Java Program Verification for Smart Cards

Bart Jacobs
bart@cs.kun.nl

Department of Computer Science
University of Nijmegen, The Netherlands

Joint work with
Joachim van den Berg, Cees-Bart Breunesse, Engelbert Hubbers,
Hans Meijer, Martijn Oostdijk, Erik Poll, Martijn Warnier

Outline

I. Intro on smart cards and JavaCard
II. European IST-project VeriCard (www.verificard.org)
III. LOOP project at Nijmegen: Java and JML to PVS
IV. Java semantics using coalgebras
V. Hoare logic for JML
VI. Examples:
   - Decimal class for Purse applet
   - Phone card applet
VII. Conclusions & future work (scalability & security).

Smart Cards

Latest generation of cards has microprocessor plus memory (RAM, ROM, EEPROM) capable of:

- storing information (tamper-resistant!)
- processing information, in particular: encryption / decryption using private key on card.

Main applications:

Now: bank cards, mobile phone SIMs, settop boxes
Future: general identity cards, with PKI support
Cards and keys

Key pair of person $A$: $\langle$ private key $k_1$, public key $k_2$ $\rangle$

<table>
<thead>
<tr>
<th>Name</th>
<th>Public key</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$A$</td>
<td>$k_2$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$:$</td>
<td>$:$</td>
</tr>
</tbody>
</table>

Smart card of $A$  PKI = Public Key Infrastructure

Old and New Cards

*Traditional smart cards:*
- Code in native, low-level assembly language, burned into ROM
- difficult to develop, and to re-use: inflexible

*New generation of open cards:*
- mini OS, providing a common basis: abstract machine, API, crypto
- abstraction from hardware & vendor-specific details
- multi-application: several applications (“applets”)
- post-issuance: new applets can be downloaded

JavaCard

JavaCard is a superset of a subset of Java, for programming the new generation of smart cards:

- It is a *simplification* of Java:
  - No threads, floats, strings, multi-dim arrays, optional GC; because of hardware constraints

- It is an *extension* of Java:
  - Persistent and transient objects + transaction mechanism
  - Increased security: standard sandbox (cf. web-browsers) + firewall between applets

II. European Project VerifiCard
VerifiCard: background

- **Multi-application + post-issuance downloading** is both strength and weakness:
  Imagine a *virus* on your smart card!
- A *malicious* applet may do much damage, by:
  - exploiting a weakness of the platform
  - interfering with other applets
- Smart card industry is faced with higher certification demands, e.g. **Common Criteria (CC) evaluations**:
  - seven levels of evaluation
  - highest levels (6 + 7) require *formal* verifications
  - current card evaluations at levels 4 and 5.

VerifiCard: Relevance

JavaCard is an **ideal target** for use of formal methods:

- The systems involved are relatively *small*, and within reach of current verification technology + tools
- Correctness + security are essential for *acceptance* of new card-based services
- Cards are distributed in *large* numbers
- Smart card industry is *open to formal methods* (because of evaluations + security is their “product”)

Potential *killer* application for formal methods in software (both positively and negatively)

VerifiCard: Topics

*Tool-assisted* verification for JavaCard, along the lines:

<table>
<thead>
<tr>
<th>JavaCard</th>
<th>platform</th>
<th>applets</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte code</td>
<td>JCVM, Verifier, CAP converter, compiler formalisation</td>
<td>abstract interpretation &amp; model checking</td>
</tr>
<tr>
<td>source code</td>
<td>language semantics &amp; logic, and API annotation</td>
<td>applet annotation in JML &amp; Hoare-style verification</td>
</tr>
</tbody>
</table>

III. **LOOP project at Nijmegen**
**LOOP project: overview**

- **Java + JML**
- **LOOP translation tool**
- **PVS proof tool**
- **logical theories**
- **qed**
- JML annotations become PVS **predicates**, which should be proved for the (translated) Java code.
- The **semantic prelude** contains the semantics in PVS of Java language constructs like composition, if-then-else, while, try-catch-finally, ...

**LOOP project: results**

- Translation covers essentially **all of sequential Java**
- Translation of **JML** is still under construction, but covers the basics: class invariants & constraints, method specifications including modifiable clauses, but not yet model variables
- Major **case studies**:
  - non-trivial invariant for Java’s Vector class
  - functional specification & verification of JavaCard’s AID class
  - applet case studies
  - ESC/Java + JML JavaCard API specs (on web).

