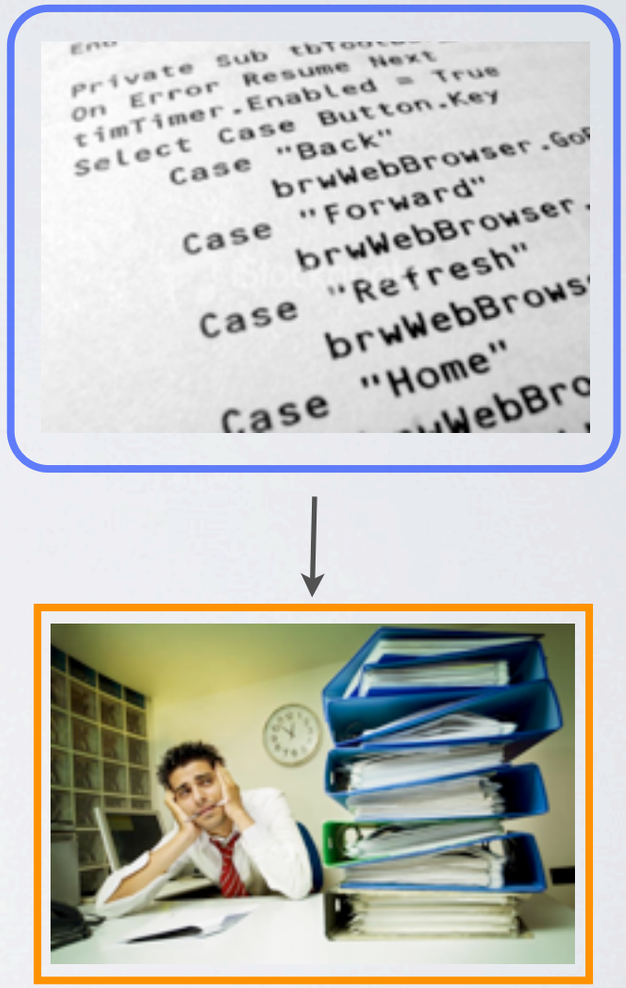
COMPUTER AIDED VERIFICATION

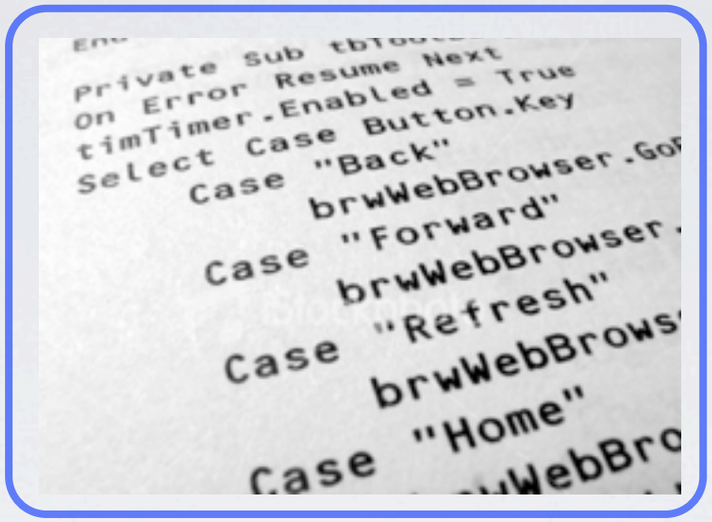
Lecture 11

Proving correctness of programs.
Java Modeling Language.



I. Proving correctness

PROVING CORRECTNESS



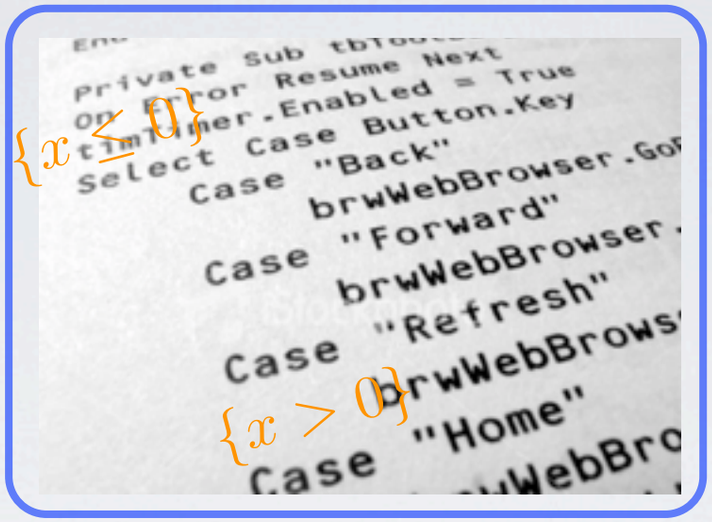
proof obligations

prover / proof assistant

proof

?

PROVING CORRECTNESS



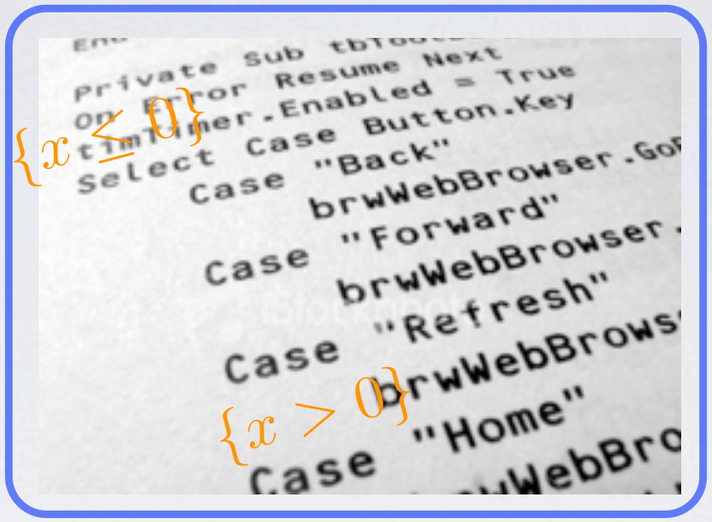
proof obligations

prover / proof assistant

proof

?

