb, cdab\}$, while the synchronization of the trace language $\{[ab]\}$ over $\{a, b\}^2$ with the trace language $\{[cd]\}$ over $\{c, d\}^2$ is the singleton trace language $\{[abcd]\}$ over $\{a, b\}^2 \cup \{c, d\}^2$. It is yet another justification of using traces instead of strings for representing runs of concurrent systems.

Synchronization operation can be easily extended to an arbitrary number of arguments: for a family of trace languages $\{T_i\}_{i \in I}$, with T_i being a trace language over D_i, we define $\|_{i \in I} T_i$ as the trace language over $\bigcup_{i \in I} D_i$ such that

$$t \in \|_{i \in I} T_i \Leftrightarrow \forall_{i \in I} \pi_{D_i}(t) \in T_i.$$

Fact 1.7.7 *If T_1, T_2 are trace languages over a common dependency, then $T_1 \parallel T_2 = T_1 \cap T_2$.*

Proof: Let T_1, T_2 be trace languages over dependency D; then $t \in (T_1 \parallel T_2)$ if and only if $\pi_D(t) \in T_1$ and $\pi_D(t) \in T_2$; but since t is a trace over D, $\pi_D(t) = t$, hence $t \in T_1$ and $t \in T_2$. □

A trace language T is *prefix-closed*, if $s \sqsubseteq t \in T \Rightarrow s \in T$.

Proposition 1.7.8 *If T_1, T_2 are prefix-closed trace languages, then so is their synchronization $T_1 \parallel T_2$.*

Proof: Let T_1, T_2 be prefix-closed trace languages over dependencies D_1, D_2, respectively. Let $t \in (T_1 \parallel T_2)$ and let t' be a prefix of t; then there is t'' such that $t = t't''$. But then, by properties of projection, $\pi_{D_1}(t) = \pi_{D_1}(t't'') = \pi_{D_1}(t')\pi_{D_1}(t'') \in T_1$ and $\pi_{D_2}(t) = \pi_{D_2}(t't'') = \pi_{D_2}(t')\pi_{D_2}(t'') \in T_2$; since T_1, T_2 are prefix-closed, $\pi_{D_1}(t') \in T_1$ and $\pi_{D_2}(t') \in T_2$; hence, by definition, $t' \in (T_1 \parallel T_2)$. □

Fixed-point equations. Let $\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n, \mathbf{Y}$ be families of sets, $n > 0$, and let $f : \mathbf{X}_1 \times \mathbf{X}_2 \times \cdots \times \mathbf{X}_n \longrightarrow \mathbf{Y}$; f is *monotone*, if

$$(\forall i : X_i' \subseteq X_i'') \Rightarrow f(X_1', X_2', \ldots X_n') \subseteq f(X_1'', X_2'', \ldots, X_n'')$$

for each $X_i', X_i'' \in \mathbf{X}_i, i = 1, 2, \ldots, n$.

Fact 1.7.9 *Concatenation, union, iteration, and synchronization, operations on string (trace) languages are monotone.*

Proof: Obvious in view of corresponding definitions. □

Fact 1.7.10 *Superposition (composition) of any number of monotone operations is monotone.*

Proof: Clear. □

Let $D_1, D_2, \ldots, D_n, D$ be dependencies, $n > 0$, and let

$$f : 2^{\Sigma^*_{D_1}} \times 2^{\Sigma^*_{D_2}} \times \cdots \times 2^{\Sigma^*_{D_n}} \longrightarrow 2^{\Sigma^*_D};$$

f is *congruent*, if

$$(\forall i : [X'_i]_{D_i} = [X''_i]_{D_i}) \Rightarrow [f(X'_1, X'_2, \ldots X'_n)]_D = [f(X''_1, X''_2, \ldots, X''_n)]_D$$

for all $X'_i, X''_i \in 2^{\Sigma^*_{D_i}}, i = 1, 2, \ldots, n$.

