Formal Specification and Verification of JavaCard's Application Identifier Class

Bart Jacobs

Dep. Comp. Sci, Univ. Nijmegen

bart@cs.kun.nl

http://www.cs.kun.nl/~bart

Joint work with:

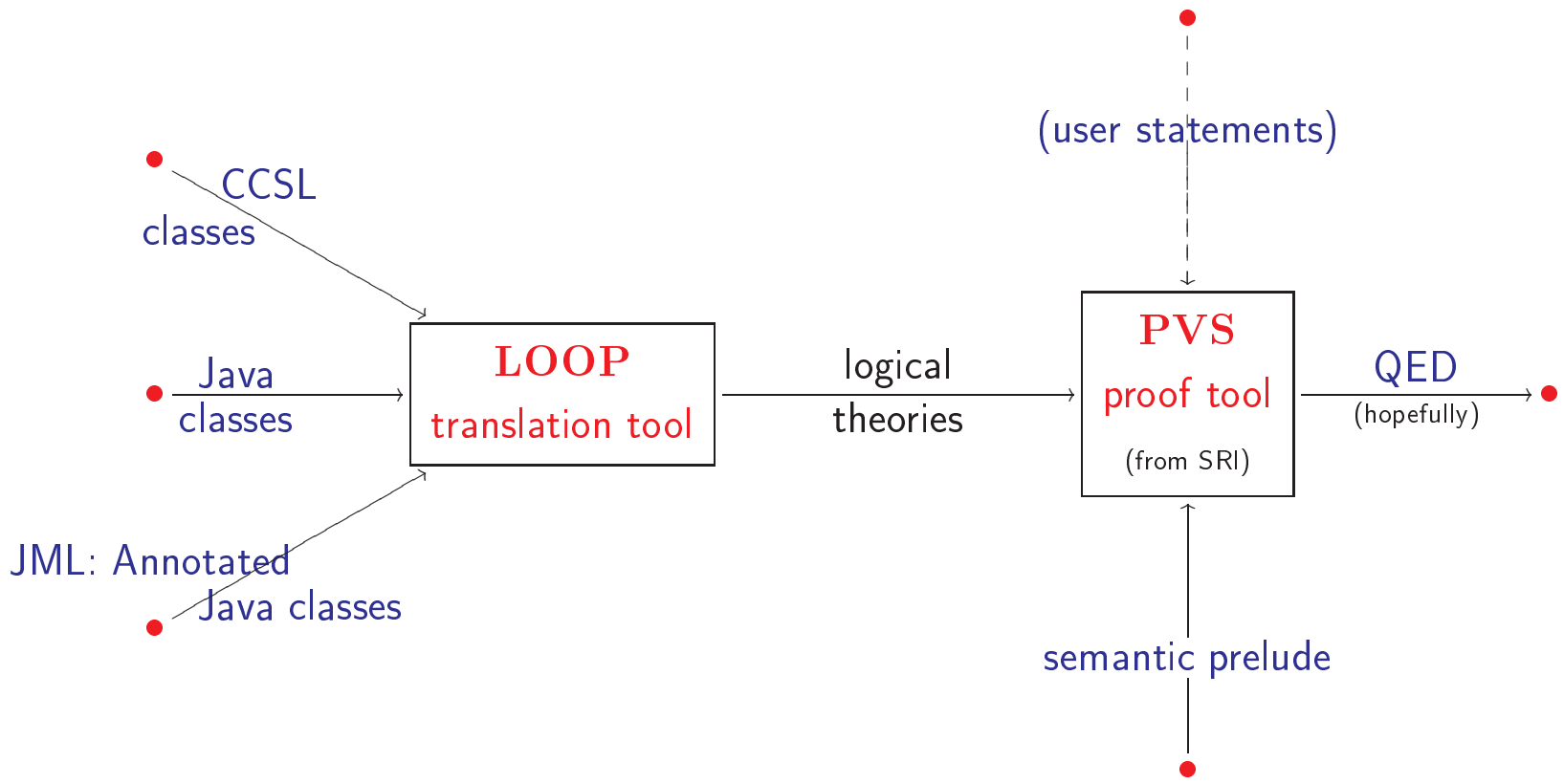
Joachim van den Berg

Erik Poll

Contents

- The LOOP project: Logic of Object-Oriented Programming
- JML: a behavioural interface specification language for Java
- JavaCard API specification in JML: the Util class
- JavaCard API verification in PVS: the AID class
- Related work & Conclusions

LOOP tool



The LOOP tool is implemented in Ocaml and has an Emacs interface.

