JavaCard Program Verification

Erik Poll

University of Nijmegen

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

Outline

Background:

- **Smart cards and JavaCard**

LOOP project at Nijmegen:
verification of JML-annotated JavaCard programs in PVS

- **a semantics of Java**
- **JML specification language for Java**
- **a logic for JML**
- **JML specifications for the JavaCard API**

Smart Cards and Java Card

smart cards

Card with a chip providing CPU and memory (ROM, RAM, EEPROM) capable of

- **storing information (tamper resistant!)**
- **processing information, notably en/de-cryption**
  NB private keys never have to leave the card !

Applications

- **Now: bank card, mobile phone SIM**
- **Future: ID cards, access control for networks, PKI support, access control for networks, . . .**

Old vs new smart cards

Traditional smart cards:

- one program (or 'applet')
- written in chip-specific machine code
- burnt into ROM

New generation smart cards:

- several applets, written in high level language
- compiled to byte code
- stored in EEPROM
- executed on virtual machine and mini-OS, which hide hardware details

Java Card

Subset of Java for programming smart cards,

- without threads, floats, . . . , very limited API

extended with

- persistent and transient objects (EEPROM and RAM)
- transaction mechanism

and increased security:

- standard sandbox + firewall between applets.

Pros & cons

Advantages of new generation smart cards:

- development quicker and cheaper
- multi-application: several (possibly interacting) applets on one smart card
- post-issuance download: adding or deleting applets on a card (cf. downloading applets in web browser, but controlled with digital signatures, using Visa OP)

but additional security threats !

Security issues for smart cards

Like a virus, a malicious applet could exploit weaknesses in platform or in other applets.

- Is (an implementation of) the platform secure ?
- Is a given applet secure/not malicious ?

Increasing demands for security evaluation.

Common Criteria (CC), the ISO standard for security evaluation, distinguishes 7 levels.

Current cards are evaluated up to levels 4+5.
The highest levels (6+7) require formal verification.

Formal methods for JavaCard

JavaCard is an ideal target for use of formal methods:

- Programs involved are small
- Platform is relatively small and simple
- Correctness & security are of vital importance
- Cards are distributed in large numbers
- Smart card industry is open to formal methods

Potential killer application for formal methods ?

But if we can't even do this . . .

VerifiCard Project

- EU-sponsered IST-project of 2 smartcard producers and 5 academic research groups in tool-assisted verification for Java(Card).
- Work on
  - formal descriptions of platform (JCVM, byte code verifier) for CC evaluations
  - applet verification
  - non-interference and information-flow properties
  - case studies (banking, GSM) provided by industrial partners.

VerifiCard: Topics

JavaCard	platform	applets
byte code	VM, bytecode verifier, compiler formalisation	abstract interpretation & model checking
source code	API annotation in JML & Hoare-style verification	applet annotation in JML & Hoare-style verification

VerifiCard: other partners

Academic:

INRIA	Barthe, Bertot	France
TU Munich	Nipkow	Germany
Univ. Hagen	Poetzsch-Heffter	Germany
SICS	Dam	Sweden

Industrial:

Gemplus	Lanet	France
Schlumberger CP8	Goire	France

The LOOP project

The LOOP Project

Verification of JML-annotated Java(Card) programs based on

- a denotational semantics for sequential Java, formalised in PVS
- a compiler – the LOOP tool – which translates `A.java` to `A.pvs` describing its semantics.
- a logic for reasoning about JML, formalised in PVS

ie. a shallow embedding of Java and JML in PVS

LOOP tool for Java/JML

JML-annotated Java class A.java

$\Downarrow$

LOOP compiler

$\Downarrow$

meaning of Java class $[\![A]\!]$.pvs with
proof obligations as Hoare sentences

$\Downarrow$

PVS theorem prover

$\Downarrow$

QED

The LOOP Project: results so far

- translation covers essentially all sequential Java
- translation of JML under construction, but covers basics.
- case studies:
  - non-trivial invariant for Java's Vector class
  - AID class from JavaCard API
  - Purse applet (under development)
  - large collection of smaller test examples

Rest of this talk

- Semantics of Java
- Java specification language JML
- Hoare logic for Java/JML
- JML specifications of JavaCard API

Semantics of Java

Java Semantics

Standard denotational semantics of imperative program P:

$$S \xrightarrow{[\![P]\!]} 1 + S$$

where S is the state space and $1 = \{\perp\}$ stands for nontermination.

