Introduction to JML

tool-supported specification for Java

Erik Poll

Radboud University Nijmegen

Overview

- The specification language JML
- Tools for JML, in particular
  - runtime assertion checking using jmlc and jmlrac
  - extended static checking using ESC/Java2

Java Modeling Language

JML by Gary Leavens et al.

Formal specification language for Java

- to specify behaviour of Java classes
- to record design/implementation decisions

by adding assertions to Java source code, eg

- preconditions
- postconditions
- invariants

as in Eiffel (Design-by-Contract), but much more expressive.

Goal: JML should be easy to use for any Java programmer.

JML

To make JML easy to use:

- **JML assertions are added as comments in .java file, between `/*@ ...@*/`, or after `//@`.**
- **Properties are specified in Java-like syntax: as Java boolean expressions, extended with a few operators.**

Pre- and postconditions

Pre- and post-conditions for methods, eg.

```
/*@ requires amount >= 0;
    ensures  balance == \old(balance)-amount &&
             \result == balance;
  @*/
 public int debit(int amount) {
   ...
 }
```

`\old(x)` is the value of `x` before execution of the method.

Pre- and postconditions

Specification is difficult! **(We can only hope for *incomplete* specs!)**
JML specs can be as weak as you want, eg.

```
/*@ requires amount >= 0;
    ensures  true;
  @*/
 public int debit(int amount) {
   ...
 }
```

Default postcondition "`ensures true`" can be omitted.
Idem for default precondition "`requires true`".

Design-by-Contract

Pre- and postconditions define a contract between a class and its clients:

- **Client must ensure precondition and may assume postcondition**
- **Method may assume precondition and must ensure postcondition**

Eg, in the example specs for `debit`, it is the obligation of the client to ensure that `amount` is positive.

Documenting preconditions – making these obligations explicit – can prevent problems due to lack of input validation.

Invariants

Invariants are properties that must be established by all constructors and maintained by all methods, eg

```
public class  BankAccount {
  public static final short MAX_BAL = 1000;
  private short balance;
   /*@ invariant 0 <= balance
                  && balance <= MAX_BAL;
     @*/
```

Intuitively: invariants included in all preconditions and all postconditions.

Invariants

- Invariants often document design decisions.
- Making them explicit helps in understanding the code.
- In larger programs, often the only interesting properties (to begin) to specify are invariants
- Invariants often lead to pre-conditions:

  Eg. in the `BankAccount` example, the precondition `amount <= balance` is needed to preserve the invariant `0 <= balance`

Exceptional postconditions

`signals` clauses specify when exceptions may be thrown

```
/*@ requires amount >= 0;
    ensures  true;
    signals (DebitException e)
                      amount > balance &&
                      balance == \old(balance);
  @*/
 public int debit(int amount) {
  ...
```

non_null

Common invariants, pre- and postconditions are about references not being `null`

```
public class Directory {

  private File[] files;
   //@ invariant files != null;

   //@ requires name != null;
  void createSubdir(String name){...}

   //@ ensures \result != null;
  Directory getParent(){...}
```

non_null

non_null is a convenient short-hand for these.

```
public class Directory {

  private /*@ non_null @*/ File[] files;

  void createSubdir(/*@ non_null @*/ String name){

  Directory /*@ non_null @*/ getParent(){...}
```

(Note that this is almost like typing.)

assert clauses

An `assert` clause specifies a property that should hold at some point in the code, eg.

```
if (i <= 0 || j < 0) {
      ...
  } else if (j < 5) {
      //@ assert i > 0 && 0 < j && j < 5;
      ...
  } else {
      //@ assert i > 0 && j > 5;
      ...
  }
```

`assert` now included in Java (since Java 1.4)

pure

Pure methods are methods without side-effects:

```
public /*@ pure @*/ int getBalance(){...
```

Pure methods – and only pure methods – can be used *in* JML specifications.

That's all, folks!

This covers what you need to know to start using JML!