PROVING CORRECTNESS



proof obligations

automatic
or
interactive

prover / proof assistant

proof

?

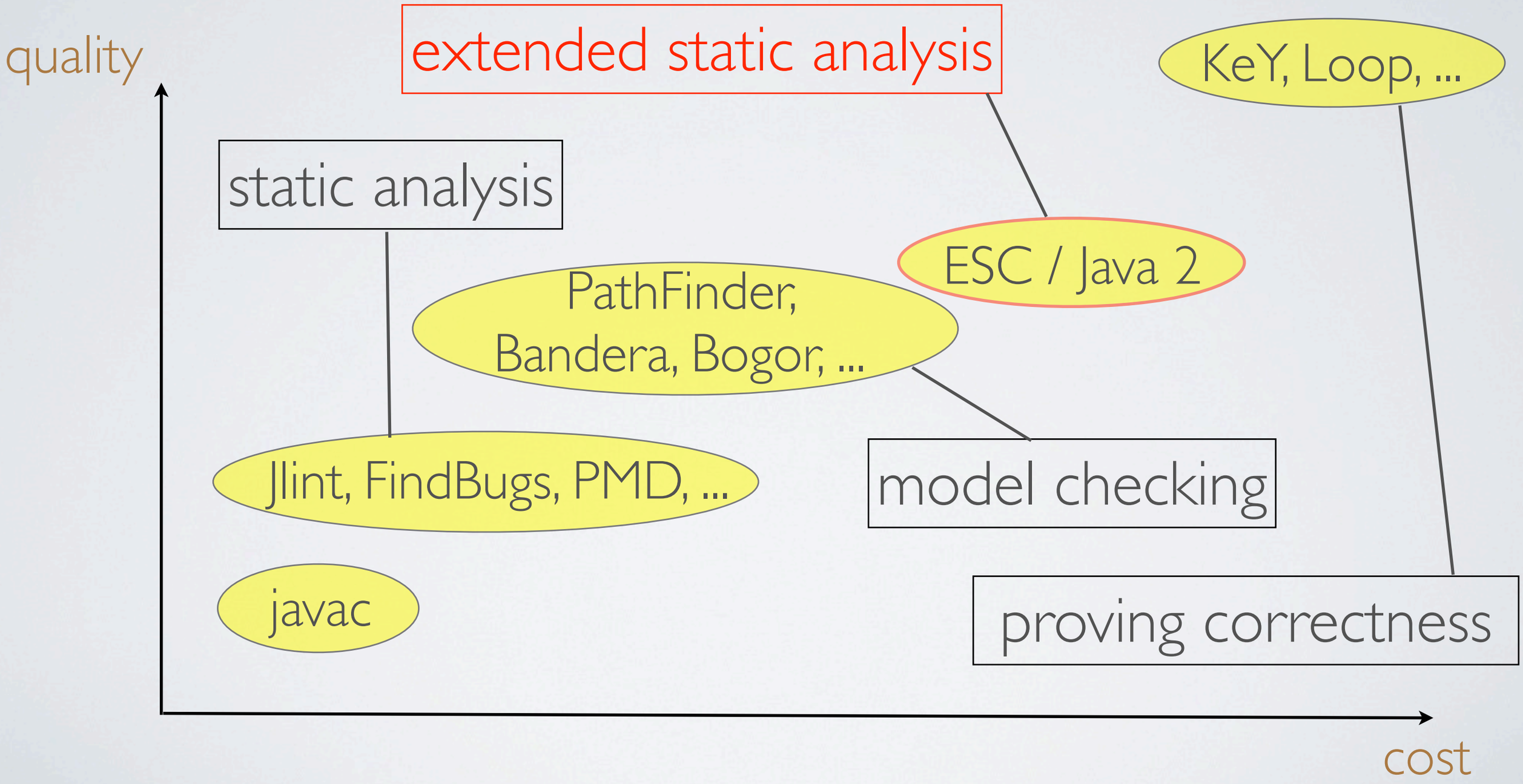
PROVING CORRECTNESS - PROPERTIES

- decorated source code is analyzed
- in general not fully automatic
- in general human assistance needed
- does not scale
- allows for parametrization / generalization

ESC / JAVA 2

- Prover Simplify
  - semantics of Java and JML in FOL (around 100 axioms)
- Proving correctness in compile time (static analysis)
- Neither complete nor correct
  - does not find all errors (false positives)
  - false alarms (false negatives)

EXTENDED STATIC ANALYSIS



ESC / JAVA 2

```
//@ requires x >= 0.0;
/*@ ensures JMLDouble
  @         .approximatelyEqualTo
  @          (x, \result * \result, eps);
  @*/
public static double sqrt(double x) {
  /*...*/
}
```

```
q1, q2, q3, q4 := Q1, Q2, Q3, Q4;
do q1 > q2 → q1, q2 := q2, q1
 [] q2 > q3 → q2, q3 := q3, q2
 [] q3 > q4 → q3, q4 := q4, q3
od    .
```

"base" conditions

(loop unfolding)

verification conditions (VS)