Fact 1.7.11 *Concatenation, union, iteration, and synchronization operations on string languages are congruent.*

Proof: It follows from Proposition 1.7.1 and Proposition 1.7.4. □

Fact 1.7.12 *Superposition (composition) of any number of congruent operations is congruent.*

Proof: Clear. □

Let $D_1, D_2, \ldots, D_n, D$ be dependencies, $n > 0$, and let

$$f : 2^{\Sigma^*_{D_1}} \times 2^{\Sigma^*_{D_2}} \times \cdots \times 2^{\Sigma^*_{D_n}} \longrightarrow 2^{\Sigma^*_D}$$

be *congruent.* Denote by $[f]_D$ the function

$$[f]_D : 2^{\mathbf{T}_{D_1}} \times 2^{\mathbf{T}_{D_2}} \times \cdots \times 2^{\mathbf{T}_{D_n}} \longrightarrow 2^{\mathbf{T}_D}.$$

defined by the equality

$$[f]_D([L_1]_{D_1}, [L_2]_{D_2}, \ldots, [L_n]_{D_n}) = [f(L_1, L_2, \ldots, L_n)]_D$$

This definition is correct by congruence of f.

We say that the set x_0 is the least fixed point of function $f : 2^X \longrightarrow 2^X$, if $f(x_0) = x_0$ and for any set x with $f(x) = x$ the inclusion $x_0 \subseteq x$ holds.

Theorem 1.7.13 *Let D be a dependency and $f : 2^{\Sigma^*_D} \longrightarrow 2^{\Sigma^*_D}$ be monotone and congruent. If L_0 is the least fixed point of f, then $[L_0]_D$ is the least fixed point of $[f]_D$.*

Proof: Let D, f, and L_0 be such as defined in the formulation of Theorem. First observe that for any L with $f(L) \subseteq L$ we have $L_0 \subseteq L$. Indeed, let L' be the least set of strings such that $f(L') \subseteq L'$; then $L' \subseteq L$. Since f is monotone, $f(f(L') \subseteq f(L')$, hence $f(L')$ meets also the inclusion and consequently $L' \subseteq f(L')$; hence, $L' = f(L')$ and by the definition of L_0 we have $L_0 \subseteq L' \subseteq L$. Now,

$$[L_0]_D = [f(L_0)]_D = [f]_D([L_0]_D)$$

by congruence of f. Thus, $[L_0]_D$ is a fixed point of $[f]_D$. We show that $[L_0]_D$ is the least fixed point of $[f]_D$. Let T be a trace language over D such that $[f]_D(T) = T$ and set $L = \bigcup T$. Thus $[L]_D = T$ and $\bigcup T = \bigcup [L]_D = L$. By (1.43) we have $f(L) \subseteq \bigcup [f(L)]_D$; by congruence of f we have $\bigcup [f(L)]_D = \bigcup [f]_D([L]_D)$; by definition of L we have $\bigcup [f]_D([L]_D) = \bigcup [f]_D(T)$, and by definition of T we get $\bigcup [f]_D(T) = \bigcup T$ and again by definition of L: $\bigcup T = L$. Collecting all above relations together we get $f(L) \subseteq L$. Thus, as we have already proved, $L_0 \subseteq L$, hence $[L_0]_D \subseteq [L]_D = T$. It proves $[L_0]$ to be the least fixed point of $[f]_D$. □

This theorem can be easily generalized for tuples of monotonic and congruent functions. As we have already mention, functions built up from variables and constants by means of union, concatenation and iteration operations can serve as examples of monotonic and congruent functions. Theorem 1.7.13 allows to lift the well-known and broadly applied method of defining string languages by fixed point equations to the case of trace languages. It offers also a useful and handy tool for analysis concurrent systems represented by elementary net systems, as shown in the next section.

It is worthwhile to note that, under the same assumptions as in Theorem 1.7.13, M_0 can be the greatest fixed point of a function f, while $[M_0]$ is not the greatest fixed point of $[f]$.

1.8 Elementary Net Systems

In this section we show how the trace theory is related to net models of concurrent systems. First, *elementary net systems*, introduced in [238] on the base of Petri nets and will be presented. The behaviour of such systems will be represented by string languages describing various sequences of system actions that can be executed while the system is running. Next, the trace behaviour will be defined and compared with the string behaviour. Finally, compositionality of behaviours will be shown: the behaviour of a net composed from other nets as components turns out to be the synchronization of the components behaviour.

