Advances in Programming Languages

APL14: Practical tools for Java Correctness

David Aspinall
(slides originally by Ian Stark)

School of Informatics
The University of Edinburgh

Friday 12 November 2010
Semester 1 Week 8

Topic: Augmented Languages for Correctness

This block of lectures covers some language techniques and tools for improving program quality:

- Augmentations and Certifying Correctness
- Assertions and Hoare Logic
- Practical tools for Java Correctness

This lecture introduces some practical tools for helping establish correctness properties of Java programs.

Outline

Outline

Model-based specification

Modeling (sic) is an abstraction technique for system design and specification.

A *model* is a representation of the desired system.

A *formal model* is one that has a precise description in a formal language.

A model differs from an implementation in that it might:

- capture only some aspects of the system (e.g., interfaces);
- be partial, leaving some parts unspecified;
- not be executable.

An implementation of the system can be compared to the model.

Sometimes the model is iteratively refined to give the implementation.

Sample applications of modeling in computer software development:

- **VDM** the *Vienna Development Method*.
- **B** the *B language* and *B method*.
- **Extended ML** the extension of Standard ML with specifications.
- **OCL** the *Object Constraint Language* extension of UML.

The Java Modeling Language

The *Java Modeling Language*, JML, combines model-based and contract approaches to specification.

Some design features:

The specification lives close to the code
: Within the Java source, in *annotation comments* /*@...@*/

Uses Java syntax and expressions
: Rather than a separate specification language.

Common language for many tools and analysis
: Tools add their own extensions, and ignore those of others.

Web site: jmlspecs.org

JML: basics

```
public class Account {
  public int credit;

  /*@ requires credit > amount && amount > 0;
    @ ensures credit > 0 && credit == \old(credit) − amount;
    @*/
  public int withdraw(int amount) {
    ...
  }
}
```

JML conditions combine logical formulae (&&,==) with Java expressions (credit, amount). Expressions must be *pure*: no side-effects.

There are also visibility controls, glossed over in these examples: `credit` ought not to be public!

JML: exceptions

```
public class Account {
  public int credit;

  /*@ requires credit > amount && amount > 0;
    @ ensures credit > 0 && credit == \old(credit) − amount;
    @ signals (RefusedException) credit == \old(credit);
    @*/
  public int withdraw throws RefusedException (int amount) {
    ...
  }
}
```

Where **ensures** speaks about normal termination, **signals** specifies properties of the state after exceptional termination.

JML: logical formulae

```
public class IntArray {
  public int[] contents;

  /*@ requires (\forall int i,j;
    @                 0<i && i<j && j<contents.length;
    @                 contents[i] <= contents[j]);
    @
    @ ensures contents[\result] == value || \result == -1;
    @*/
  public int search (int value) { ... }
}
```

The search routine requires that array contents be sorted on entry. This would, for example, be necessary if it used binary chop to locate value.

JML: class invariants

```
public class IntArray {
  public int[] contents;

  /*@ invariant (\forall int i,j;
    @                    0<i && i<j && j<contents.length;
    @                    contents[i] <= contents[j]);
    @*/

  /*@ ensures contents[\result] == value || \result == -1
    @*/
  public int search (int value) { ... }
}
```

Now contents must be sorted whenever it is visible to clients of IntArray.

JML: assumptions and assertions

```
/*@ assume j*j < contents.length @*/
contents[j*j] = j;

...

a[0] = complexcomputation(a,v);
/*@ assert (\forall int i; 1<i && i<10; a[0] < a[i]) @*/
```

An *assumption* may help a static analysis tool.

An *assertion* must always be checked — similarly to Java's runtime **assert**.

JML: models and ghosts

```
public class IntArray {
  public int[] contents;

  /*@ model int total;
    @ represents total = arraySum(contents)
    @*/

  /*@ ghost int cursor;
    @ set cursor = contents.length / 2
    @*/
  ...
}
```

A *model* field represents some property of the model that does not appear explicitly in the implementation.

A *ghost* field is a local variable used only by other parts of the specification.