Simplify

proof

counterexample → warning

ESC / JAVA 2

```
//@ requires x >= 0.0;
/*@ ensures JMLDouble
  @         .approximatelyEqualTo
  @          (x, \result * \result, eps);
  @*/
public static double sqrt(double x) {
  /*...*/
}
```

```
q1, q2, q3, q4 := Q1, Q2, Q3, Q4;
do q1 > q2 → q1, q2 := q2, q1
 [] q2 > q3 → q2, q3 := q3, q2
 [] q3 > q4 → q3, q4 := q4, q3
od    .
```

"base" conditions

(loop unfolding)

Simplify

verification conditions (VS)

weakest precondition

proof

counterexample → warning

II. JML

LITERATURE

- G. T. Leavens, Y. Cheon, Design by Contract with JML
- P. Chalin, J. R. Kiniry, G. T. Leavens, E. Poll, Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2
- J. R. Kiniry, G. T. Leavens, E. Poll, A JML Tutorial: Modular Specification and Verification of Functional Behavior for Java, CAV 2007 Tutorial
- K. Rustan, M. Leino, G. Nelson, J.B. Saxe, ESC/Java User's Manual
- G. T. Leavens, C. Clifton, H. Rajan, Robby, Introduction to the Java Modelling Language, OOPSLA 2009 Tutorial

OVERVIEW

- formal specification (and documentation) language
- inspiration:
    - Hoare logic
        - Dijkstra's weakest preconditions
- easy for use by programmers (assertions)
- "lightweight" partial specifications
- DBC: design by contract

CONTRACT

```
import org.jmlspecs.models.JMLDouble;

public class SqrtExample {

    public final static double eps = 0.0001;

    //@ requires x >= 0.0;
    //@ ensures JMLDouble.approximatelyEqualTo(x, \result * \result, eps);
    public static double sqrt(double x) {
        return Math.sqrt(x);
    }
}
```

CONTRACT

```
import org.jmlspecs.models.JMLDouble;

public class SqrtExample {

    public final static double eps = 0.0001;

    //@ requires x >= 0.0;
    //@ ensures JMLDouble.approximatelyEqualTo(x, \result * \result, eps);
    public static double sqrt(double x) {
        return Math.sqrt(x);
    }
}
```

- pre-conditions

CONTRACT

```
import org.jmlspecs.models.JMLDouble;

public class SqrtExample {

    public final static double eps = 0.0001;

    //@ requires x >= 0.0;
    //@ ensures JMLDouble.approximatelyEqualTo(x, \result * \result, eps);
    public static double sqrt(double x) {
        return Math.sqrt(x);
    }
}
```

- pre-conditions
- post-conditions

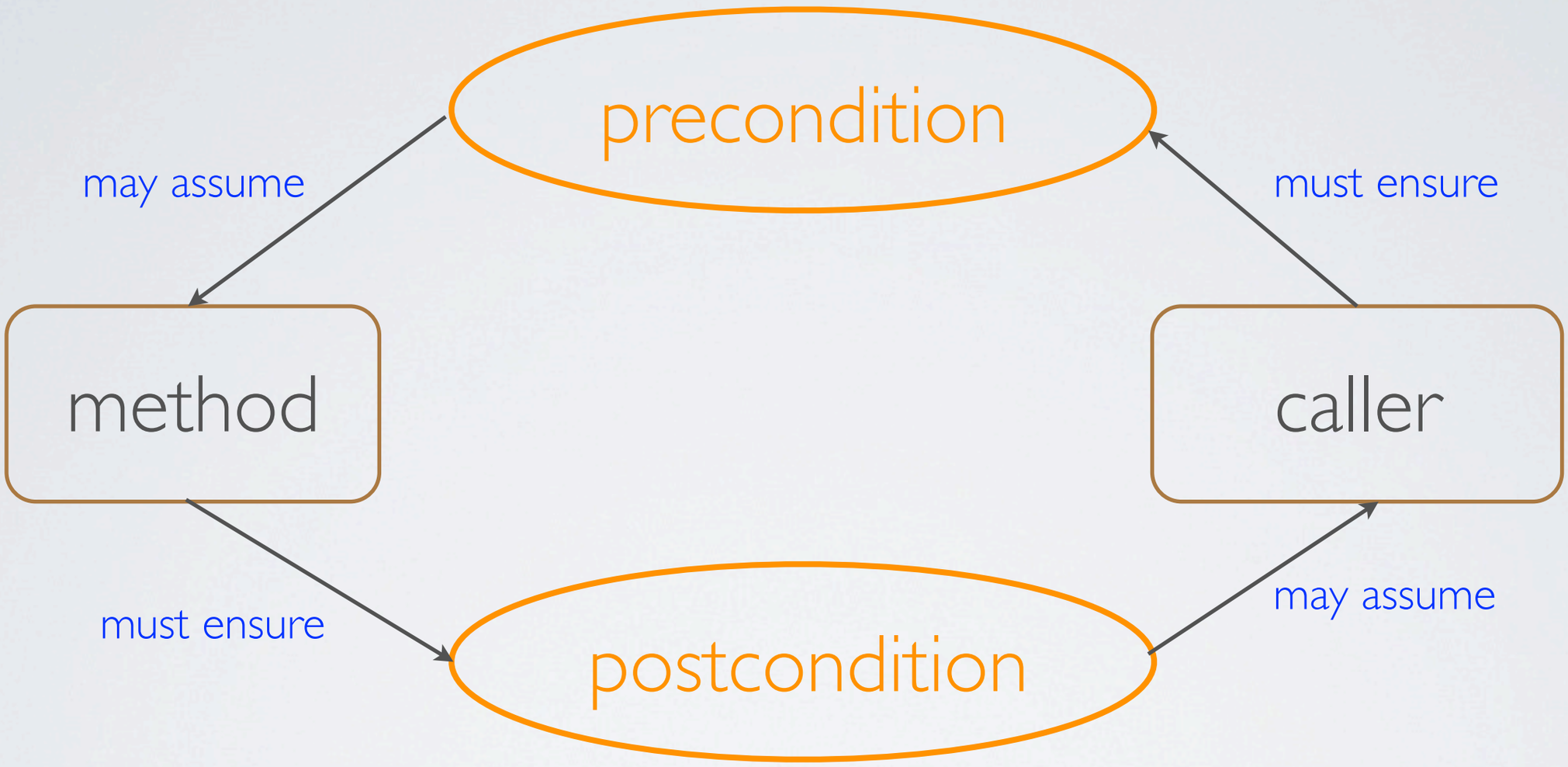
CONTRACT

```
import org.jmlspecs.models.JMLDouble;

public class SqrtExample {

    public final static double eps = 0.0001;

    //@ requires x >= 0.0;
    //@ ensures JMLDouble.approximatelyEqualTo(x, \result * \result, eps);
    public static double sqrt(double x) {
        return Math.sqrt(x);
    }
}
```

