WP-reasoning for Java

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. Phone Card Verification
II. Java Modeling Language JML
III. LOOP project at Nijmegen
IV. Hoare logic for Java & JML
V. WP for Java & JML
VI. Applet Case Study: Electronic Purse
VII. Future work & Conclusions
VIII. Demo (time permitting)

Context

- We’re pushing the technology in actual program verification …
- … for small, sequential Java programs …
- … for the latest generation of smart cards.

I. Phone Card Verification
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 structured byte sequences, called apdu’s.
- The applet uses two instance variables:

  ```java
  private byte value;
  private boolean issued;
  ```

Process implementation

```java
/** @ behavior 
  @
  @*/

public void process(APDU apdu) {
  byte[] buffer = apdu.getBuffer();
  if (value <= 0 && issued) {
    ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
  } else {
    switch( buffer[ISO7816.OFFSET_INS] ) {
      case INS_DECBAL: value--; break;
      case INS_GETBAL: apdu.setOutgoing();
      apdu.setOutgoingLength((short)1);
      buffer[0] = value;
      apdu.sendBytes((short)0,(short)1); break
      case INS_SETBAL: if (issued) {
        ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
      } else {
        value = buffer[ISO7816.OFFSET_P1];
        issued = true; }
      break;
    }
  }
}
```

Process specification

```java
/** @ behavior 
  @ requires apdu != null;
  @ modifiable value, issued, apdu.buffer[*];
  @ ensures \old(issued) ==> (issued &&
  @   value <= \old(value));
  @ signals (ISOException) \old(issued) &&
  @   (value <= 0 || apdu.buffer[
  @   ISO7816.OFFSET_INS] == INS_SETBAL)
  @   && issued == \old(issued)
  @   && value == \old(value);
  @ signals (APDUException)
  @   issued == \old(issued)
  @   && value == \old(value);
  @*/

public void process(APDU apdu) {
  ...
}
```

Correctness proof of process [30 min]
Expanded (WP-)proof of process

II. Java Modeling Language

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 <precondition 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 >= 0;
@ modifiable \nothing;
@ ensures \result * \result <= x &&
@ \result < (\result + 1) * (\result + 1)
@*/

int f(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:

- 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.

Integration of these tools is still a weak point.

III. LOOP project at Nijmegen

- 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, model variables (partially)
- 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).

IV. Hoare logic for Java & JML

Hoare logic and Weakest Preconditions

Reasoning in terms of the semantics of programs is not efficient. Therefore, special program logics are used.

- **Hoare logic** uses special triples
  precondition-program-postcondition \( \{ P \} s \{ Q \} \),
  with rules based on the structure of the program \( s \).
  Within a theorem prover, Hoare logic requires much user-interaction.

- **Weakest Precondition (WP)** functions \( wp(s, Q) \)
  compute such preconditions, such that:
  \[
P \Rightarrow wp(s, Q) \quad \text{iff} \quad \{ P \} s \{ Q \}
\]

Hoare logic issues for Java & JML

- 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

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

For \( \{ Pre \} m \{ Post \} \) write

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

For JML one needs:

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

Corresponding to Java’s termination options

Filled in only during proofs

### Hoare composition Rule

\[
\begin{align*}
\text{diverges} & = \lambda x, b \\
\text{requires} & = Pre \\
\text{statement} & = s_1 \\
\text{ensures} & = Q \\
\text{signals} & = P_{exc} \\
\text{return} & = P_{ret} \\
\text{break} & = P_{brk} \\
\text{continue} & = P_{cnt}
\end{align*}
\]

\[
\begin{align*}
\text{diverges} & = \lambda x, b \\
\text{requires} & = Pre \\
\text{statement} & = s_2 \\
\text{ensures} & = Post \\
\text{signals} & = P_{exc} \\
\text{return} & = P_{ret} \\
\text{break} & = P_{brk} \\
\text{continue} & = P_{cnt}
\end{align*}
\]

Intermediate predicate provided by the user

### Weakest Precondition issues

- Most recent addition: WP, integrated within Hoare logic for JML.
- Implemented as (provable) rules in PVS, used for automatic rewriting in special command (\texttt{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.

### V. WP for Java & JML
WP function definition

- WP for JML uses multiple result predicates:
  \[
  \text{wp}(\text{stat}, \begin{pmatrix}
  \text{diverges} &= D \\
  \text{ensures} &= \text{Post} \\
  \text{signals} &= \text{P}_{\text{exc}} \\
  \text{return} &= \text{P}_{\text{ret}} \\
  \text{break} &= \text{P}_{\text{brk}} \\
  \text{continue} &= \text{P}_{\text{cnt}}
  \end{pmatrix})
  \]
  \[
  \text{wp}(\text{expr}, \begin{pmatrix}
  \text{diverges} &= D \\
  \text{ensures} &= \text{Post} \\
  \text{signals} &= \text{P}_{\text{exc}}
  \end{pmatrix})
  \]
  and computes a weakest precondition.
- Aside: within PVS, this \text{wp} involves a higher-order definition.