JML: model methods and classes

```
/*@ ensures \result = (\sum int i; 0<i && i<a.length; a[i])
  @
  @ public model int arraySum(int[] a);
  @*/

/*@ public model class JMLSet { ... } @*/
```

Specifications may refer to *model methods* and even entire *model classes* to represent and manipulate desired system properties.

JML provides specifications for the standard Java classes, as well as a library of model classes for mathematical constructions like sets, bags, integers and reals (i.e. of arbitrary size and precision).

Dynamic JML tools: running and testing

JML annotations can be used to drive various runtime checks.

- **jmlc** is a compiler which inserts runtime tests for every assertion; if an assertion fails, an error message provides static and dynamic information about the failure.
- **jmlunit** creates test classes for JUnit based on preconditions, postconditions and invariants. These automatically exercise and test assertions made in the code.

JML annotations also provide formal documentation:

- **jmldoc** generates human-readable web pages from JML specifications, extending the existing javadoc tool.

Outline

JML tools: static analysis

- The *ESC/Java 2* framework carries out a range of static checks on Java programs. These include formal verification of JML annotations using a fully-automated theorem prover.

  Controversially, the checker is neither sound nor complete: it warns about many potential bugs, but not all actual bugs.

  This is by design: the aim is to find many possible bugs, quickly.

- The *LOOP* tool also attempts to verify JML specifications. Some can be done automatically; where this is not possible it provides *proof obligations* for the interactive PVS theorem prover.

- The *JACK* tool generates proof obligations from JML annotations on Java and JavaCard programs; these can then be tackled with a variety of automatic and semi-automatic theorem provers.

ESC/Java2

> "The Extended Static Checker for Java version 2 (ESC/Java2) is a programming tool that attempts to find common run-time errors in JML-annotated Java programs by static analysis of the program code and its formal annotations."

http://kind.ucd.ie/products/opensource/ESCJava2

It is available both as a command-line tool and a plugin for the *Eclipse* development environment.

ESC/Java performs different kinds of **static** check:

- checks based on types, flow of data, existing Java declarations;
- JML annotation checking that can be carried out directly;
- logical assertions that need an external proof tool.

These last ones are passed to the *Simplify* automated theorem prover.

Recent versions of ESC/Java also support other provers.

History

ESC/Modula-3 DEC Systems Research Center (SRC) 1991–1996

ESC/Java Compaq SRC, then Hewlett-Packard 1997–2002

ESC/Java2 University of Nijmegen, University College Dublin 2004–2009

emerging JML+ESC successors
University of Central Florida,
Kansas State University,
Concordia Unversity, ...

K. Rustan M. Leino. Extended Static Checking: A Ten-Year Perspective in *Informatics: 10 Years Back, 10 Years Ahead*. Lecture Notes in Computer Science 2000, Springer.

Many different checks

ESC/Java2 checks for very many things. These include:

- Null pointer dereference
- Negative array index
- Array index too large
- Invalid type casts
- Array storage type mismatch
- Divide by zero
- Negative array size
- Unreachable code
- Deadlock in concurrent code
- Race condition
- Unchecked exception
- Object invariant broken
- Loop invariant broken
- Precondition not satisfied
- Postcondition not satisfied
- Assertion not satisfied

JML assumptions and assertions can help with all of these.

Soundness and Completeness

As a practical tool ESC/Java makes some compromises: it is not perfect.

- Not sound: it may approve an incorrect program.
- Not complete: it may complain about a correct program.

However, it reliably checks straightforward specifications, and automatically points out many potential bugs.

In particular:

- Distinguishes between *errors* (definitely bad), *warnings* (could be bad) and *cautions* (can't be sure it's good).
- Sources of unsoundness and incompleteness are documented.

Soundness and Completeness

As a practical tool ESC/Java makes some compromises: it is not perfect.

- Not sound: it may approve an incorrect program.
- Not complete: it may complain about a correct program.