But: Java has abrupt termination because of

- exceptions, `throw(E)`
- `return` that exits a method
- `break` that exits a repetition
- `continue` that skips remainder of a repetition

so the semantics becomes more complicated . . .

Example: Java control flow

```
public int arrayProduct(int[] a)
{
  if (a == null) throw new MyNullPointerException(
  if (a.length == 0) return 1;
  int prod = 1;
  for (int i=0; i < a.length; i++){
       if (a[i]==0) {prod = 0; break;} ;
       if (a[i]==1) continue;
       prod = prod * a[i];
  };
  return prod;
}
```

Java semantics: statements

Semantics of Java statement P:

$$S \xrightarrow{[\![P]\!]} 1 + S + \text{StatAbn}$$

where

$$\begin{array}{lll} \text{StatAbn} = & (S \times \text{RefType}) & \text{state with exception object} \\ & +S & \texttt{return} \\ & +(S \times (1 + \text{String})) & \texttt{break} \text{ with label} \\ & +(S \times (1 + \text{String})) & \texttt{continue} \text{ with label} \end{array}$$

Example: composition

For two statements

$$s_1, s_2 : S \longrightarrow 1 + S + \text{StatAbn}$$

the composition is defined as

$$\begin{array}{rl} (s_1 \, ; \, s_2) \cdot x = & \text{CASES } s_1 \cdot x \text{ OF } \{ \\ & \quad \text{hang} \longmapsto \text{hang} \\ & \quad \mid \text{norm}(x') \longmapsto s_2 \cdot x' \\ & \quad \mid \text{abnorm}(a) \longmapsto \text{abnorm}(a) \ \} \end{array}$$

In this way all Java constructs are translated into PVS.

Java semantics : expressions

Semantics of Java expression E e:

$$S \xrightarrow{[\![e]\!]} 1 + S \times E + \text{ExprAbn}$$

where

$$\text{ExprAbn} = (S \times \text{RefType}) \quad \text{state with exception object}$$

Example: Addition

For two int-expressions

$$e_1, e_2 : S \longrightarrow 1 + S \times Int + \text{ExprAbn}$$

addition is defined as $(e_1 + e_2) \cdot x =$

$$\begin{array}{l} \text{CASES } e_1 \cdot x \text{ OF } \{ \\ \quad \text{hang} \longmapsto \text{hang} \\ \mid \text{norm}(x', val_1) \\ \quad \longmapsto \text{CASES } e_2 \cdot x' \text{ OF } \{ \\ \qquad \text{hang} \longmapsto \text{hang} \\ \qquad \mid \text{norm}(x'', val_2) \longmapsto \text{norm}(x'', val_1 + val_2 \\ \qquad \mid \text{abnorm}(a') \longmapsto \text{abnorm}(a') \ \} \\ \mid \text{abnorm}(a) \longmapsto \text{abnorm}(a) \ \} \end{array}$$