Elementary net systems. By an *elementary net system*, called simply *net* from now on, we understand any ordered quadruple

$$N = (P_N, E_N, F_N, m_N^0)$$

where P_N, E_N are finite disjoint sets (of *places* and *transitions*, resp.), $F_N \subseteq P_N \times E_N \cup E_N \times P_N$ (the *flow* relation) and $m_N^0 \subseteq P_N$. It is assumed that $Dom(F_N) \cup Cod(F_N) = P_N \cup E_N$ (there is no "isolated" places and transitions) and $F \cap F^{-1} = \emptyset$. Any subset of P_N is called a *marking* of N; m_N^0 is called the (*initial marking*). A place in a marking is said to be *marked*, or *carrying a token.*

As all other types of nets, elementary net systems are represented graphically using boxes to represent transitions, circles to represent places, and arrows leading from circles to boxes or from boxes to circles to represent the flow relation; in such a representation circles corresponding to places in the initial marking are marked

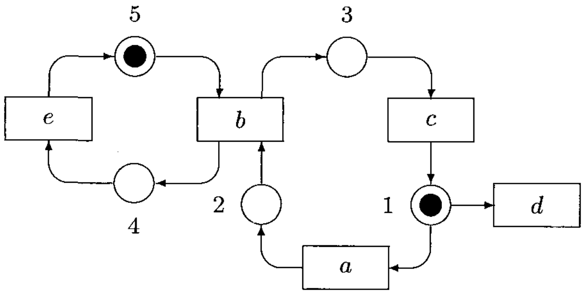

Figure 1.9: Example of an elementary net system.

with dots. In Fig. 1.9 the net (P, E, F, m) with

$$\begin{aligned}
P &= \{1,2,3,4,5,6\},\\
E &= \{a,b,c,d,e\},\\
F &= \{(1,a),(a,2),(2,b),(b,3),(3,c),(c,1),\\
&\quad (1,d),(d,4),(5,b),(b,6),(6,e),(e,5)\}),\\
m &= \{1,5\}
\end{aligned}$$

is represented graphically.

Define Pre, Post, Prox as functions from E to 2^P such that for all $a \in E$: as follows:

$$\operatorname{Pre}_N(a) = \{p \mid (p,a) \in F_N\}; \tag{1.57}$$
$$\operatorname{Post}_N(a) = \{p \mid (e,p) \in F_N\}; \tag{1.58}$$
$$\operatorname{Prox}_N(a) = \operatorname{Pre}_N(a) \cup \operatorname{Post}_N(a); \tag{1.59}$$

places in $\operatorname{Pre}_N(a)$ are called the *entry* places of a, those in $\operatorname{Post}_N(a)$ are called the *exit* places of a, and those in $\operatorname{Prox}_N(a)$ are *neighbours* of a; the set $\operatorname{Prox}_N(a)$ is the neighbourhood of a in N. The assumption $F \cap F^{-1} = \emptyset$ means that no place can be an entry and an exit of the same transition. The subscript N is omitted if the net is understood.

Transition function of net N is a (partial) function $\delta_N : 2^P_N \times E_N \longrightarrow 2^{P_N}$ such that

$$\delta_N(m_1, a) = m_2 \Leftrightarrow \begin{cases} \operatorname{Pre}(a) \subseteq m_1, \operatorname{Post}(a) \cap m_1 = \emptyset, \\ m_2 = (m_1 - \operatorname{Pre}(a)) \cup \operatorname{Post}(a); \end{cases}$$

As usual, subscript N is omitted, if the net is understood; this convention will hold for all subsequent notions and symbols related to them.

Clearly, this definition is correct, since for each marking m and transition a there exists at most one marking equal to $\delta(m, a)$. If $\delta(m, a)$ is defined for marking m and transition a, we say that a is *enabled* at marking m. We say that the transition function describes single steps of the system.

Reachability function of net N is the function $\delta_N^* : 2^P \times E^* \longrightarrow 2^P$ defined recursively by equalities:

$$\delta_N^*(m, \epsilon) = m, \tag{1.60}$$

$$\delta_N^*(m, wa) = \delta_N(\delta_N^*(m, w), a) \tag{1.61}$$

for all $m \in 2^P, w \in E^*, a \in E$.

Behavioural function of net N is the function $\beta_N : E^* \longrightarrow 2^P$ defined by $\beta_N(w) = \delta^*(m^0, w)$ for all $w \in E^*$.

As usual, the subscript N is omitted everywhere the net N is understood.