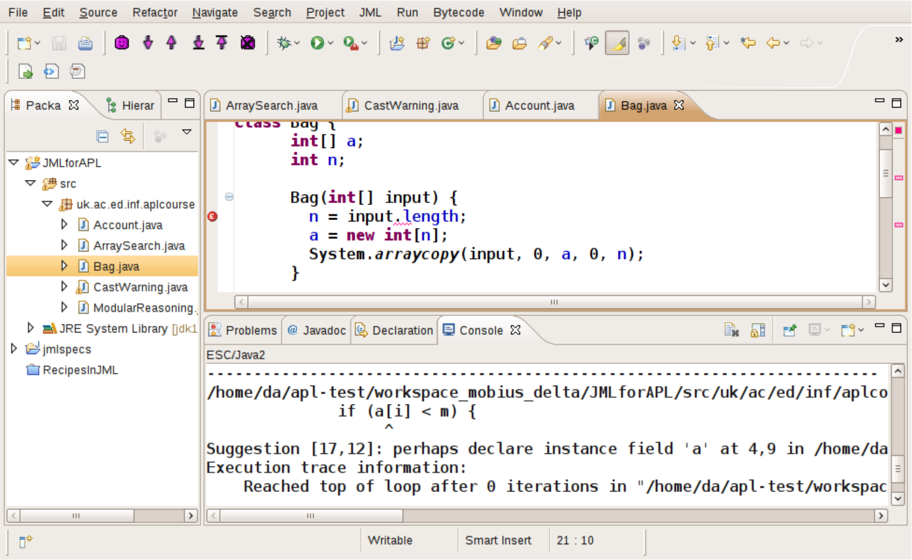
However, it reliably checks straightforward specifications, and automatically points out many potential bugs.

In particular:

- Distinguishes between *errors* (definitely bad), *warnings* (could be bad) and *cautions* (can't be sure it's good).
- Sources of unsoundness and incompleteness are documented.

> . . . as we know, there are "known knowns"; there are things we know we know. We also know there are "known unknowns"; that is to say we know there are some things we do not know. But there are also "unknown unknowns" — the ones we don't know we don't know.
>
> (Donald Rumsfeld, 2002)

ESC/Java2 in Eclipse



Alternatively: try the command line tools. Here is a pseudo-demo.

Common specification idioms: non null

JML and ESC/Java2 introduce keywords for common specifications.

One of the most common specification requirements in Java is that objects be non-null. That's because one of the most common Java programming errors is NullPointerException.

```
//@ non_null
Object o;
```

Now every method invocation on o is known to not cause an exception, *but* every assignment to o must be checked to be non-null.

This is so important that it is about to enter the Java language as an official annotation @NonNull, to be exploited by ordinary compilers.

Common specification idioms: non null

JML and ESC/Java2 introduce keywords for common specifications.

One of the most common specification requirements in Java is that objects be non-null. That's because one of the most common Java programming errors is NullPointerException.

```
//@ non_null
Object o;
```

Now every method invocation on o is known to not cause an exception, *but* every assignment to o must be checked to be non-null.

This is so important that it is about to enter the Java language as an official annotation @NonNull, to be exploited by ordinary compilers.

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965. [...] My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference
>
> (Tony Hoare, 2009)

Behavioural subtyping

Part of the object-oriented paradigm: an object in a subclass can **behave like** an object in a superclass.

Sometimes known as Liskov's *principle of substitutivity*:

> *properties that can be proved using the specification of an object's presumed type should hold even though the object is actually a subtype of that type* [Liskov and Wing, 1994]

This is captured by requiring, when A **extends** B

- each invariant in subclass A $\Longrightarrow$ an invariant in B.
- precondition for A.m $\Longleftarrow$ precondition for B.m
- postcondition for A.m $\Longrightarrow$ postcondition for B.m

Inherited specifications

Behavioural subtyping is ensured by *inherited specifications*. A child class automatically inherits the specification of its parent.

```
class Parent {
  //@ requires i >= 0;
  //@ ensures \result >= i;
  int m(int i){ ... }
}
class Child extends Parent {
  //@ also
  //@  requires i <= 0
  //@  ensures \result <= i;
  int m(int i){ ... }
}
```

Inherited specifications: a puzzle

The specification for Child is short for:

```
class Child extends Parent {
  /*@   requires i >= 0;
    @   ensures \result >= i;
    @ also
    @   requires i <= 0
    @   ensures \result <= i;
    @*/
  int m(int i){ ... }
}
```

What can the result of m(0) be?