It is possible to use also Isabelle as a back-end theorem prover.

LOOP essentials

A fresh attack on the old goal of program verification combining:

- modern semantical theories, using coalgebras to model state-based (object-oriented) systems;
- powerful tools for translating (LOOP) and reasoning (PVS/Isabelle);
- a formal specification language JML that grew out of experience with Larch/C++ and builds on the design-by-contract approach from Eiffel.
- application to real (non-idealised) programming languages (namely Java and JavaCard), with all their messy details.

Applet Verification

- is important, given the sensitivity of the applications
- seems feasible, given the limited size & complexity of the language and the platform
- much wanted by the card issuers, in order to protect their products, services and their image.

Our starting point: formal specifications for the JavaCard API, based on existing informal specifications.

Currently, the API is about 45 classes, of which 20 trivial.

We use JML as formal specification language.

(There is also platform verification dealing for example with the JavaCard virtual machine and run-time environment; it is a different topic.)

JML: Java Modeling Language

JML is a behavioural interface specification language, developed by Gary Leavens et al at Iowa State Univ, in collaboration with the ESC/Java group at Compaq SRC, and the LOOP group in Nijmegen.

In JML, Java programs can be annotated with assertions, that give

- pre- and postconditions (like in Eiffel)
- invariants and constraints
- specification-only variables
- frame conditions (modifiability constraints)
- ...

These annotations are written as special comments: `/*@ ... @*/`

Pre-/post-conditions, invariants and constraints in JML are Java boolean expressions, extended with special operators like `\forall`, `\exists`, `\old( )`, ...

Goal: JML should be usable by Java programmers, without assuming mathematical training.

JML Example

```
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, and may assume postcondition
- object may assume precondition, and must ensure postcondition

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

Weaker specs for `sqrt`:

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

or

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

At this stage, our JML specifications for the JavaCard API are such lightweight (incomplete, weak) specs.

These specs only describe

- preconditions that rule out "bad" behaviour ("bad" termination, i.e. unwanted exceptions)
- the weakest postconditions we can get away with (nearly always `"true"`): no functionality

Such weak specifications are useful because they make many implicit assumptions explicit.

JML Tools

Tools for JML are developed at Iowa, Compaq SRC (Leino et al) and Nijmegen:

- Iowa: parser, typechecker, generator for run-time checks, javadoc-like tool
- Compaq: Extended Static Checker ESC/Java
- Nijmegen: LOOP tool

Ultimate Goal

Verification of JavaCard applets, on the basis of a (verified) specification of the JavaCard API.
Note: this goes well beyond testing.

Current Goal

Specification of the JavaCard API, and verification of the current reference implementation (version 2.1).

The specification should be verified again for each new version of the library.

Current subgoal

"Lightweight" specification (and verification) of the API, giving only termination, modification and invariant clauses, but no functionality.

Two examples, of native methods: `arrayCompare` and `arrayCopy` from `Util.java`

arrayCompare in JML

```
public static final native 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;
  @*/
```

arrayCopy in JML

```
public static final native 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;
  @ modifiable: dest[*];
  @    ensures: true;
  @    signals: (TransactionException) true;
  @*/
```

Note:

- `behavior` denotes a partial Hoare formula (actually: two)
- We could be more precise and say that only `dest[destOff]...dest[destOff+length-1]` are modified

The JavaCard API contains an Application Identifier class `AID`, which contains one field:

