Specification of the JavaCard API in JML

Erik Poll
Joint work with Joachim van den Berg and Bart Jacobs

LOOP group,
University of Nijmegen,
the Netherlands

Cardis'2000, 20 September 2000

Contents

- The LOOP project
- The specification language JML
- Specifications for the JavaCard API: Examples
- Tool support for JML
- Conclusions

The LOOP project

Work in the LOOP group (Logic of Object Oriented Programming):

- a formal semantics of Java
- based on this semantics: verification of Java programs, using theorem provers (PVS and Isabelle/HOL), and the specification language JML
- case study: JavaCard

JML (Java Modeling Language)

Behavioural specification language for Java, developed by Gary Leavens (Iowa State Univ), in collaboration with ESC/Java group at Compaq SRC, and LOOP group in Nijmegen.

In JML, Java programs can be **annotated** with **assertions**, ie.

- pre- and postconditions *as in Eiffel/Design by Contract*
- invariants
- specification-only variables
- frame conditions (modifiability constraints)
- . . .

Annotations are special Java comments: `/*@ . . . @*/`

Pre/post-conditions and invariants are Java boolean expressions, extended with `\forall`, `\exists`, `\old( )`, . . .

JML (Java Modeling Language)

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

so that JML specs can be part of the normal development process

Example JML specification

```
public int sqrt (int n)
/*@ normal_behavior
  @  requires: n >= 0;
  @   ensures: \result * \result <= n &&
  @            n < (\result+1) * (\result+1);
  @*/
```

Pre- and postconditions form a contract between an object and its clients:

- client must ensure precondition, may assume postcondition
- object may assume precondition, must ensure postcondition

as in Eiffel/Design by Contract, and UML (OCL).

Work on/with JML

- Iowa State University (Gary Leavens):
  Parser & typechecker,
  Generator for runtime checks,
  Javadoc-like tool
- Compaq SRC (Rustan Leino et al):
  Extended static checking (ESC), ie. automatic checking of simple assertions, e.g. no NullPointerExceptions
- Nijmegen:
  Semantics of JML,
  Verification using LOOP tool and theorem provers,
  Case study: formal specs for the JavaCard API in JML.

JML specifications for the JavaCard API

Our goals:

- writing formal specs for the JavaCard API in JML
  based on existing informal specs (and the reference implementation)
- use these specs to verify reference API implementation,
  and to specify and verify applets

First step: very lightweight JML specifications for the JavaCard API.

Lightweight JML specifications

Simple complete formal specs, as for `sqrt`, are rare.
A weaker spec for `sqrt`:

```
public int sqrt (int n)
/*@ normal_behavior
  @   requires: n >= 0
  @    ensures: \result >= 0;
  @*/
```

or, weaker still:

```
public int sqrt (int n)
/*@ normal_behavior
  @   requires: n >= 0;
  @    ensures: true;
  @*/
```


Lightweight JML specifications

Our JML specifications for the JavaCard API are such lightweight specifications.

These specs only specify

- preconditions that rule out 'bad' behaviour
  ('bad' termination, ie. unwanted exceptions)
- the weakest postconditions we can get away with
  (often just `"true"`).

NB. a postcondition `"true"` does still specify something, namely termination.

Pre- and Postconditions: `arrayCompare`

```
public byte arrayCompare(byte[] src,  short srcOff,
                         byte[] dest, short destOff,
                         short length)
throws NullPointerException, IndexOutOfBoundsException;
/*@ normal_behavior
  @     requires: src != null && dest != null &&
  @               srcOff >= 0 && destOff >= 0 && length >= 0 &&
  @               srcOff + length <= src.length &&
  @               destOff + length <= dest.length;
  @   modifiable: nothing;
  @      ensures: true;
  @*/
```


Pre- and Postconditions: `arrayCompare`

Note

- This spec is *very incomplete*: it only specifies when `arrayCompare` terminates normally (i.e. does not throw an exception).
- This spec follows *immediately from the existing informal spec.*
- We could 'improve' the spec by saying that an `NullPointerException` may be thrown if `srcOff` or `destOff` is null, etc. *But does the API user have to(want to) know this ?*