LOOP Java semantics

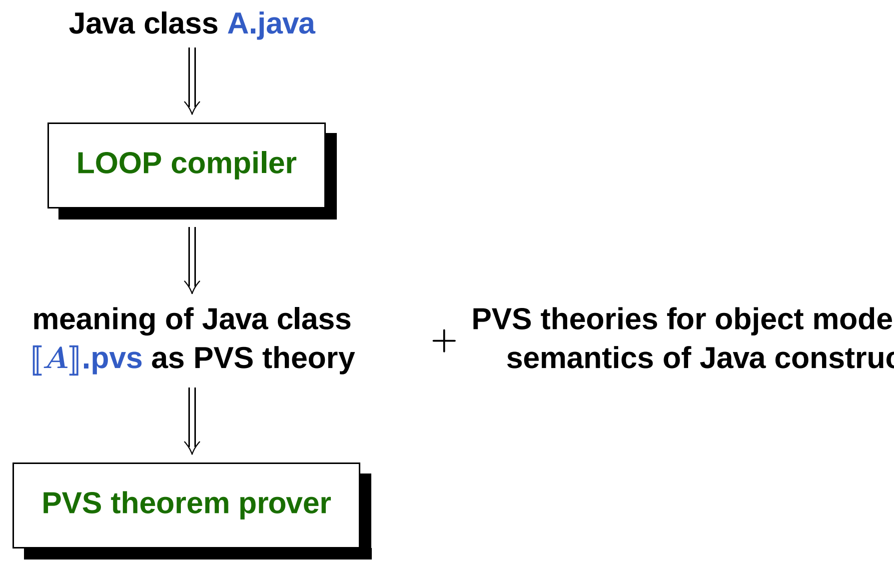
- representation of state space (or object model), mapping references to values
- semantics of all Java statements constructs:

```
;
if (...)  {...} else {...}
try {...} catch ... finally {...}
...
```

- semantics of Java expression constructs:
  `+ , - , ... , && , & , ... , x = ... , ...`
- semantics of classes incl. inheritance: method tables of statements/expression-valued functions, generated by the LOOP tool

Together covering essentially all of sequential Java

LOOP tool for Java



JML

Java Modeling Language JML

Specification language by Gary Leavens (Iowa Univ.) for annotating Java programs with

- pre- and postconditions } cf. Eiffel and
- invariants } Design by Contract
- frame conditions (modifiability constraints)
- specification-only variables (model variables)
- ...

Pre-, postconditions, and invariants in JML are Java boolean expressions, extended with `\forall`, `\exists`, `==>`, `\old( )`, ...

JML example

```
class A {
  int i;

  public void change_i(int j) throws MyException




   { if (j == 0) return ;

     if (i+j > MAX) throw new MyException();
     i = i+j;
   }
```


JML example

```
class A {
  int i;
   //@ invariant 0 <= i && i <= MAX;
  public void change_i(int j) throws MyException




   { if (j == 0) return ;

     if (i+j > MAX) throw new MyException();
     i = i+j;
   }
```


JML example

```
class A {
  int i;
   //@ invariant 0 <= i && i <= MAX;
  public void change_i(int j) throws MyException
   /*@    requires j >= 0;

           ensures i = \old(i)+j;

     @*/
   { if (j == 0) return ;

     if (i+j > MAX) throw new MyException();
     i = i+j;
   }
```

pre- and post-condition

JML example

```
class A {
  int i;
   //@ invariant 0 <= i && i <= MAX;
  public void change_i(int j) throws MyException
   /*@    requires j >= 0;

           ensures i = \old(i)+j;
           signals (MyException) i+j > MAX;
     @*/
   { if (j == 0) return ;

     if (i+j > MAX) throw new MyException();
     i = i+j;
   }
```

“exceptional” postcondition

JML example

```
class A {
  int i;
   //@ invariant 0 <= i && i <= MAX;
  public void change_i(int j) throws MyException
   /*@    requires j >= 0;

            ensures i = \old(i)+j;
            signals (MyException) i+j > MAX;
     @*/
   { if (j == 0) return ;
     //@ assert j!=0;
     if (i+j > MAX) throw new MyException();
     i = i+j;
   }
```

assertions in code

JML example

```
class A {
  int i;
   //@ invariant 0 <= i && i <= MAX;
  public void change_i(int j) throws MyException
   /*@    requires j >= 0;
        modifiable i;
            ensures i = \old(i)+j;
            signals (MyException) i+j > MAX;
     @*/
   { if (j == 0) return ;
     //@ assert j!=0;
     if (i+j > MAX) throw new MyException();
     i = i+j;
   }
```

NB just pre- and postconditions do not suffice!