---

**JML: Java Modeling Language**

In **JML** [Leavens et al.] one may add specifications as special comments in Java code, for:

- Class invariants and constraints
- Method specifications:

```java
/*@ behavior
  @ requires <precondition>
  @ modifiable <items that may be modified>
  @ diverges <precond for non-termination>
  @ ensures <postcond for normal termination>
  @ signals <postcond for exceptional termination>
*/

void method() { ... }
```

- Model variables (specification only)

**JML: example**

JML method specifications may clarify the behaviour of Java methods:

```java
/*@ normal_behavior
  @ requires x >= 1;
  @ modifiable \nothing;
  @ ensures result * result <= x &&
  @ x < (\result + 1) * (\result + 1)
*/

int square(int x) {
  int count = 0, sum = 1;
  while (sum <= x) {
    count++;
    sum += 2 * count + 1;
  }
  return count;
}
```
JML: difference with javadoc

- Javadoc also uses special comments `/**...*/` to annotate programs, with special tags like `@param` and `@return`.
- A special compiler recognises these comments and produces html pages for uniform documentation.
- **But**: JML has a formal semantics and its assertions can be formally verified.
- For this purpose, its language is richer (e.g. has class invariants).

JML: Tool support

Especially appealing: range of options for JML:

- **runtime checking** — **static checking** — **interactive verification**

- **Iowa** (Leavens et al.): JML parser, typechecker, inserter of run-time checks
- **Compaq** (Leino et al.): extended static checking: automatic verification of simple assertions
- **MIT** (Ernst): Daikon invariant detection tool
- **Nijmegen** theorem proving, via LOOP tool: interactive verification of arbitrary assertions.

JML with its tools is also great in (formal) Java courses!

IV. Java Semantics

- Statements and expressions are standardly modeled as state transformer functions acting on a suitable (global) state space
- Before looking at the situation in Java, we first look at statements with possible non-termination
- One can distinguish two representations:
Option 1: Modify the state space:

\[(\{\text{hang}\} + \text{State}) \xrightarrow{\text{stat}} (\{\text{hang}\} + \text{State})\]

with requirement:

\[\text{stat}(\text{hang}) = \text{hang}\]

- The state space gets more complicated, but statement composition is simply function composition.
- However, side-conditions have to be kept in mind.

Option 2: Modify the output type:

\[\text{State} \xrightarrow{\text{stat}} (\{\text{hang}\} + \text{State})\]

- State space remains simple, and there are no side-conditions.
- But statement composition must deal with non-termination:

\[(s_1 ; s_2) \cdot x = \text{CASES } s_1 \cdot x \text{ OF } \{\]
  \[\quad \text{hang } \mapsto \text{hang} \]
  \[\quad y \mapsto s_2 \cdot y \}

Modeling exceptions: Two ways:

- **In the state space**
  - Most common, e.g. used by *Alves-Foss & Lam* (with continuations), or *von Oheimb & Nipkow*.
  - **Disadvantage**: complicates the state space; not so transparent; non-trivial side-conditions.

- **In the output type of the state transformers**
  - *Coalgebraic* way, with statements and expressions:

\[\text{State} \xrightarrow{\text{stat}} \{\text{hang}\} + \text{State} + (\text{State} \times \text{Excp})\]

\[\text{State} \xrightarrow{\text{expr}} \{\text{hang}\} + (\text{State} \times \text{Out}) + (\text{State} \times \text{Excp})\]

Coalgebras

- General form: functions with *structured* output type:

\[\text{State} \xrightarrow{} \cdots \text{State} \cdots\]

- No clutter in state space, nor complicated side conditions
- The type system forces explicit handling of all termination modes via pattern matching.
  - This works very naturally, both in specification and verification. See exception mechanism paper at [ESOP’01].
Java Expressions

- State $\xrightarrow{\text{expr}} \{\text{hang}\} + (\text{State} \times \text{Out}) + (\text{State} \times \text{Excp})$
- with constructors

$$\text{expr} \cdot x = \begin{cases} \text{hang} \\ \text{norm}(y, r) \\ \text{excp}(z, e) \end{cases}$$

allowing pattern matching

- Possible in a simple type theory with products $\times$ and coproduct $+$ types.

Java Statements I

- First version

$$\text{State} \xrightarrow{\text{stat}} \{\text{hang}\} + \text{State} + (\text{State} \times \text{Excp})$$

is too simple: expressions in Java can only terminate abruptly because of an exception, but statements also because of a return, break, or continue. Instead:

- State $\xrightarrow{\text{stat}} \{\text{hang}\} + \text{State} + \text{Abrupt}$

where Abrupt is:

$$(\text{State} \times \text{Excp}) + \text{State} + \text{State} \times \text{Lift} \text{(string)} + (\text{State} \times \text{Lift} \text{(string})$$

Java Statements II

This representation involves nested pattern matching:

$$\text{stat} \cdot x = \begin{cases} \text{hang} \\ \text{norm}(y) \\ \text{abnorm}(a) \end{cases}$$

where:

$$a = \begin{cases} \text{excp}(z, e) \\ \text{rtrn}(z) \\ \text{break}(z, \ell) \\ \text{cont}(z, \ell) \end{cases}$$