WP rule for composition

\[
\text{wp}(s_1; s_2, \begin{pmatrix}
  \text{diverges} &= \lambda x, b \\
  \text{ensures} &= \text{Post} \\
  \text{signals} &= \text{P}_{\text{exc}} \\
  \text{return} &= \text{P}_{\text{ret}} \\
  \text{break} &= \text{P}_{\text{brk}} \\
  \text{continue} &= \text{P}_{\text{cnt}}
  \end{pmatrix}) =
\]

\[
\text{wp}(s_1, \begin{pmatrix}
  \text{diverges} &= \lambda x, b \\
  \text{ensures} &= \text{Post} \\
  \text{signals} &= \text{P}_{\text{exc}} \\
  \text{return} &= \text{P}_{\text{ret}} \\
  \text{break} &= \text{P}_{\text{brk}} \\
  \text{continue} &= \text{P}_{\text{cnt}}
  \end{pmatrix})
\]

WP rule for conditional and \\&\

\[
\text{wp}(e_1 && e_2, \begin{pmatrix}
  \text{diverges} &= \lambda x, b \\
  \text{ensures} &= \text{Post} \\
  \text{signals} &= \text{P}_{\text{exc}}
  \end{pmatrix}) =
\]

\[
\text{wp}(e_1, \begin{pmatrix}
  \text{diverges} &= \lambda x, b \\
  \text{ensures} &= \lambda x, \lambda r, \neg r \implies \text{Post} \cdot x \\
  \text{signals} &= \text{P}_{\text{exc}}
  \end{pmatrix})
\]
\[
\wedge
\]

\[
\text{wp}(e_1, \begin{pmatrix}
  \text{diverges} &= \lambda x, b \\
  \text{ensures} &= \lambda x, \lambda r, r \implies \text{post} \cdot \text{exc} \\
  \text{signals} &= \text{P}_{\text{exc}}
  \end{pmatrix})
\]

PVS strategies for WP

- Special PVS strategies examine the WP proof goal, and decide which rule to apply.
- They work in forward style: after the rule
  \[
  \text{wp}(s_1; s_2, \ldots) = \text{wp}(s_1, +++ \text{wp}(s_2, \ldots) +++)
  \]
  the WP-computation continues with \(s_1\).
- Ultimate success depends on “proof engineering”: clever management of large proof obligations, via appropriately timed simplifications.
- WP-rules for while and for loops require explicit instantiation for variant \\&\
i

WP for Java, Bart Jacobs (p.23 of 41)

WP for Java, Bart Jacobs (p.26 of 41)
Hoare & WP

In larger verifications,

- **Hoare rules** are interactively used to break up the proof goal, and
- **WP rules** are used for automatically completing these smaller goals.

VI. Applet Case Study: Electronic Purse

Purse example

- **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 from INRIA): numerous small bugs.

- Utility class **Decimal** verified with the **LOOP** tool:
  Several obvious bugs; complicated methods verified.

Purse example: Decimal class 1

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

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

  with fixed \text{PRECISION} = 1000.

- Decimal class invariant:

  ```
  /*@ invariant -\text{PRECISION} < \text{decPart} 
  && \text{decPart} < \text{PRECISION}; @*/
  ```
**Purse example: Decimal class 2**

- €1.35 can be
  - (intPart = 1, decPart = 350)
  - (intPart = 2, decPart = -650)

- Relevant value:
  \[ \text{intPart} \times \text{PRECISION} + \text{decPart} \]

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

**Purse example: add method**

```java
/** normal behavior */
@ requires -PRECISION < f && f < PRECISION;
@ modifiable intPart, decPart;
@ ensures intPart * PRECISION + decPart ==
@ (\old(intPart)+e)*PRECISION+\old(decPart)+f;
@*/

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

**Hoare logic proof tree in PVS**

**Weakest Precondition proof [45 min]**

```
(auto−rewrite "int_remainder")

(jml−start)

(wp−assert)
```
**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
  - Weakest precondition reasoning, integrated with annotated programs (proof outlines)

**Related work**

- **ESC/Java** (Leino et al.) is static checker which:
  - uses an (invisible) back-end prover
  - involves an intermediate guarded command language, on which the WP-rules work
  - is neither sound nor complete, but can trap many errors.

- **Jive** (Poetzsch-Heffter et al.) forms a Java proof environment which:
  - generates verification conditions for a separate theorem prover (PVS)
  - works on a limited part of Java.

**VII. Future work & conclusions**
Conclusions

- 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.
- Latest emphasis: from program correctness to program security
- Transfer to industry is happening: Gemplus, Schlumberger, Oberthur, TNO, . . .
- Actual certification case studies are soon possible.

More info at: **www.cs.kun.nl/~bart/LOOP**