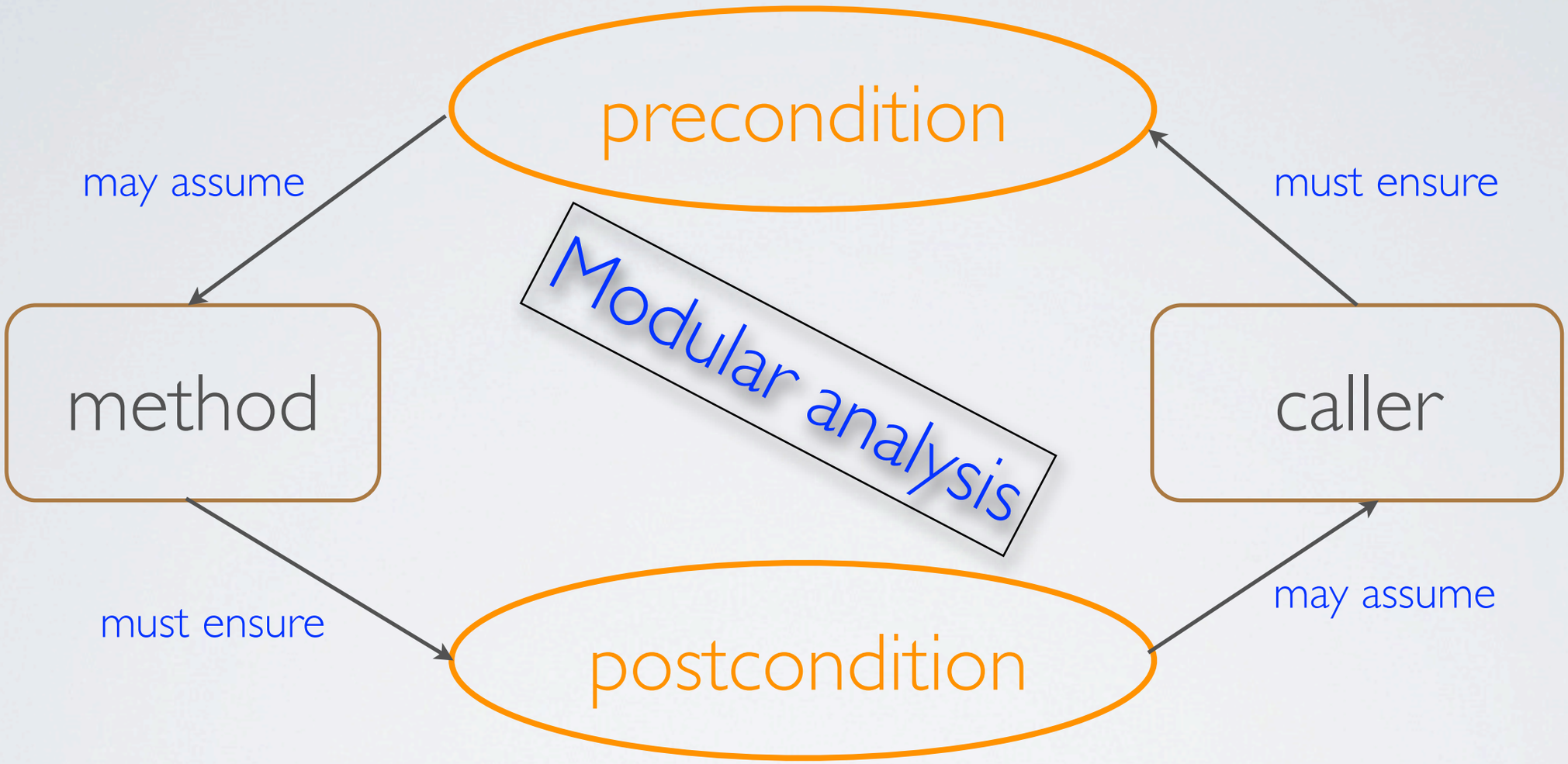
- pre-conditions
- post-conditions
- semantics: partial correctness

... (pages 37-39)

CONTRACT

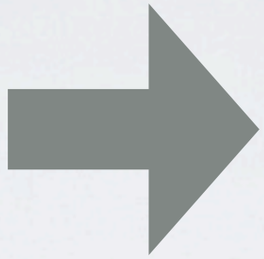

precondition
may assume
must ensure
method
caller
must ensure
may assume
postcondition