Example: Composition

- For two statements

$$\text{State} \xrightarrow{s_1} \{\text{hang}\} + \text{State} + \text{Abrupt} \xrightarrow{s_2} \{\text{hang}\} + \text{State} + \text{Abrupt}$$

- Define the composite statement as:

$$\left( s_1 ; s_2 \right) \cdot x = \text{CASES } s_1 \cdot x \text{ OF } \{ \right.$$  \\
$$| \text{hang} \mapsto \text{hang} \\
| \text{norm}(y) \mapsto s_2 \cdot y \\
| \text{abnorm}(a) \mapsto \text{abnorm}(a) \}$$

In this way all Java constructs are translated into PVS.
V. Hoare Logic for JML

Hoare logic for JML 1 [FASE’01]

- This Hoare logic not at syntactic, but semantic level:
  I.e. not \( \{ P \} \ m \ { Q \} \), but \( \{ P \} \ \| \ m \ \| \ { Q \} \),
  Since \( \| s_1 ; s_2 \| = \| s_1 \| ; \| s_2 \| \) proofs are still “syntax driven”

- Complications in Hoare logic for Java:
  - exceptions and other abrupt control flow
  - expressions may have side effects

  Thus:
  - not Hoare triples but Hoare n-tuples,
  - both for statements & expressions

Hoare Logic for JML 2

For \( \{ Pre \} \ m \ { Post \} \) write
\[
\begin{align*}
\text{requires} & = \text{Pre} \\
\text{statement} & = m \\
\text{ensures} & = \text{Post}
\end{align*}
\]

For JML one needs:
\[
\begin{align*}
\text{diverges} & = D \\
\text{requires} & = \text{Pre} \\
\text{statement} & = m \\
\text{ensures} & = \text{Post} \\
\text{returns} & = \text{Pre} \\
\text{signals} & = \text{exc} \\
\text{return} & = \text{ret} \\
\text{break} & = \text{brk} \\
\text{continue} & = \text{cnt}
\end{align*}
\]

Logic for JML 3: Composition Rule

\[
\begin{align*}
\text{requires} & = \text{Pre} \\
\text{statement} & = s_1 \\
\text{ensures} & = \text{Post} \\
\text{signals} & = \text{exc} \\
\text{return} & = \text{ret} \\
\text{break} & = \text{brk} \\
\text{continue} & = \text{cnt}
\end{align*}
\]

\[
\begin{align*}
\text{requires} & = Q \\
\text{statement} & = s_2 \\
\text{ensures} & = \text{Post} \\
\text{signals} & = \text{exc} \\
\text{return} & = \text{ret} \\
\text{break} & = \text{brk} \\
\text{continue} & = \text{cnt}
\end{align*}
\]

\[
\begin{align*}
\text{requires} & = \text{Pre} \\
\text{statement} & = s_1 ; s_2 \\
\text{ensures} & = \text{Post} \\
\text{signals} & = \text{exc} \\
\text{return} & = \text{ret} \\
\text{break} & = \text{brk} \\
\text{continue} & = \text{cnt}
\end{align*}
\]

Filled in only during proofs

Corresponding to coalgebra +-options
Weak Precondition Calculus (WPC)

- Most recent addition: WPC, integrated within Hoare logic for JML.
- Implemented as (provable) rules in PVS, used for automatic rewriting in special command (WP-ASSERT), using dedicated PVS strategies.
- Automatically computes WPs for all Java constructs, excepts loops—which need (in)variant annotations.
- Requires much computational power
- Failure traces, if any, are hard to interpret—like with model checking.

WP-rule for plus

\[
\begin{align*}
\forall i_1, i_2.
\text{requires} & = Pre \\
\text{stateexprs} & = \ell ; (e_1 \Rightarrow i_1) ; (e_2 \Rightarrow i_2) \\
\text{ensures} & = \lambda x. i = i_1 + i_2 \Rightarrow Post(x) \\
\text{signals} & = P_{exc} \\
\text{return} & = P_{ret} \\
\text{break} & = P_{brk} \\
\text{continue} & = P_{cnt}
\end{align*}
\]

Purse example [AMAST’02]

- Electronic purse applet case study provided by smart card manufacturer Gemplus.
  (debit, credit, currency conversion, communication with loyalty applets)
- About 65 K of Java code, check in ESC/Java (Cataño & Huisman): numerous small bugs.
- Utility class Decimal verified with the LOOP tool:
  Several obvious bugs; complicated methods verified.

VI. Applet Case Studies:

- Purse applet
- Phone applet
Purse example: Decimal class 1

- There are no floats in JavaCard, so a currency amount is a pair of *shorts*

\[(intPart, \text{decPart})\]

with fixed \text{PRECISION} = 1000.