Pre- and Postconditions: `arrayCopy`

```
public byte arrayCopy(byte[] src,  short srcOff,
                      byte[] dest, short destOff,
                      short length)
throws NullPointerException, IndexOutOfBoundsException,
       TransactionException;
/*@ behavior
  @    requires: src != null && dest != null &&
  @              srcOff >= 0 && destOff >= 0 && length >= 0 &&
  @              srcOff + length <= src.length &&
  @              destOff + length <= dest.length;
  @     ensures: true;
  @  modifiable: dest[*];
  @     signals: (TransactionException) true;
  @*/
```


Pre- and Postconditions: `arrayCopy`

Note

- Precondition does not rule out all exceptions, hence the use of `behaviour` instead of `normal_behaviour`
- Again, this spec follows *immediately* from the existing informal spec.
- Again, we have chosen not to specify what may happen if the precondition is not met.
- We can be more precise and say

```
modifiable: dest[destOff]..dest[destOff+length-1]
```


Invariants: `AID`

The AID class has a field

```
byte[] theAID;
```

to record an Applet ID, which consists of a 5 bytes RID (Registered applet provider ID) plus up to 11 bytes to identify the applet.

Hence the following invariant should hold for any `AID`-object:

```
/*@ invariant: theAID != null &&
  @            5 <= theAID.length && theAID.length <=16;
  @*/
```

The API implementation relies on this invariant!
This invariant is not mentioned in the informal specs!

Informal verification of specifications

```
public boolean equals( Object anObject )
  /*@ normal_behavior
    @  requires: true;
    @   ensures: true;    // Is this spec satisfied ?
    @*/
{if ( !(anObject instanceof AID)
      || ((AID)anObject).theAID.length != theAID.length)
      return false;
 return (Util.arrayCompare(((AID)anObject).theAID, 0,
                           theAID, 0, theAID.length)
        == 0);
}
```

We need the AID-invariant for both `this` *and* `anObject`.

These very basic JML specifications are already more informative than the existing informal specs (or the code):

- some JML specs mention possible exceptions not mentioned in the informal specs or in the code
- some JML specs rule out exceptions that are mentioned in the informal specs or in the code

For example, the AID constructor . . .

The AID constructor

```
public AID(byte[] bArray, short offset, byte length )
throws SystemException;
/*@ behavior
  @    requires: bArray != null &&
  @              5 <= length && length <= 16 && offset >= 0 &&
  @              offset+length <= bArray.length;
  @     ensures: true;
  @     signals: (TransactionException) true;
  @*/
{ if (length < 5 || length > 16)
       SystemException.throwIt(SystemException.ILLEGAL_VALUE);
 theAID = new byte[length];
 Util.arrayCopy(bArray, offset, theAID, 0, length); }
```

The informal spec fails to mention this possible `TransactionException`

Specification inheritance

A class **inherits** the specifications of the superclass it extends, and of the interfaces it implements, ie.

- subclass must maintain invariants of superclass
- subclass methods must have *weaker* pre- and *stronger* postconditions
- subclass can only modify what superclass is allowed to modify

In other words, the specification of a class or interface constrains its (future) subclasses. This requires some foresight!

Java itself already enforces specification inheritance for *checked exceptions*.

Specification inheritance: the PIN interface

Specifying

```
public interface PIN {
   :
   boolean isValidated()
    // Returns true if a valid PIN value has been presented
    // since the last card reset or last call to reset().
    /*@ normal_behavior
      @  requires: true;
      @   ensures: true;
      @*/
```

means that all classes that implement `PIN` *must satisfy this spec.*

This spec just says "`isValidated` always terminates normally."