CONTRACT

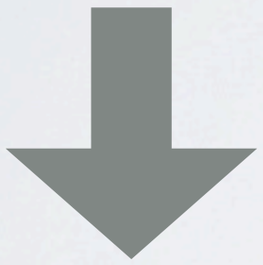

precondition
may assume
must ensure
method
caller
Modular analysis
must ensure
may assume
postcondition

CONTRACT

```
//@ requires P;
//@ ensures Q;
public void m() {
   S
}
```



```
//@ assert P;
o.m();
//@ assume Q;
```



```
public void m() {
   //@ assume P;
   S
   //@ assert Q;
}
```

MODULARITY

```
import org.jmlspecs.models.JMLDouble;

public class SqrtExample {

    public final static double eps = 0.0001;

    //@ requires x >= 0.0;
    //@ ensures JMLDouble.approximatelyEqualTo(x, \result * \result, eps);
    public static double sqrt(double x) {
        return Math.sqrt(x);
    }
}
```

```
//@ assert 9.0 >= 0.0;
double res = SqrtExample.sqrt(9.0);

/*@ assert JMLDouble
  @         .approximatelyEqualTo
  @          (9.0, res * res,
  @           SqrtExample.eps);
  @*/
```

MODULARITY

```
import org.jmlspecs.models.JMLDouble;

public class SqrtExample {

    public final static double eps = 0.0001;

    //@ requires x >= 0.0;
    //@ ensures JMLDouble.approximatelyEqualTo(x, \result * \result, eps);
    public static double sqrt(double x) {
        return Math.sqrt(x);
    }
}
```

static class!

```
//@ assert 9.0 >= 0.0;
double res = SqrtExample.sqrt(9.0);

/*@ assert JMLDouble
  @         .approximatelyEqualTo
  @          (9.0, res * res,
  @           SqrtExample.eps);
  @*/
```

ASSERT

- Java assert statement:
    - can have side effects
    - only Java expressions
- JML assert statement:
    - all JML features
    - no side effects

LOGIC

```
(\forall Student s;
  juniors.contains(s);
    s.getAdvisor() != null)
```

- && ==> ! ==
- quantification:
  - \forall \exists \sum \product \min \max \num_of
- \result \old( )
- natural language

```
/*@ also
  @ requires kgs >= 0;
  @ requires weight + kgs >= 0;
  @ ensures weight == \old(weight + kgs);
  @*/
public void addKgs(int kgs);
```

```
/*@ also
  @ ensures \result != null
  @   && (* \result is a displayable
  @       form of this person *);
  @*/
public String toString() {
  return "Person(\"" + name + "\","
    + weight + ")";
}
```

EXECUTABILITY

```
/*@ requires a != null
  @        && (\forall int i;
  @                0 < i && i < a.length;
  @                a[i-1] <= a[i]);
  @*/
int binarySearch(int[] a, int x) {
  // ...
}
```

```
(\forall Student s;
  juniors.contains(s);
    s.getAdvisor() != null)
```

EXECUTABILITY

```
/*@ requires a != null
  @        && (\forall int i;
  @                0 < i && i < a.length;
  @                a[i-1] <= a[i]);
  @*/
int binarySearch(int[] a, int x) {
  // ...
}
```

```
(\forall Student s;
  juniors.contains(s);
    s.getAdvisor() != null)
```

- dynamic analysis possible (in runtime)

EXECUTABILITY

```
/*@ requires a != null
  @        && (\forall int i;
  @                0 < i && i < a.length;
  @                a[i-1] <= a[i]);
  @*/
int binarySearch(int[] a, int x) {
  // ...
}
```

```
(\forall Student s;
  juniors.contains(s);
    s.getAdvisor() != null)
```

- dynamic analysis possible (in runtime)
- jmlc

POSTCONDITION

- normal: ensures
- exceptional: signals
  - signals_only

```
class SettableClock extends TickTockClock {

    // ...

    /*@ public normal_behavior
      @   requires   0 <= hour && hour <= 23 &&
      @              0 <= minute && minute <= 59;
      @   assignable _time_state;
      @   ensures    getHour() == hour &&
      @              getMinute() == minute && getSecond() == 0;
      @ also
      @  public exceptional_behavior
      @   requires     !(0 <= hour && hour <= 23 &&
      @                  0 <= minute && minute <= 59);
      @   assignable   \nothing;
      @   signals      (IllegalArgumentException e) true;
      @   signals_only IllegalArgumentException;
      @*/
    public void setTime(int hour, int minute) {
      if (!(0 <= hour & hour <= 23 & 0 <= minute & minute <= 59)) {
        throw new IllegalArgumentException();
      }
      this.hour = hour;
      this.minute = minute;
      this.second = 0;
    }
}
```