Keywords:

> requires, ensures, invariant, signals, assert

Modifiers:

> non_null, pure

Operators:

> \result, \old, ==>, \forall

Tools for JML

Tools for checking JML

- parsing and typechecking and jmldoc
- runtime assertion checking: test for violations of assertions during execution, jmlrac (Iowa Univ.)
- verification: prove (at compile-time) that contracts are never violated
  - automatic verification of simple properties
    - extended static checker ESC/Java(2) (Compaq) (Kodak/Nijmegen)
    - JACK (Gemplus/INRIA)
  - interactive verification of complex properties
    - LOOP tool (Nijmegen)
    - Jive (Kaiserslautern+ETH Zürich), Krakatoa (INRIA)

Tools for checking JML

- model checking of multi-threaded Java code by Bogor (Kansas State Univ.)

Support for concurrency in JML is in its infancy!

Tools for writing JML

- **runtime detection of invariants** by **Daikon** (Ernst, MIT)
  postulates common assertions, then uses runtime asserting checking to check them.
- **static detection of invariants** by **Houdini** (Leino et. al., Compaq)
  postulates common assertions, then uses ESC/Java to check them.

Tools for JML

The most interesting tools for industrial use today:

- **runtime assertion checking with jmlc & jmlrac**
  execute code and test all assertions *for a given set of inputs*
- **extended static checking with ESC/Java(2)**
  verify aka prove that assertions are never violated, *for all possible inputs*

Runtime assertion checking with jmlc and jmlrac

jmlc & jmlrac

- **jmlc compiles `.java` file with JML assertions to `.class` file with runtime checks**
- **jmlrac can execute these `.class` files**

  during execution, *all* assertions are then tested and any violation of an assertion produces an Error

  Of course, an assertion violation can be error in code *or* error in specification . . .

- **jmlunit combines jmlc/jmlrac with jmlunit.**

jmlrac vs conventional testing

- Better testing, because more properties are tested, at more places in the code

  Eg. "Invariant violated in line 8000" after 1 minute instead of "NullPointerException in line 2000" after 4 minutes

  Information about cause of problem, rather than effect.

jmlrac vs conventional testing

- Some testcode generated automatically by `jmlc`.

  Eg for

```
/*@ ...
      signals (Exception)
                balance == \old(balance);
  @*/
 public int debit(int amount) { ...
```

jmlrac will test that if `debit` throws an exception, the balance hasn't changed, and all invariants still hold.

jmlrac checks `\forall` if the domain of quantification is finite

jmlrac vs conventional testing

- **Less work to come up with test scenarios**

  With rich specs, test scenarios only have to provide inputs but *not* the expected outputs.