The set (of strings) $S_N = Dom(\beta)$ is the (sequential) behaviour of net N; the set (of markings) $R_N = Cod(\beta)$ is the set of *reachable* markings of N. Elements of S will be called *execution sequences* of N; execution sequence w such that $\beta(w) = m \subseteq P$ is said to *lead* to m.

The *sequential behaviour* of N is clearly a prefix closed language and as any prefix closed language it is ordered into a tree by the prefix relation. Maximal linearly ordered subsets of S can be viewed as sequential observations of the behaviour of N, i.e. observations made by observers capable to see only a single event occurrence at a time. The ordering of symbols in strings of S reflects not only the (objective) causal ordering of event occurrences but also a (subjective) observational ordering resulting from a specific view over concurrent actions. Therefore, the structure of S alone does not allow to decide whether the difference in ordering is caused by a conflict resolution (a decision made in the system), or by different observations of concurrency. In order to extract from S the causal ordering of event occurrences we must supply S with an additional information; as such information we take here the dependency of events.

Dependency relation for net N is defined as the relation $D_N \subseteq E \times E$ such that

$$(a, b) \in D_N \Leftrightarrow \mathrm{Prox}_N(a) \cap \mathrm{Prox}_N(b) \neq \emptyset.$$

Intuitively speaking, two transitions are dependent if either they share a common entry place (then they "compete" for taking a token away from this place), or they share a common exit place (and then they compete for putting a token into the place), or an entry place of one transition is the exit place of the other (and then one of them "waits" for the other). Transitions are independent, if their neighbourhoods are disjoint. If both such transitions are enabled at a marking, they can be executed concurrently (independently of each other).

Define the non-sequential behaviour of net N as the trace language $[S]_D$ and denote it by B_N. That is, the non-sequential behaviour of a net arises from its sequential behaviour by identifying some execution sequences; as it follows from the dependency definition, two such sequences are identified if they differ in the order of independent transitions execution.

Sequential and non-sequential behaviour consistency is guaranteed by the following proposition (where the equality of values of two partial functions means that either both of them are defined and then their values are equal, or both of them are undefined).

Proposition 1.8.1 *For any net N the behavioural function β is congruent w.r. to D, i.e. for any strings $u, v \in E^*, u \equiv_D v$ implies $\beta(u) = \beta(v)$.*

Proof: Let $u \equiv_D v$; it suffices to consider the case $u = xaby$ and $v = xbay$, for independent transitions a, b and arbitrary $x, y \in E^*$. Because of the behavioural function definition, we have to prove only

$$\delta(\delta(m, a), b) = \delta(\delta(m, b), a)$$

for an arbitrary marking m. By simple inspection we conclude that, because of independency of a and b, i.e. because of disjointness of neighbourhoods of a and b, both sides of the above condition are equal to a configuration

$$(m - (\mathrm{Pre}\,(a) \cup \mathrm{Pre}\,(b))) \cup (\mathrm{Post}\,(a) \cup \mathrm{Post}\,(b))$$

or both of them are undefined. $\square$

This is another motivation for introducing traces for describing concurrent systems behaviour: from the point of view of reachability possibilities, all equivalent execution sequences are the same. As we have shown earlier, firing traces can be viewed as partially ordered sets of symbol occurrences; two occurrences are not ordered, if they represent transitions that can be executed independently of each other. Thus, the ordering within firing traces is determined by mutual dependencies of events and then it reflects the causal relationship between event occurrences rather than the non - objective ordering following from the string representation of net activity.

Two traces are said to be *consistent*, if both of them are prefixes of a common trace. A trace language T is *complete*, if any two consistent traces in T are prefixes of a trace in T.

Proposition 1.8.2 *The trace behaviour of any net is a complete trace language.*

Proof: Let B be the trace behaviour of net N, I be the independence induced by D, and let $[w], [u] \in B$, and let $[wx] = [uy]$ for some strings x, y. Then by Levi Lemma for traces there are strings v_1, v_2, v_3, v_4 such that $v_1v_2 \equiv w, v_3v_4 \equiv x, v_1v_3 \equiv u, v_2v_4 = y$ and $(v_2, v_3) \in I$. Let $m = \beta(v_1)$. Since $w \in Dom(\beta), v_1v_2 \in Dom(\beta)$; since $u \in \beta, (v_1v_3) \in Dom(\beta)$; hence, $\delta^*(m, v_2)$ is defined and $\delta^*(m, v_3)$ is defined; since $(v_2, v_3) \in I$, $v_1v_2v_3 \equiv v_1v_3v_2$; moreover, $\delta^*(m, v_2v_3)$ is defined, $\delta^*(m, v_3v_2)$ is defined, and $\delta^*(m, v_2v_3) = \delta^*(m, v_3v_2)$. It means that $v_1v_2v_3 \equiv v_1v_3v_2 \in Dom(\beta)$, i.e. $wv_3 \equiv uv_2 \in Dom(\beta)$. It ends the proof. $\square$