Inherited specifications: the answer

This specification is in fact equivalent to:

```
class Child extends Parent {
  /*@  requires i <= 0 || i >= 0;
    @  ensures i >= 0 ==> \result >= i;
    @  ensures i <= 0 ==> \result <= i;
    @*/
  int m(int i){ ... }
}
```

Inherited specifications: the answer

This specification is in fact equivalent to:

```
class Child extends Parent {
  /*@ requires i <= 0 || i >= 0;
    @ ensures i >= 0 ==> \result >= i;
    @ ensures i <= 0 ==> \result <= i;
    @*/
  int m(int i){ ... }
}
```

- moral: take care specifying methods that may be overridden
- complex specifications may use a test

  `typeof(this)==\type(Parent)`

  to guard properties that are likely to change in child classes.

Methods leading to madness

Imperative programs can be very difficult to verify because of *reference escape* and *aliasing*.

```
class MyClass {
  int i;

  //@ modifies i;
  void m(MyClass o) {
    i = 3;
    o.i = 2; // ESC/Java2 gives a warning
}
```

Frame conditions

When verifying, we want to use *frame conditions* that say what stays the same when a method is executed.

Usually we want to assume that as much as possible is unchanged, but the conservative default in ESC/Java2 is:

```
//@ modifies \everything
```

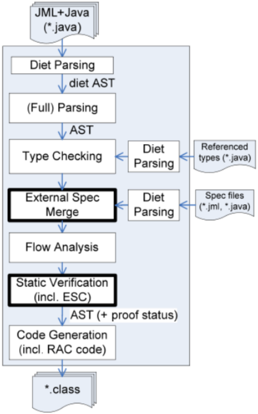
Another example where the functional paradigm is very useful:

```
//@ pure
public int getX() { return x; }
```

The **pure** annotation implies **modifies \nothing**.

Future: JMLn and ESCn

- ESC/Java2 and other JML tools have an old-fashioned *batch mode* architecture
- they're also stuck on Java 1.4
- **JML4** proposed an *Integrated Verification Environment*
- . . . integrated with Eclipse JDT
- . . . allowing multi-threaded verification, with per-method and per-class parallelism
- Development is now suspended, may be superseded by JMLEclipse and OpenJML.



JML4 compiler phases

from James, Chalin, Giannas, Karabotsos: *Distributed, Multi-threaded Verification of Java Programs*, SAVCBS 2008.

Outline

FindBugs™

- Developed since 2004 at University of Maryland, led by Bill Pugh.
- Idea: look for patterns that suggest buggy code, partly aided by program analysis techniques (e.g., for typing information, reachability).
- Examples:
  - Using == compare objects rather than `.equals`
  - Synchronizing access to a field some times but not others
  - Returning references to private fields in public APIs
  - ...and many more, built up by experience of buggy code
- To *run* FindBugs takes no effort at all by programmer
  - runs standalone on compiled class files, or in Eclipse IDE
  - false positives, coding conventions means results variable
  - *triage* very time consuming
- Some mistakes matter, others do not: http://www.cs.umd.edu/~pugh/MistakesThatMatter.pdf

Outline

Summary

The Java Modeling Language

- JML combines model-based and contract specification
- Annotations within code: **requires**, **ensures**, ...
- Provides *model* fields, methods and classes.
- Common input language for many tools

ESC/Java 2

- Combines several analysis techniques (types, dataflow, proof)
- Many checks, but exhibits false positives and missing defects
- Primarily batch mode, Java 1.4. Handles **non_null**, **modifies**, **pure**
- Follow-ups: watch jmlspecs.org and the JML specs wiki.

Findbugs and Friends

- Bug detection via bad patterns, and lightweight verification (e.g. `@NonNull`)