- Decimal class invariant:

\[
\text{
}@\text{ invariant } -\text{PRECISION} < \text{decPart} \\
&& \text{decPart} < \text{PRECISION}; @
\]

Purse example: Decimal class 2

- €1.35 can be

\[
\text{(intPart = 1, decPart = 350)} \\
\text{(intPart = 2, decPart = -650)}
\]

- Relevant value:

\[
\text{intPart} * \text{PRECISION} + \text{decPart}
\]

should be used as *model variable*.

- Hence we use 1350 *milli-€*, to prove correctness of addition, multiplication, subtraction, etc.

Purse example: add method

\[
\text{/*@ normal behavior} \\
\text{@ requires } -\text{PRECISION} < f && f < \text{PRECISION}; \\
\text{@ modifiable intPart, decPart;} \\
\text{@ ensures intPart } * \text{PRECISION} + \text{decPart} == \\
\text{@ } (\text{old(intPart)}+e) * \text{PRECISION} + (\text{old(decPart)}+f); \\
\text{/*@}
\]

\[
\text{private void add(short e, short f) \{} \\
\text{intPart } += e; \\
\text{if (} \text{intPart} ) \{} \\
\text{intPart } -= \text{short}(\text{decPart }+ \text{PRECISION}); \} \\
\text{intPart } -= \text{short}(\text{decPart }- \text{PRECISION}); \} \\
\text{decPart } += f; \\
\text{if (} \text{intPart} ) \{} \\
\text{decPart } += \text{short}(\text{decPart }+ \text{PRECISION}); \} \\
\text{intPart } += \text{short}(\text{decPart }- \text{PRECISION}); \} \\
\text{else } \{} \\
\text{short retenue } = \text{short}(0); \text{ short signe } = 1; \\
\text{if (} \text{decPart} < 0 \{} \text{signe } = \text{short}(-1); \} \\
\text{retenue } = \text{short}(\text{decPart }/ \text{PRECISION}); \\
\text{decPart } -= \text{short}(\text{decPart } \% \text{PRECISION}); \\
\text{retenue } *= \text{signe}; \text{ decPart } *= \text{signe}; \text{ intPart } += \text{retenue}; \}
\]

Hoare logic proof tree in PVS
Weak Precondition proof

- Add example involves too many case distinctions: PVS becomes too big and crashes after handling > 66.
- Two solutions ...

WP solution 1: cheat

Simplification:
replace \&\& by \&, to avoid many splits.

Resulting prooftree:

WP solution 2: combine Hoare + WP

Use Hoare logic rules to break up the proof goal into manageable parts, to be handled by (WP–ASSERT).

For the add method:

Example: simple phone card applet

- 3 operations: decrement-value, show-value, set-value
- set-value may only be used once before the card is ‘issued’
- Communication with card proceeds (in two directions) via special byte sequences, called apdu’s. Apdu’s are handled by a method process.
- The applet uses two instance variables:
  private byte value;
  private boolean issued;
Process specification + implementation

/*@ behavior  
@ requires apdu != null;  
@ modifiable value, issued, apdu.buffer[*];  
@ ensures \old(issued) ==> (issued &&  
  @ value <= \old(value));  
@ signals (ISOException e) \old(issued) &&  
@ (value <= 0 || apdu.buffer[1] == 0x73)  
@ && issued && value == \old(value);  
@*/

public void process(APDU apdu) {
    byte[] buffer = apdu.getBuffer();
    if (value <= 0 && issued)
        ISOException.throwIt((short)0x6982);
    else
        switch (buffer[1]) {
        case 0x71: value--; break;
        case 0x72: apdu.setOutgoing();
            apdu.setOutgoingLength((short)1);
            buffer[0] = value;
            apdu.sendBytes((short)0,(short)1); break
        case 0x73: if (issued)
                     ISOException.throwIt((short)0x6982);
                     else
                     value = buffer[2]; issued = true;
                     break;
        }
}

Scalability

The LOOP verification technology for Java + JML works. Now we have to scale it further, via:

- Just waiting for faster hardware!
- Reducing the user-interaction:
- even better PVS strategies
- Weak precondition reasoning, integrated with
- annotated programs (proof outlines)
Extension to Security

Authentication Protocol messages

Organisation — Individual

refinement is missing

ultimate goal: whole spectrum

Bit-sequences

Terminal — Smart card

traditional security research

current own correctness research

Conclusions

- JavaCard presents a real challenge and opportunity for formal methods to deliver in the software area.
- The tool-based verification technology for Java(Card) is there, but scaling it up to larger programs is still a challenge.
- Standard approaches (Hoare, WP) work, after non-trivial extensions.
- Transfer to industry is happening: Gemplus, Schlumberger, Ericsson, IBM, France Telecom, TNO, ... 

More info at: [www.cs.kun.nl/~bart/LOOP](http://www.cs.kun.nl/~bart/LOOP)