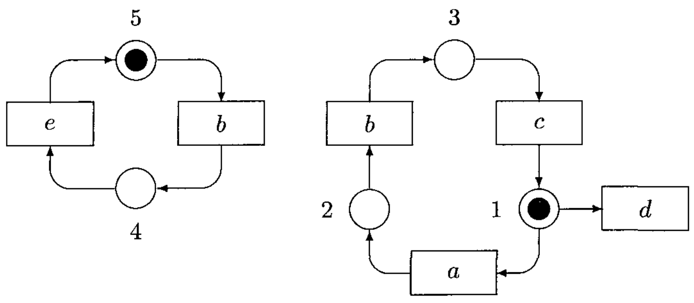

Figure 1.10: Decomposition of a net (into two sequential components).

Composition of nets and their behaviours. Here we are going to present a modular way of finding the behaviour of nets. It will be shown how to construct the behaviour of a net from behaviours of its parts: The first step is to decompose a given net into modules, next, to find their behaviours, and finally, to put them together finding in this way the behaviour of the original net.

Let $I = \{1, 2, \ldots, n\}$; we say that a net $N = (P, E, F, m)$ is composed of nets $N_i = (P_i, E_i, F_i, m_i), i \in I$, and write

$$N = N_1 + N_2 + \cdots + N_n,$$

if $i \neq j$ implies $P_i \cap P_j = \emptyset$ (N_i are pairwise place-disjoint), and

$$P = \bigcup_{i=1}^{n} P_i, E = \bigcup_{i=1}^{n} E_i, F = \bigcup_{i=1}^{n} F_i, m = \bigcup_{i=1}^{n} m_i.$$

The net in Fig. 1.9 is composed from two nets presented in Fig. 1.10.

Composition of nets could be defined without assuming the disjointness of sets of places; instead, we could use the disjoint union of sets of places rather than the set-theoretical union in the composition definition. However, it would require a new definition of nets, with two nets considered identical if there is an isomorphism identifying nets with different sets of places and preserving the flow relation and the initial marking. For sake of simplicity, we assume sets of places of the composition components to be disjoint, and we use the usual set-theoretical union in our definition.

Proposition 1.8.3 *Composition of nets is associative and commutative.*

Proof: Obvious. □

Let $I = \{1, 2, \ldots, n\}$ and let $N = N = N_1 + N_2 + \cdots + N_n, N_i = (P_i, E_i, F_i, m_i^0)$ for all $i \in I$. Let $\text{Pre}_{N_i}, \text{Post}_{N_i}, \text{Prox}_{N_i}$ be denoted by $\text{Pre}_i, \text{Post}_i, \text{Prox}_i$; moreover, let π_i be the projection functions from E onto E_i. Finally, let $m_i = m \cap P_i$ for any $m \subseteq P$.

Proposition 1.8.4 *Let $Q, R \subseteq P$ and set $Q_i = Q \cap P_I, R_i = R \cap P_I$. If $P = \bigcup_{i \in I} P_i$ and the members of the family $\{P_i\}_{i \in I}$ are pairwise disjoint, then*

$$\begin{aligned} Q \subseteq m &\Leftrightarrow \forall i \in I : Q_i \subseteq m_i, \\ R = Q \cap m &\Leftrightarrow \forall i \in I : R_i = Q_i \cap m_i, \\ m = Q \cup R &\Leftrightarrow \forall i \in I : m_i = Q_i \cup R_i, \\ R = m - Q &\Leftrightarrow \forall i \in I : R_i = m_i - Q_i. \end{aligned}$$

and $Q = \bigcup_{i \in I} Q_i, R = \bigcup_{i \in I} R_i$.

Proof: Clear. □

Proposition 1.8.5 $D_N = \bigcup_{i=1}^n D_{N_i}$.