JML example

```
class A {
  int i;
   //@ invariant 0 <= i && i <= MAX;
  public void change_i(int j) throws MyException
   /*@    requires j >= 0;
        modifiable i;
            ensures i = \old(i)+j;
            signals (MyException) i+j > MAX;
     @*/
   { if (j == 0) return ;
     //@ assert j!=0;
     if (i+j > MAX) throw new MyException();
     i = i+j;
   }
```


Tool support for JML

- **Iowa (Leavens et al.):**
  parser, typechecker, contract compiler inserting runtime checks for violations of assertions
- **MIT (Ernst):**
  Daikon tool for runtime detection of invariants
- **Compaq (Leino et al.):**
  extended static checker ESC/Java for automatic verification of simple assertions,
  e.g. no IndexOutOfBoundsExceptions**.**
- **Nijmegen:**
  LOOP tool as a front-end to theorem prover PVS for interactive verification of any assertion.

a logic for JML

- Hoare logic not at syntactic, but at semantic level:

  ie. not $\{P\}\ m\ \{Q\}$, but $\{P\}[\![m]\!]\{Q\}$

  But $[\![s_1; s_2]\!] = [\![s_1]\!] ; [\![s_2]\!]$, so proofs still syntax directed.

- Complicating factors in Java:
  - exceptions and other abrupt control flow
  - expressions can have side-effects

  Therefore
  - not Hoare triples but Hoare n-tuples,
  - for expressions as well as statements.

Hoare 4-tuples

Because of exceptions, instead of a Hoare triple

$$\{P\}\ m\ \{Q\} \quad \text{, in our notation: } \left(\begin{array}{rcl} \text{requires} & = & P \\ \text{statement} & = & m \\ \text{ensures} & = & Q \end{array}\right)$$

we need a Hoare 4-tuple $\left(\begin{array}{rcl} \text{requires} & = & P \\ \text{statement} & = & m \\ \text{ensures} & = & Q \\ \text{signals} & = & Q_{excp} \end{array}\right)$

including an exceptional postcondition Q_{excp}

Example Hoare 4-tuple

```
requires  =  j >= 0
statement =  [[ if (j == 0) return;
                if (i+j>MAX) throw new Exception(
                i = i+j; ]]
ensures   =  i = \old(i)+j
signals   =  i+j > MAX
```

> NB I leave out the invariant `0` $\leq$ `i` $\leq$ `MAX` and the modifiability constraint!

Hoare 5-tuples

Inside method bodies, apart from exceptions, also abrupt control flow via `return`: we need 5-tuples

$$\left(\begin{array}{rcl} \textsf{requires} & = & P \\ \textsf{statement} & = & s \\ \textsf{ensures} & = & Q \\ \textsf{signals} & = & Q_{excep} \\ \textsf{return} & = & Q_{ret} \end{array}\right)$$

Initially, Q_{ret} is equal to the postcondition Q.

Example Hoare 5-tuple

$$\left(\begin{array}{rcl} \textsf{requires} & = & \texttt{j} \geq 0 \\ \textsf{statement} & = & [\![\,\texttt{if (j == 0) return;} \\ & & \;\;\texttt{if (i+j>MAX) throw new Exception()} \\ & & \;\;\texttt{i = i+j;}\;]\!] \\ \textsf{ensures} & = & \texttt{i = \textbackslash old(i)+j} \\ \textsf{signals} & = & \texttt{i+j > MAX} \\ \textsf{return} & = & \texttt{i = \textbackslash old(i)+j} \end{array}\right.$$

Rule for Composition

$$\frac{\left(\begin{array}{rcl} \textsf{requires} & = & P \\ \textsf{statement} & = & s_1 \\ \textsf{ensures} & = & P' \\ \textsf{signals} & = & Q_{excp} \\ \textsf{return} & = & Q_{ret} \end{array}\right) \qquad \left(\begin{array}{rcl} \textsf{requires} & = & P' \\ \textsf{statement} & = & s_2 \\ \textsf{ensures} & = & Q \\ \textsf{signals} & = & Q_{excp} \\ \textsf{return} & = & Q_{ret} \end{array}\right)}{\left(\begin{array}{rcl} \textsf{requires} & = & P \\ \textsf{statement} & = & s_1; s_2 \\ \textsf{ensures} & = & Q \\ \textsf{signals} & = & Q_{excp} \\ \textsf{return} & = & Q_{ret} \end{array}\right)}$$

NB this is a lemma in PVS!