```
byte[] theAID;
```

plus several associated methods: `getBytes`, `equals` (twice), `partialEquals`, `RIDEquals`.

The first five bytes are reserved for the RID (National Registered Application provider identifier). Eleven further bytes (PIX) may be used for identification.

Therefor we have as AID class invariant in JML:

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

This property is established by the AID constructor, and preserved by all AID methods (whether terminating normally or abruptly). Problem/Bug: `theAID` is not private.

getBytes from AID

```
public byte getBytes (byte[] dest, short offset)
   throws ArrayIndexOutOfBoundsException, NullPointerException,
          TransactionException // not in the original source
    /*@ behavior
      @     requires: dest != null &&
      @               offset >= 0 &&
      @               offset + theAID.length <= dest.length;
      @   modifiable: dest[*];
      @      ensures: true;
      @      signals: (TransactionException) true;
      @*/
   {
     Util.arrayCopy(theAID, (short)0, dest, offset, (short)theAID.length);
     return (byte)theAID.length;
   }
```

getBytes translation

Let

$$\mathsf{Pre} \;=\; \Big(\texttt{dest != null \&\& offset >= 0 \&\& offset + theAID.length <= dest.length}\Big)$$

$$\mathsf{Inv} \;=\; \Big(\texttt{theAID != null \&\& 5 <= theAID.length \&\& theAID.length <= 16}\Big)$$

$$\mathsf{Mod} \;=\; \Big(\cdots\Big) \quad // \quad \text{Interpretation of modifies clause: all memory cells remain the same, except the ones referred to.}$$

The LOOP tool translates the JML spec of `getBytes` into two partial Hoare triples in PVS:

$$\Big\{\, [\![\, \texttt{Inv \&\& Pre} \,]\!] \,\Big\}\ \texttt{getBytes(dest, offset)}\ \Big\{\, [\![\, \mathsf{Inv} \,]\!] \wedge \mathsf{Mod} \,\Big\}$$

$$\Big\{\, [\![\, \texttt{Inv \&\& Pre} \,]\!] \,\Big\}\ \texttt{getBytes(dest, offset)}\ \Big\{\, \mathsf{Excp}\big([\![\, \mathsf{Inv} \,]\!] \wedge \mathsf{Mod}, \mathsf{TransactionException}\big) \,\Big\}$$

Normal termination

Assume pre-condition and invariant, and getBytes terminates normally. To prove:

1. Only the elements of the parameter array `dest` may have changed. This follows from the specification of arrayCopy, which can be used because its pre-condition is satisfied.
2. The invariant holds in the post-state. Again this follows from the specification of arrayCopy. We have to be careful about aliasing: possibly `dest == theAID`, but in that case the invariant remains valid because only the contents of `dest` may change.

Abrupt termination

Assume pre-condition and invariant, and getBytes terminates abruptly. To prove:

1. Only the elements of the parameter array `dest` may have changed.
2. The invariant still holds.

The proofs are as before, except that we first need to show that the abrupt termination is caused by `arrayCopy`. Then the relevant specification can be applied.

Source code work

- The group of Arnd Poetzsch-Heffter in Hagen (Germany) has developed the Jive tool, using a Hoare logic on the program level (instead of semantic level). Its semantics is not so well-developed yet (e.g. no exceptions).

- The group of Tobias Nipkow in Munich (Germany) works on meta-level formalisation in Isabelle of large parts of Bali and JavaCard for:
  - proving type-safety
  - soundness and completeness of a Hoare logic

  (There is also much bytecode level work in this group.)

- The Key group in Karlsruhe uses a translation of an extension of OCL (assertion language of UML) to dynamic logic. The prover part is still under construction.

  See the next talk ....

Conclusions

- JavaCard is an ideal target for the (tool-supported) use of formal methods: size, importance, interest, openness of the industry.
- JML is an appropriate language for specifying the behaviour of Java(Card) classes and methods.
- The semantics of Java developed within the LOOP project extends smoothly to JML, and brings forward many subtle issues in the semantics of object-oriented specification.

  (The LOOP translation of JML is far from complete.)
- The actual verification results sofar are modest, but the fact alone that problems with such complexity can now be managed is already a significant step forwards.

More information at the URL `http://www.cs.kun.nl/~bart/LOOP`