Proof: Let $(a, b) \in D(N)$; by definition, $\text{Prox}(a) \cap \text{Prox}(b) \neq \emptyset$; by (1.8.4) $\text{Prox}(a) \cap \text{Prox}(b) = Q \Leftrightarrow \forall_{i \in I} \text{Prox}_i(a) \cap \text{Prox}_i(b) = Q_i$; hence, by Proposition 1.8.4, $(a, b) \in D \Leftrightarrow \text{Prox}(a) \cap \text{Prox}(b) \neq \emptyset \Leftrightarrow Q \neq \emptyset \Leftrightarrow \bigcup_{i=1}^n Q_i \neq \emptyset \Leftrightarrow \bigcup_{i=1}^n \text{Prox}_i(a) \cap \text{Prox}_i(b) \neq \emptyset \Leftrightarrow (a, b) \in \bigcup_{i=1}^n D_{N_i}$. □

Proposition 1.8.6 *Let δ_i denotes the transition function of N_i. Then*

$$\delta(m', a) = m'' \Leftrightarrow \forall i \in I : (a \in E_i \wedge \delta_i(m'_i, a) = m''_i \vee a \notin E_i \wedge m'_i = m''_i).$$

for all $m', m'' \in P$ and $a \in E$.

Proof: It is a direct consequence of Proposition 1.8.4. □

From the above proposition it follows by an easy induction:

$$\delta^*(m', w) = m'' \Leftrightarrow \forall i \in I : \delta_i^*(m'_i, \pi_i(w)) = m''_i. \tag{1.62}$$

The main theorem of this section is the following:

Theorem 1.8.7

$$B_{N_1 + N_2 + \cdots + N_n} = B_{N_1} \parallel B_{N_2} \parallel \cdots \parallel B_{N_n}.$$

Proof: Set $\eta(m) = \{w \in E^* \mid \delta^*(m^0, w) = m\}$ for each $m \subseteq P$ and $\eta_i(m) = \{w \in E_i^* \mid \delta_i^*(m_i^0, w) = m\}$ for each $m_i \subseteq P_i$; by Proposition 1.62 $\delta^*(m^0, w) = m \Leftrightarrow \forall_{i \in I} \delta_i^*(m_i^0, \pi_i(w)) = m_i$; thus, $w \in \eta(m) \Leftrightarrow \forall_{i \in I} : \pi_i(w) \in \eta_i(m_i)$. It means that

$$\eta(m) = \eta_1(m_1) \parallel \eta_2(m_2) \parallel \cdots \parallel \eta_n(m_n).$$

Thus, by definition of the behaviour, the proof is completed. □

This theorem allows us to find the behaviour of a net knowing behaviours of its components. One can expect the behaviour of components to be easier to find than that of the whole net. As an example let us find the behaviour of the net in Fig. 1.9 using its decomposition into two components, as in Fig. 1.10. These components are sequential; from the theory of sequential systems it is known that their behaviour can be described by the least solutions of equations:

$$\begin{aligned} X_1 &= X_1 be \cup b \cup \epsilon \\ X_2 &= X_2 abc \cup ab \cup a \cup d \cup \epsilon \end{aligned}$$

for languages X_1, X_2 over alphabets $\{b, e\}, \{a, b, c, d\}$, respectively. By definition of trace behaviour of nets, by Theorem 1.8.7, and Theorem 1.7.13 and we have

$$\begin{aligned} [X_1]_{D_1} &= [X_1]_{D_1}[be]_{D_1} \cup [b]_{D_1} \cup [\epsilon]_{D_1} \\ [X_2]_{D_2} &= [X_2]_{D_2}[abc]_{D_2} \cup [ab]_{D_2} \cup [a]_{D_2} \cup [d]_{D_2} \cup [\epsilon]_{D_2} \end{aligned}$$

with $D_1 = \{b, e\}^2, D_2 = \{a, b, c, d\}^2$. The equation for the synchronization $X_1 \parallel X_2$, is, by properties of synchronization, as follows:

$$[X_1]_{D_1} \parallel [X_2]_{D_2} =$$
$$([X_1]_{D_1}[be]_{D_1} \cup [b]_{D_1} \cup [\epsilon]_{D_1}) \parallel ([X_2]_{D_2}[abc]_{D_2} \cup [ab]_{D_2} \cup [a]_{D_2} \cup [d]_{D_2} \cup [\epsilon]_{D_2})$$

and after some easy calculations (using Proposition 1.7.5 we get

$$[X_1 \parallel X_2]_D =$$
$$[X_1 \parallel X_2]_D \cup [abc \cup abe \cup ab \cup a \cup d \cup \epsilon]_D$$

with $D = D_1 \cup D_2 = \{b, e\}^2 \cup \{a, b, c, d\}^2$. Observe that composition of two components that are sequential results in a net with some independent transitions; in the considered case, the independent pairs of transitions are $(a, e), (c, e), (d, e)$ (and their symmetric images).

