Advances in Programming Languages

APL6: JML — The Java Modeling Language

Ian Stark

School of Informatics
The University of Edinburgh

Monday 28 January 2008
Semester 2 Week 4

Topic: Some Formal Verification

This is the middle of three lectures about some techniques and tools for formal verification, specifically:

- Hoare logic
- JML: The Java Modeling Language
- ESC/Java 2: The Extended Static Checker for Java

Outline

1. Introduction
2. Samples of JML
3. JML Tools

Outline

1. Introduction
2. Samples of JML
3. JML Tools

Hoare Logic

- Hoare assertions $\{P\}\ C\ \{Q\}$ state that if *precondition* P holds before running code C then *postcondition* Q will hold afterwards.
- Assertions $\vdash \{P\}\ C\ \{Q\}$ can be derived using Hoare *rules*; they may also be tested against a *semantics* $\vDash \{P\}\ C\ \{Q\}$.
- This allows logical reasoning about program behaviour: notably in formal *specification* and *verification*.
- Hoare assertions are widely used in tools and languages for formal methods. (e.g. Praxis SPARK Examiner)
- Assertions may be strengthened to *contracts* for code, placing obligations on both caller and called. (e.g. Eiffel)

Model-Based Specification

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

A *model* is a representation of the desired system. It differs from a complete implemented system in that:

- A model might capture only some aspect of the system;
- A model might be partial, leaving some parts unspecified;
- A model might not be executable.

Any implementation of the system can then 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*;
- The *Z notation*, the *B language* and *B method*;
- *UML*, the *Unified Modeling Language*, a standardized general-purpose modeling language for object-oriented software engineering.

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`

Outline

1 Introduction

2 Samples of JML

3 JML Tools

JML: Basics

```
public class Account {
  private 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.

JML: Exceptions

```
public class Account {
  private 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 Formalae

```
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.

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).

Outline

1 Introduction

2 Samples of JML

3 JML Tools

JML Tools: Running and Testing

JML annotations can be used to drive various runtime checks.

- The jmlc compiler inserts runtime tests for every assertion; if they fail, error messages provide static and dynamic information about the failure.
- The jmlunit tool creates test classes for JUnit based on preconditions, postconditions and invariants. These automatically exercise and test assertions made in the code.

They also serve as documentation:

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

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.

More tools

- The *Key* dynamic logic tool has a JML front end.
- *Krakatoa* is another verification tool accepting JML.
- *Jive*, the Java Interactive Verification Environment, uses JML.
- *Houdini* will suggest JML annotations and test them with ESC/Java.
- *Daikon* analyses program runs to suggest likely JML invariants.

Finally:

- Microsoft's *Spec#* is to C# as ESC/Java 2 is to Java.

Summary

The Java Modeling Language

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

Homework

The next lecture will be on ESC/Java 2. Before Thursday, read the following two short articles:

- Leavens and Cheon. Design by Contract with JML.
- Burdy et al. An overview of JML tools and applications.

Both available from `http://jmlspecs.org`

Extra challenge activity: install and run ESC/Java 2.
(Tips: Java 1.4, Eclipse 3.3, ESC/Java 2 Eclipse plugin)