Example: applying composition rule

What predicate will hold here ? `j > 0`

$$\left(\begin{array}{rcl} \textsf{requires} & = & \texttt{j >= 0} \\ \textsf{statement} & = & [\![\,\texttt{if (j == 0) return ;} \\ & & \;\;\texttt{if (i+j>MAX) throw new Exception()} \\ & & \;\;\texttt{i = i+j;}\qquad]\!] \\ \textsf{ensures} & = & \texttt{i = \textbackslash old(i)+jj > 0} \\ \textsf{signals} & = & \texttt{i+j > MAX} \\ \textsf{return} & = & \texttt{i = \textbackslash old(i)+j} \end{array}\right.$$

Example: applying composition rule

$$\left(\begin{array}{rcl}
\text{requires} & = & \texttt{j > 0} \\
\text{statement} & = & [\![\\
 & & \quad \texttt{if (i+j>MAX) throw new Exception();} \\
 & & \quad \texttt{i = i+j;}\ ]\!] \\
\text{ensures} & = & \texttt{i = \textbackslash old(i)+j} \\
\text{signals} & = & \texttt{i+j > MAX} \\
\text{return} & = & \texttt{i = \textbackslash old(i)+j}
\end{array}\right)$$

Rule for if-then

$$\frac{\left(\begin{array}{rcl}
\text{requires} & = & P \\
\text{expression} & = & cond \\
\text{ensures}(b) & = & \text{IF } b \text{ THEN } P' \\
 & & \quad \text{ELSE } Q \\
\text{signals} & = & Q_{excp} \\
\text{return} & = & Q_{ret}
\end{array}\right)
\quad
\left(\begin{array}{rcl}
\text{requires} & = & P' \\
\text{statement} & = & s \\
\text{ensures} & = & Q \\
\text{signals} & = & Q_{exc} \\
\text{return} & = & Q_{ret}
\end{array}\right.}
{\left(\begin{array}{rcl}
\text{requires} & = & P \\
\text{statement} & = & \texttt{if}(cond)s; \\
\text{ensures} & = & Q \\
\text{signals} & = & Q_{excp} \\
\text{return} & = & Q_{ret}
\end{array}\right)}$$

Again, this is a lemma in PVS.

Hoare 7-tuples

Apart from exceptions and `return`'s, also abrupt control flow via `break` and `continue`:

$$\left(\begin{array}{rcl}
\text{requires} & = & P \\
\text{statement} & = & s \\
\text{ensures} & = & Q \\
\text{signals} & = & Q_{excep} \\
\text{return} & = & Q_{ret} \\
\text{continue} & = & Q_{cont} \\
\text{break} & = & Q_{break}
\end{array}\right)$$

Initially Q_{break} and Q_{cont} are false; they only get a value inside repetitions (namely the postcondition of the repetition and the invariant, resp.)

Rule for while

$$\frac{\left(\begin{array}{rcl}
\text{requires} & = & \text{inv} \\
\text{expression} & = & cond \\
\text{ensures}(b) & = & \text{IF } b \text{ THEN} \quad \text{inv'} \\
 & & \quad \text{ELSE} \quad Q \\
\text{signals} & = & Q_{excp}
\end{array}\right)
\quad
\left(\begin{array}{rcl}
\text{requires} & = & \text{in} \\
\text{statement} & = & s \\
\text{ensures} & = & \text{in} \\
\text{continue} & = & \text{in} \\
\text{break} & = & Q \\
\text{signals} & = & Q \\
\text{return} & = & Q
\end{array}\right.}
{\left(\begin{array}{rcl}
\text{requires} & = & \text{inv} \\
\text{statement} & = & \texttt{while}(cond)\{s\} \\
\text{ensures} & = & Q \\
\text{signals} & = & Q_{excp} \\
\text{return} & = & Q_{ret}
\end{array}\right)}$$

It gets a bit more complicated with labelled `break`'s and `continue`'s for nested repetitions . . .