There exists a standard set of very simple nets, namely nets with a singleton set of places, the behaviours of which can be considered as known. Such nets are called *atomic* or *atoms*; an arbitrary net can be composed from such nets; hence, the behaviour of any net can be composed from the behaviours of atoms. Thus, giving the behaviours of atoms, we supply another definition of the elementary net system behaviour. More formally, let $N = (P, E, F, m^0)$ be a net; for each $p \in P$ the net

$$N_p = (\{p\}, \mathrm{Prox}(p), F_p, (m^0)_p),$$

where

$$F_p = \{(e, p) \mid e \in \mathrm{Pre}(p)\} \cup \{(p, e) \mid e \in \mathrm{Post}(p)\}, (m^0)_p = m^0 \cap \{p\},$$

is called an *atom* of N (determined by p). Clearly, the following proposition holds:

Proposition 1.8.8 *Each net is the composition of all its atoms.*

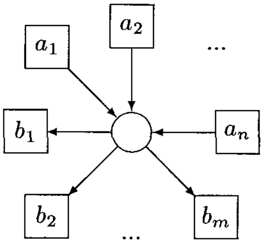

Figure 1.11: An example of atomic net

The behaviour of atoms can be easily found. Namely, let the atom N_0 be defined as $N_0 = (\{p\}, A \cup Z, F, m)$, where $A = \{e \mid (e,p) \in F\}, Z = \{e \mid (p,e) \in F\}$. Say that N_0 is marked, if $m = \{p\}$, and unmarked otherwise, i.e. if $m = \emptyset$. Then, by the definition of behavioural function, trace behaviour B_{N_0} is the trace language $[(ZA)^*(Z \cup \epsilon)]_D$, if it is marked, and $[(AZ)^*(A \cup \epsilon)]_D$, if it is unmarked, where $D = (A \cup Z)^2$.

1.9 Conclusions

Some basic notions of trace theory has been briefly presented. In the next chapters of this book this theory will be made broader and deeper; the intention of the present chapter was to show some initial ideas that motivated the whole enterprise. In the sequel the reader will be able to get acquainted with the development trace theory as a basis of non-standard logic as well as with the basic and involved results from the theory of monoids; with properties of graph representations of traces and with generalization of the notion of finite automaton that is consistent with the trace approach. All this work shows that the concurrency issue is still challenging and stimulating fruitful research.

Acknowledgements. The author is greatly indebted to Professor Grzegorz Rozenberg for his help and encouragement; without him this paper would not have appeared.

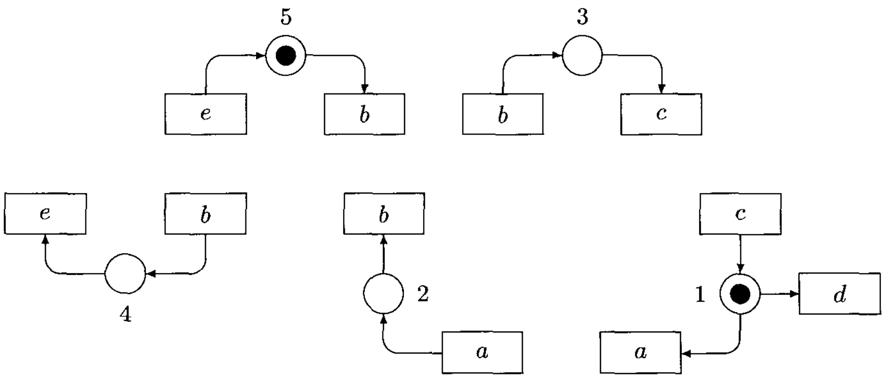

Figure 1.12: Atoms of net in Fig. 1.9