EXCEPTIONS

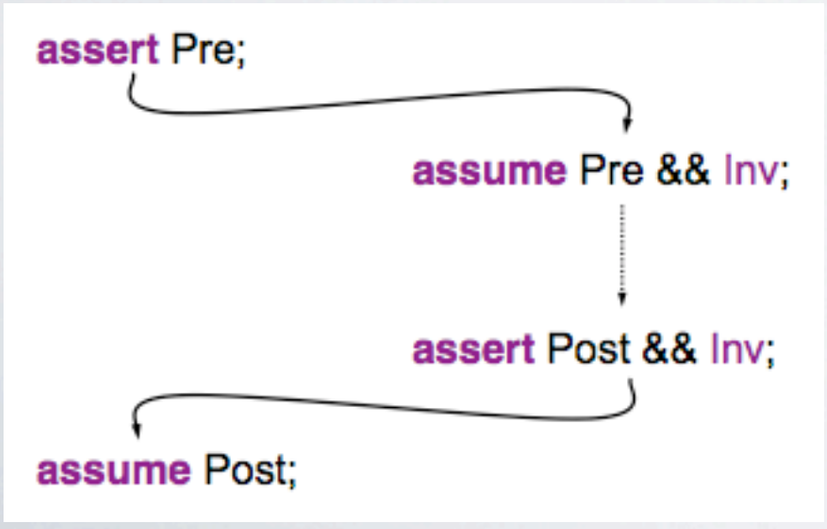
signals(Exception) false

ensures false

```
class SettableClock extends TickTockClock {

    // ...

    /*@ public normal_behavior
      @   requires   0 <= hour && hour <= 23 &&
      @              0 <= minute && minute <= 59;
      @   assignable _time_state;
      @   ensures    getHour() == hour &&
      @              getMinute() == minute && getSecond() == 0;
      @ also
      @  public exceptional_behavior
      @   requires     !(0 <= hour && hour <= 23 &&
      @                  0 <= minute && minute <= 59);
      @   assignable   \nothing;
      @   signals      (IllegalArgumentException e) true;
      @   signals_only IllegalArgumentException;
      @*/
    public void setTime(int hour, int minute) {
      if (!(0 <= hour & hour <= 23 & 0 <= minute & minute <= 59)) {
        throw new IllegalArgumentException();
      }
      this.hour = hour;
      this.minute = minute;
      this.second = 0;
    }
}
```

OBJECT INVARIANTS

- implicit in
    - pre- and postconditions (including exceptional ones) of all methods
    - postconditions of constructors
- temporarily may be unsatisfied
- improve understanding of code




```
public class Person {
  private /*@ spec_public non_null @*/
    String name;
  private /*@ spec_public @*/
    int weight;

  /*@ public invariant !name.equals("")
    @          && weight >= 0; @*/
```

OBJECT INVARIANTS

- invariants is much more than a useful shorthand
  - invariants are inherited
  - may depend on other objects!
  - must be satisfied in all observable states (except from helper methods)

```
public void tick() {
  second++;
  // object invariant might no longer hold
  canvas.paint();
  /* ... */
}
```

OBJECT INVARIANTS

- invariants is much more than a useful shorthand
  - invariants are inherited
  - may depend on other objects!
  - must be satisfied in all observable states (except from helper methods)

```
public void tick() {
  second++;
  // object invariant might no longer hold
  canvas.paint();
  /* ... */
}
```

entrance to / exit from any method

NON_NULL NULLABLE

- shorthand notation for
  - invariant

```
private /*@ non_null @*/ String fullName;
```

  - precondition

```
public /*@ pure @*/ AlarmClock(/*@ non_null @*/ AlarmInterface alarm) {
  this.alarm = alarm;
}
```

  - postcondition

```
//@ also
//@ ensures \result != null;
public String toString();
```

ASSIGNABLE, PURE

- assignable specifies side effects allowed
- pure = assignable \nothing

```
/*@ public normal_behavior
  @   requires   0 <= hour && hour <= 23 &&
  @              0 <= minute && minute <= 59;
  @   assignable _time_state;
  @   ensures    getHour() == hour &&
  @              getMinute() == minute && getSecond() == 0;
  @ also
  @  public exceptional_behavior
  @   requires     !(0 <= hour && hour <= 23 &&
  @                  0 <= minute && minute <= 59);
  @   assignable   \nothing;
  @   signals      (IllegalArgumentException e) true;
  @   signals_only IllegalArgumentException;
  @*/
```

```
//@ ensures _time == 12*60*60;
public /*@ pure @*/ Clock() { hour = 12; minute = 0; second = 0; }

//@ ensures 0 <= \result && \result <= 23;
public /*@ pure @*/ int getHour() { return hour; }
```

DEFAULTS

- invariant true
- requires true
- ensures true
- assignable \everything
- non_null

MULTIPLE SPECIFICATION CASES

```
private /*@ spec_public @*/ int age;

/*@    requires 0 <= a && a <= 150;
  @    assignable age;
  @    ensures age == a;
  @ also
  @    requires a < 0;
  @    assignable \nothing;
  @    ensures age == \old(age);
  @*/
public void setAge(int a)
{ if (0 <= a && a <= 150) { age = a; } }
```

... (pages 79-80)

„LIGHTWEIGHT" AND „HEAVYWEIGHT" SPECS

static analysis

dynamic analysis

```
class SettableClock extends TickTockClock {

    // ...

    /*@ public normal_behavior
      @   requires   0 <= hour && hour <= 23 &&
      @              0 <= minute && minute <= 59;
      @   assignable _time_state;
      @   ensures    getHour() == hour &&
      @              getMinute() == minute && getSecond() == 0;
      @ also
      @  public exceptional_behavior
      @   requires     !(0 <= hour && hour <= 23 &&
      @                  0 <= minute && minute <= 59);
      @   assignable   \nothing;
      @   signals      (IllegalArgumentException e) true;
      @   signals_only IllegalArgumentException;
      @*/
    public void setTime(int hour, int minute) {
      if (!(0 <= hour & hour <= 23 & 0 <= minute & minute <= 59)) {
        throw new IllegalArgumentException();
      }
      this.hour = hour;
      this.minute = minute;
      this.second = 0;
    }
}
```