ESC/Java(2) :
Extended static checking

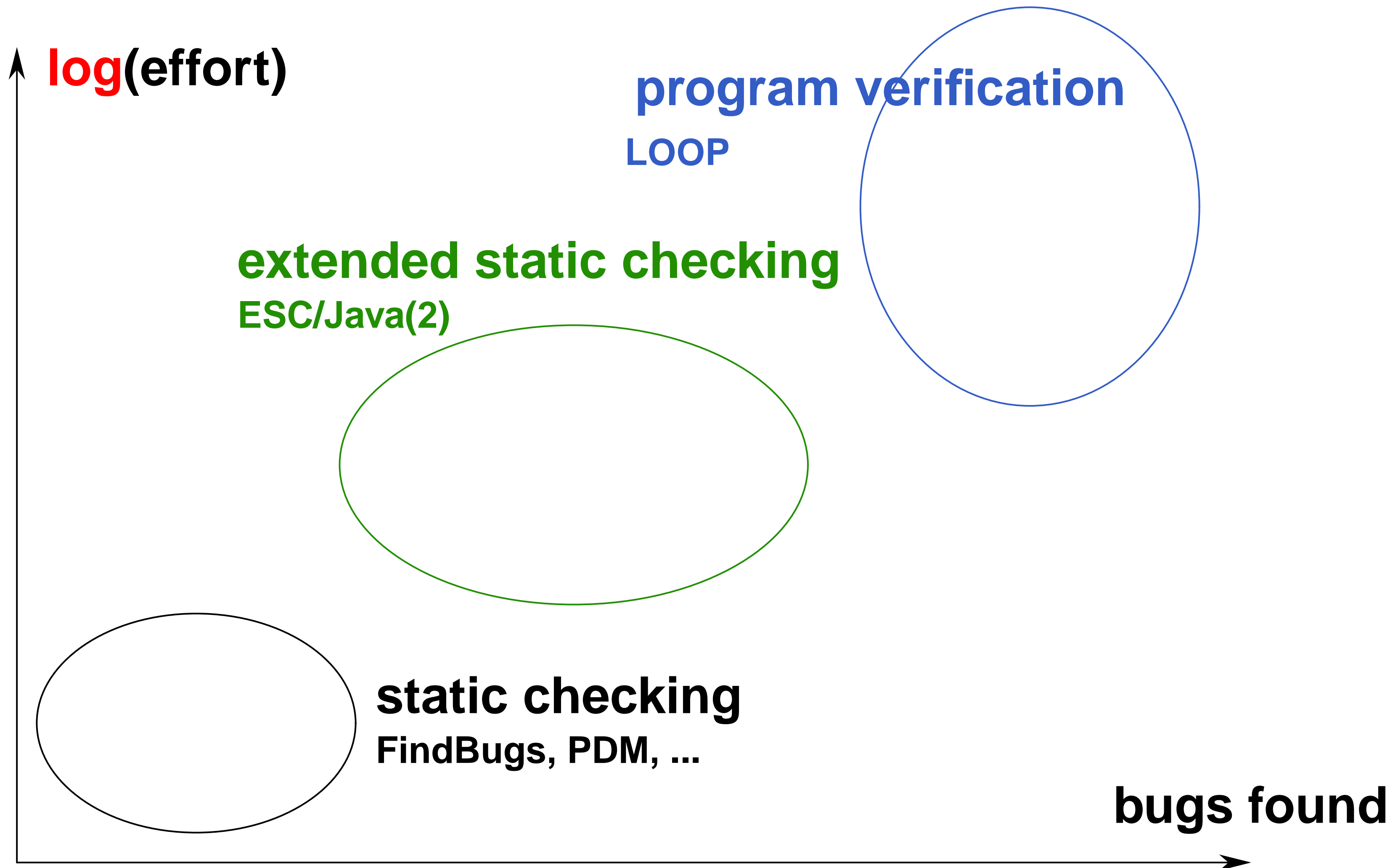
ESC/Java(2)

ESC/Java by Rustan Leino et. al. at DEC/Compaq/HP

ESC/Java2 by David Cok (Kodak) and Joe Kiniry (Nijmegen)

- **extended static checking = ‘lightweight’ program verification:**
  ESC/Java *tries* to *prove* correctness of specifications, at compile-time, fully automatically
- **User never has to interact with the back-end theorem prover (Simplify)**
- ***not sound*, *not complete*, but aim is to find lots of potential bugs quickly**

What is extended static checking?



ESC/Java is not complete

ESC/Java may produce warnings about correct programs.

```
/*@ requires 0 < n;
  @ ensures \result ==
  @                (\exists int x,y,z;
  @                  pow(x,n)+pow(y,n) == pow(z,n));
  @*/
public static boolean fermat(double n) {
  return (n==2);
}
```

Warning: ***postcondition possibly not satisfied***
(Typically, the theorem prover times out in complicated cases.)

ESC/Java is not sound

ESC/Java may fail to produce warning about incorrect program.

```
public class Positive{
    private int n = 1;  //@ invariant n > 0;

    public void increase(){ n++; }
}
```

ESC/Java(2) produces no warning, but `increase` may break the invariant, namely if `n` is $2^{31} - 1$.

> **More fundamental challenge: sound modular verification for OO programs with invariants.**

ESC/Java *is* useful

The goal is finding bugs.

Unsoundness and incompleteness just means we cannot find all bugs, and we have some false positives.

ESC/Java(2) is good at proving absence of runtime exceptions (eg Null-, ArrayIndexOutOfBounds-, ClassCast-) and verifying relatively simple properties.

Using ESC/Java(2)

The first big problem in any attempt at program verification:

what do we want to verify anyway?

Good places to start:

1. **prove no runtime exceptions can occur**
2. **verify additional crucial invariants**

In fact, verification of 1. will require many invariants to be specified.

Using ESC/Java(2)

Cost of using ESC/Java is proportional to

- size of code,
  as code needs to be annotated
- size of API used,
  as API methods used may have to be specified

We (and the OOTI students here at TU/e) succesfully applied ESC/Java(2) to Java Card smart card applets (1000's of loc).