Atomic statements

To prove properties of atomic statements, e.g.

$$\left(\begin{array}{rcl} \text{requires} & = & P \\ \text{statement} & = & [\![\texttt{obj.i=e}]\!] \\ \text{ensures} & = & Q \\ \text{signals} & = & Q_{excp} \\ \text{return} & = & Q_{ret} \end{array}\right)$$

we go back to the definition of Hoare n-tuples, i.e. we prove

$$\forall x.\ P(x) \Rightarrow \ \textsf{CASES}\ [\![\texttt{obj.i=e}]\!] \cdot x \quad \textsf{OF}\{ \\ \textsf{norm}(x') \mapsto \ Q(x') \\ \textsf{excp}(x', e) \mapsto \ Q_{excp}(x', e) \\ \textsf{ret}(x') \mapsto \ Q_{ret}(x') \ \}$$

Hoare logic for JML/Java

For more info, see FASE'2001.

Logic used to verify

- AID class from the JavaCard API (see JCW'2000).
- Purse applet (under construction)

Future work:

- better PVS strategies
- how to deal with invariants: when is it safe to assume an invariant holds ?
- how to handle the heap: ownership models ?

JML specifications for the JavaCard API

JML specs for the JavaCard API

The JavaCard API includes 47 classes (incl. Object, Throwable, NullPointerException, . . .) **of which 25 trivial.**

Formal JML specs are needed to verify applets that use the API, and to verify implementations of the API.

Basis for these specs: the detailed informal specs, and the reference implementation

So far we developed

- **lightweight specs for the whole API** (checked using ESC/Java)
- **complete specs for some classes** (incl. those needed to verify the AID class)

Serious case study in the use of JML (> 7000 lines of JML)

Lightweight specs

- Try to find a precondition that rules out all exceptions (or most of them).
- Take postcondition as weak as possible

  Usually just `true` but sometimes we may need `\result != null` or `\result > 0`

Example: Util.arrayCompare

```
public byte arrayCompare(byte[] src,  short srcOff
                         byte[] dest, short destOf
                         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;
   @*/
```


Example: Util.arrayCompare

```
/*@...
  @ also
  @ behavior
  @    requires true;
  @  modifiable nothing;
  @     ensures true;
  @     signals (NullPointerException)
  @           src == null || dest == null;
  @     signals (ArrayIndexOutOfBoundsException)
  @           srcOff < 0 ||
  @           srcOff + length > src.length ||
  @           ...
  @*/
```

but do we need this ?

Example: Util.arrayCopy

```
public short arrayCopy(byte[] src,  short srcOff,
                       byte[] dest, short destOff,
                       short length)
throws NullPointerException,
       IndexOutOfBoundsException,
       TransactionException;
 /*@ behavior
   @    requires ...
   @  modifiable dest[srcOff..srcOff+length-1];
   @     ensures true;
   @     exsures (TransactionException) true;
   @*/
```

Experience writing JML specs

The lightweight JML specs for the JavaCard API

- **often straightforward translations of informal specs to JML**
- **easy to read, write, and (informally) verify,** (though writing can take several iterations)
- **improve existing documentation, because of**
  - **precise declaration of exceptions**
  - **declaration of invariants**
- **Writing specs you (re)discover – and make explicit – some of the assumptions and considerations that have gone into the design of code**

(See CARDIS'00 and Computer Networks'01.)

Conclusions

Conclusions

Verification of JML-annotated Java(Card) programs using a shallow embedding of Java and JML in PVS, incl.

- **a denotational semantics for sequential Java**
- **a compiler which translates `A.java` to `A.pvs`**
- **a logic for JML**

Used in several case studies

- **invariant for Java's Vector class**
- **JavaCard's AID class**
- **Purse applet (under development)**

NB real programs, written in a real programming language

Future work

- **Covering more of JML, notably model variables**
- **More case studies**
- **More complete functional specs for the JavaCard API**
- **The big challenge: scaling up!**
  - **better PVS strategies ?**
  - **More modular/OO style verification ?**
    - **when can you assume an invariant holds ?**
    - **ownership models, . . . ?**
- **Looking at security properties** (confidentiality, integrity, . . .) **of JavaCard programs as part of larger systems**