INHERITANCE OF SPECIFICATIONS

- subclass inherits specification
- subclass invariant is stronger
- subclass precondition is weaker
- subclass postcondition is stronger

```
//@ assignable _time;
//@ ensures _time == \old(_time + 1) % 24*60*60;
public void tick() {
  second++;
  if (second == 60) { second = 0; minute++; }
  if (minute == 60) { minute = 0; hour++; }
  if (hour == 24)   { hour = 0; }
}
```

```
// spec inherited from superclass Clock
public void tick() {
  super.tick();
  if (getHour() == alarmHour & getMinute() == al
    alarm.on();
    //@ set _alarmRinging = true;
  }
  if ((getHour() == alarmHour & getMinute() ==
      (getHour() == alarmHour+1 & alarmMinute ==
    alarm.off();
    //@ set _alarmRinging = false;
  }
}
```

... (pages 168-170)

EXTENDING SPECIFICATION

```
import java.util.*;
public class Patient extends Person {
  protected /*@ spec_public @*/
     boolean ageDiscount = false;  //@ in age;

  /*@ also
    @    requires (0 <= a && a <= 150) || a < 0;
    @    ensures 65 <= age ==> ageDiscount;  @*/
  public void setAge(final int a) {
    super.setAge(a);
    if (65 <= age) { ageDiscount = true; }
  }
```

... (pages 173-174, 177-178)

VISIBILITY

```
//@ private invariant 0 <= alarmSecondsRemaining &&
//@                     alarmSecondsRemaining <= 60;

/*@ private invariant _alarmRinging
  @                   <==> alarmSecondsRemaining > 0; @*/
private int alarmSecondsRemaining = 0; //@ in _time;
```

```
private /*@ spec_public non_null @*/
  String name;
```

```
public class Person {
  private /*@ spec_public non_null @*/
    String name;
  private /*@ spec_public @*/
    int weight;

  /*@ public invariant !name.equals("")
    @          && weight >= 0; @*/
```

MODEL VARIABLES

- represent abstractly concrete variables
- non-modifiable
- change value along with concrete variables

```
public class Clock {
  //@ public model long _time;
  //@ private represents _time = second + minute*60 + hour*60*60;

  //@ public invariant _time == getSecond() + getMinute()*60 + getHour()*60*60;
  //@ public invariant 0 <= _time && _time < 24*60*60;

  //@ private invariant 0 <= hour && hour <= 23;
  private int hour; //@ in _time;
  //@ private invariant 0 <= minute && minute <= 59;
  private int minute; //@ in _time;
  //@ private invariant 0 <= second && second <= 59;
  private int second; //@ in _time;

  //@ ensures _time == 12*60*60;
  public /*@ pure @*/ Clock() { hour = 12; minute = 0; second = 0; }

  //@ ensures 0 <= \result && \result <= 23;
  public /*@ pure @*/ int getHour() { return hour; }

  //@ ensures 0 <= \result && \result <= 59;
  public /*@ pure @*/ int getMinute() { return minute; }

  //@ ensures 0 <= \result && \result <= 59;
  public /*@ pure @*/ int getSecond() { return second; }

  /*@ requires   0 <= hour && hour <= 23;
    @ requires   0 <= minute && minute <= 59;
    @ assignable _time;
    @ ensures    _time == hour*60*60 + minute*60;
    @*/
  public void setTime(int hour, int minute) {
    this.hour = hour; this.minute = minute; this.second = 0;
  }
```

GHOST VARIABLES

- exist only in specification
- modifiable (set)

```
/** The number of seconds remaining to keep ringing the alarm.
  * If zero, the alarm is silent (off). */
//@ private invariant 0 <= alarmSecondsRemaining &&
//@                     alarmSecondsRemaining <= 60;

/*@ private invariant _alarmRinging
  @                   <==> alarmSecondsRemaining > 0; @*/
private int alarmSecondsRemaining = 0; //@ in _time;
...

public boolean tick() {
  super.tick();
  if (alarmSecondsRemaining > 0) {
    alarmSecondsRemaining--;
    if (alarmSecondsRemaining == 0) {
      alarm.off();
      //@ set _alarmRinging = false;
    }
  } else if (getHour() == alarmHour &
             getMinute() == alarmMinute) {
           alarm.on();
           alarmSecondsRemaining = 60 - getSecond();
           //@ set _alarmRinging = true;
  }
}
```

AND OTHERS...

- loop invariants
- aliases
- data groups
- „avoiding" invariants: helper methods
- synchronization between threads
- ...

III. ESC / Java 2

PROPERTIES CHECKED

- exceptions:
    - NULL dereference
    - array index out of range, negative array index
    - no subclass cast
    - division by 0
- no runtime errors are checked, e.g., OutOfMemoryError

PROPERTIES CHECKED

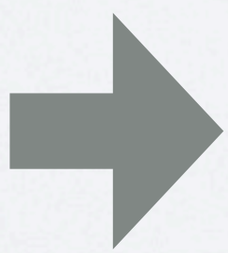
- methods:
    - partial correctness with respect to pre- and post-conditions and invariants
    - violation of assignable (pure)
    - undeclared exceptions
    - violation of non_null

PROPERTIES CHECKED

- loops
  - loop_invariant
  - decreases

Loops are unfolded requested number of times!

```
//@ loop_invariant E;
while (B) {
  S
}
```



```
//@ assert E;
if (!(B)) break;
S
```

ESC/Java User's Manual

PROPERTIES CHECKED

- control flow:
  - assert
  - reachable

PROPERTIES CHECKED

- classes - object invariants:
  - initially
  - invariant (only in observable states)

PROPERTIES CHECKED

- multi-threading
  - monitored
  - deadlock (e.g., causes by synchronized)
  - lack of synchronization / race conditions are not checked