`assignable` clauses

(Modular) program verification requires `assignable` aka `modifies` clauses, expressing frame properties.

Eg.

```
/*@   requires amount >= 0;
    assignable balance;
      ensures balance == \old(balance)-amount;
 @*/
public int debit(int amount) { ... }
```

ie. `debit` can *only* assign to the field `balance`.

NB this does *not* follow from the post-condition.

Conclusions

Potential strong points of JML

- Easy to learn: syntax & semantics very close to Java
- No need for a formal model:

  the Java source code *is* the formal model

  Consequently:
  - use of JML can be introduced gradually
  - JML can be used for existing (legacy) code & APIs

  But:
  - JML does not *provide* – or *impose* – any design methodology, as UML, B, VDM, . . . do

Tools for JML

Runtime assertion checking

- low cost & effort
- easy to do as part of normal testing

Extended checking with ESC/Java

- much higher effort
- higher assurance: independent of any test suite
- verifying a property *forces* you to specify all the invariants and API contracts that it relies on

Related Work

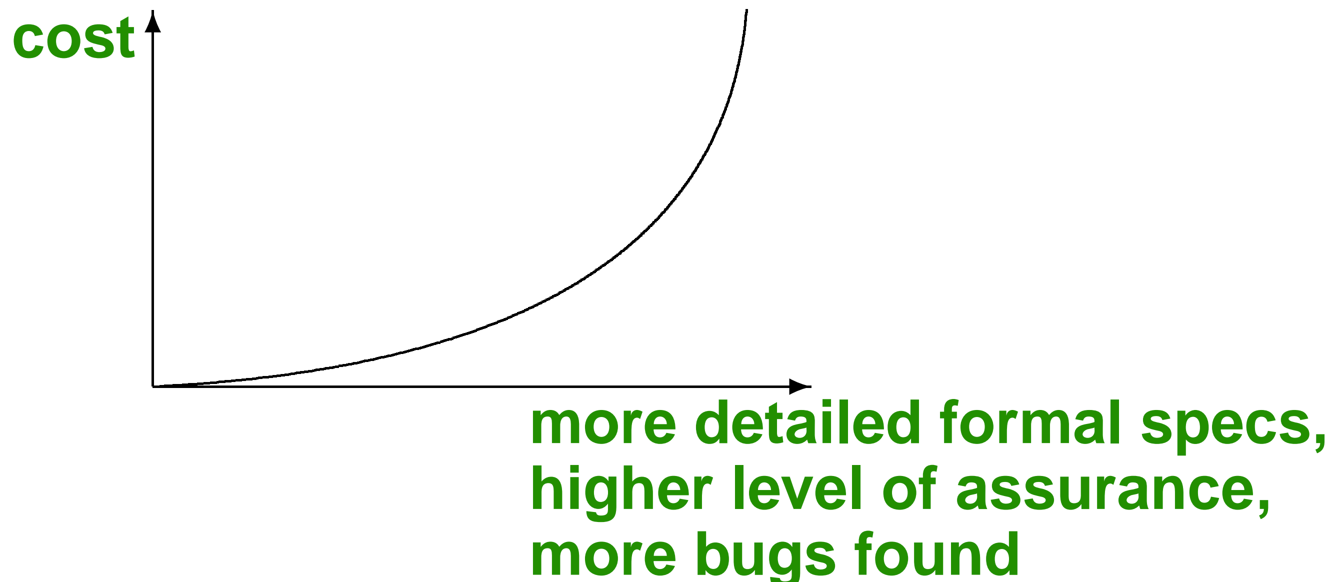
- Spec# for C# by Microsoft Research
- SparkAda by Praxis Critical Systems Ltd.
- jContract tool for Java by Parasoft
- OCL specification language for UML

The challenge in using JML

Choosing

- which properties to specify
- up to which level of detail
- using which tools

so that JML specs are cost-effective.



Pointers

More info about JML: www.jmlspecs.org

Some papers

- **gentle intro**
  Design by Contract with JML
  by Gary Leavens and Yoonsik Cheon
- **white paper**
  An overview of JML tools and applications
  by lots of people