Specification inheritance: the OwnerPIN class

The class OwnerPIN *implements the interface* PIN*:*

```
public class OwnerPIN implements PIN {
⋮
public boolean isValidated()
{ return getValidatedFlag(); }
⋮
protected boolean getValidatedFlag()
⋮
```



```
protected boolean getValidatedFlag()
/*@ behavior
  @  requires: true;
  @   ensures: true;
  @   signals: (SystemException) true;
  @*/
```

ie. getValidatedFlag *may throw a* SystemException

Specification inheritance: PIN and OwnerPIN

The correct specification of isValidated *in* OwnerPIN *and (hence) of* PIN*:*

```
/*@ behavior
  @  requires: true;
  @   ensures: true;
  @   signals: (SystemException) true;
  @*/
```

Developing lightweight JML specs reveals unexpected behaviour!

The existing informal specs of PIN *and* OwnerPIN *fail to mention this possible* SystemException*.*

Model Variables

Sometimes it is not possible to express pre/post-conditions or invariants in terms of the fields and methods of an object, because these depend on "state" which implictly exists, but is not (explicitly) recorded in fields or methods.

Therefore JML allows so-called model fields to be introduced, which only exist for specification purposes.

For example, . . .

The APDU class

The APDU class handles the communication between applets and the card terminal via the methods

```
public static short getInBlockSize()
public static short getOutBlockSize()
public short setIncomingAndReceive()
public short receiveBytes(short bOff)
public short setOutgoing()
public short setOutgoingNoChaining()
public void setOutgoingLength(short len)
public void sendBytes(short bOff, short len)
public void setOutgoingAndSend(short bOff, short len)
```

(using ISO7816, but hiding much of the complexity.)

Specifying the APDU class

There are **many** constraints on the order in which the APDU methods may be invoked. In the informal API specifications these are descibed by statement of the form

> "if method X is invoked after (before) method Y, an APDUException is thrown".

How can we formally specify this in JML ?

We must know the "state" the protocol is in: ie.

- make a finite state machine (FSM) describing the correct invocation sequences
- introduce a model variable that records the state

Using a model field to specify APDU

```
//@ public model int _APDU_state;

public short setIncomingAndReceive()
 /*@ behavior
   @    requires: _APDU_state == 1 && ... ;
   @     ensures: _APDU_state == 2 && ... ;
   @*/


public short receiveBytes(short bOff)
 /*@ behavior
   @    requires: _APDU_state == 2 && ... ;
   @     ensures: _APDU_state == 2 && ... ;
   @*/
```


Using a model field to specify APDU

Unlike most other specs, the JML specification of the APDU class does not immediately follow from the existing informal specs.

But the desciption of the APDU protocol using a FSM is much clearer and less error-prone than a large collection of statements of the form

> "if method X is invoked after (before) method Y, an APDUException is thrown".

Tool support for JML

Formal interface specifications enable **tool support**.

Tools for JML (under construction):

- Iowa:
  - parser & typechecker
  - automatic generation of runtime checks, to check for violations of preconditions and invariants during testing (as in Eiffel)
  - javadoc-like tool
- Compaq: extended static checking (ESC), automatic checking of simple assertions, eg. no NullPointerExceptions
- Nijmegen: verification using LOOP tool, interactive verification of arbitrary assertions.

Extended Static Checking (Compaq SRC)

Automatic detection of errors in program (or specification!). E.g. for

```
public boolean check(byte[] pin, short offset, byte length)
 /*@ normal_behavior
   @  requires: pin != null && length <= maxPINSize &&
   @            offset+length <= pin.length
   @   ensures: true;
   @*/
{... Util.arrayCompare(pin, offset, pinValue, 0, length); ...}
```

ESC signals possible violation of the precondition of `arrayCompare`, namely in case `offset < 0`.

Interactive verification of JML-annotated programs, using LOOP tool as a front end to theorem provers:

JML annotated program

⇓

LOOP tool

⇓

Proof obligations

⇓

PVS or **Isabelle** ⇝ **user interaction!**

⇓

QED

Conclusions

The lightweight JML specs for the JavaCard API

- are easy to read, write, and (informally) verify.
- provide useful improvement over/addition to the existing informal specs (and reference implementation), notably because
  - precise declaration of exceptions
  - invariants

  The JML specs make many **implicit** assumptions and considerations that go into design of code **explicit**

Ongoing Work

- Using tool support both on applets and API implementations
- Publishing formal API specs on the Web
- Developing more detailed specs of the API