DIAGNOSTIC INFORMATION

- not easily readable

- starting point: a formula classified by Simplify as false (satisfiable)

UNSOUNDNESS - FALSE POSITIVES

ESC / Java 2 is not able to prove presence of an error

UNSOUNDNESS - FALSE POSITIVES

- assume, nowarn
- loops - finite unfolding
- invariants of dynamically created objects ignored
- only chosen invariants taken into account (heuristics)
- lack of synchronization is not checked
- arithmetic overflow is ignored
- time upper bound of Simplify

INCOMPLETENESS - FALSE ALARMS

ESC / Java 2 is not able to prove absence of an error

INCOMPLETENESS - FALSE ALARMS

- Simplify is incomplete („potential'' counterexamples)
- Incomplete Java semantics:
  - floating-point arithmetic, bit operations, strings
  - arithmetic overflow is ignored

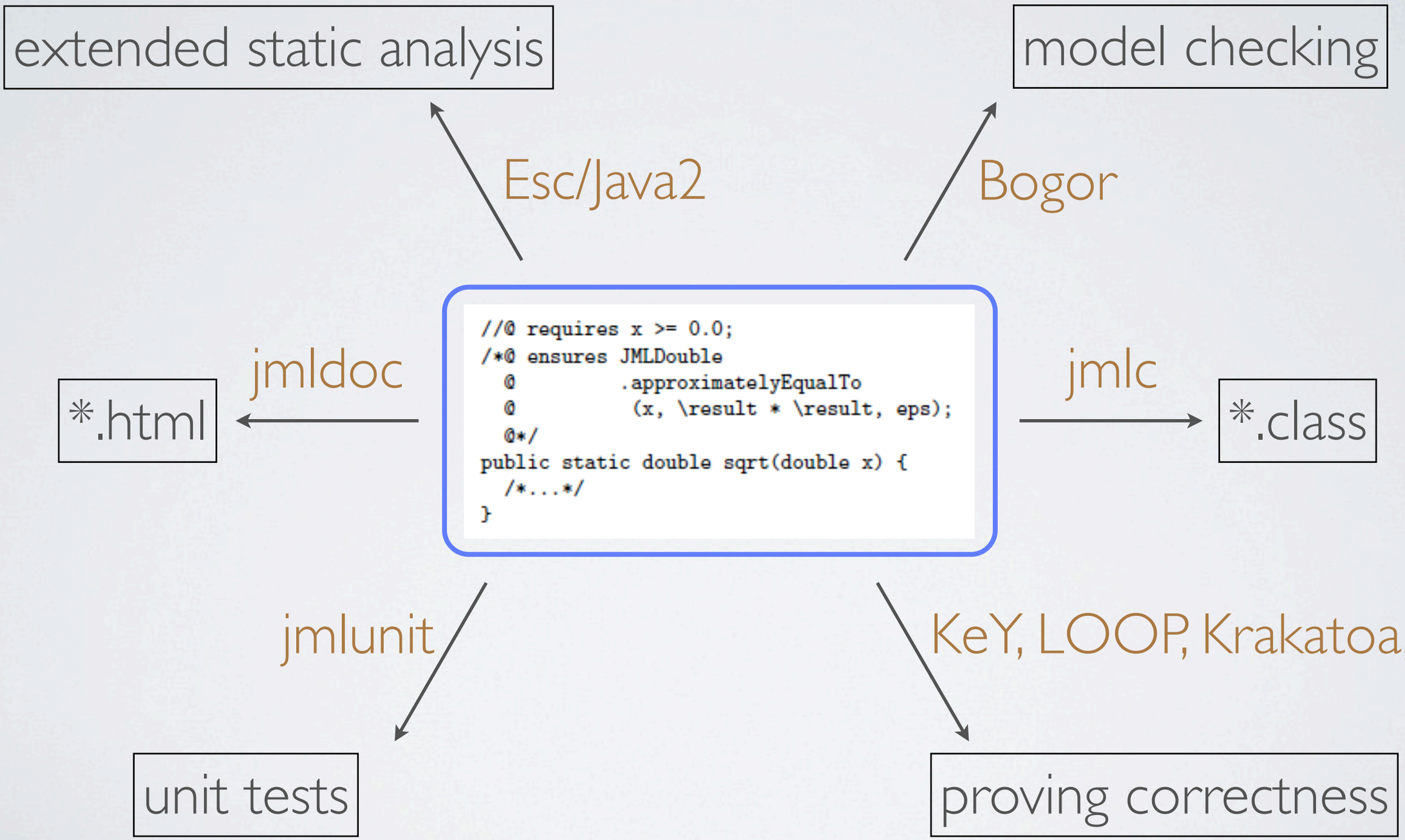
ESC / JAVA 2 - STRONG POINTS

- quickly finds many errors
- fully automatic
- may be used even without specification
- modularity
- diagnostic information
- integration with programming environments (Eclipse)

ESC / JAVA 2 - WEAK POINTS

- neither sound nor complete
- fairly complete specification is often necessary
- diagnostic information hard to understand
- only for Java 1.4
- insufficient documentation

TOOLS FOR JML



TOOLS FOR JML

extended static analysis

model checking

Esc/Java2

Bogor

jmldoc

*.html

```
//@ requires x >= 0.0;
/*@ ensures JMLDouble
  @          .approximatelyEqualTo
  @           (x, \result * \result, eps);
  @*/
public static double sqrt(double x) {
  /*...*/
}
```

jmlc

*.class

jmlunit

unit tests

Daikon

*.java

KeY, LOOP, Krakatoa, JACK

proving correctness