A Logic for the Java Modeling Language JML

Bart Jacobs and Erik Poll

University of Nijmegen

FASE'2001

Outline

- the specification language JML
- JML verification using theorem provers
- Logic for JML, based on a denotational semantics of sequential Java

JML (Java Modeling Language)

Interface specification language for Java, by Leavens (Iowa State Univ).

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

- pre- and postconditions } as in *Eiffel*
- invariants } and *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; //@ invariant 0 <= i && i <= MAX;

  /*@   requires j >= 0;
      modifiable i;
         ensures i = old(i)+j;
         signals (MyException e) i+j > MAX;
    @*/
  public void change_i(int j) throws MyException {
    if (j == 0) { return ;} ;
    if (i+j > MAX) { throw new MyException(); } ;
    i = i+j;
    }
}
```

Tool support for JML

- Iowa: generating runtime checks to check for violations of preconditions and invariants (as in Eiffel).
- Compaq: extended static checker ESC/Java for *automatic* verification of *simple* assertions,
  e.g. no NullPointerExceptions.
- Nijmegen: verification, using the LOOP tool as a front-end to theorem provers (PVS, Isabelle) for *interactive* verification of *any* assertion.

  For this we have developed a logic for reasoning about JML-annotated Java programs in PVS.

The LOOP Project

Our basis for verification of JML-annotated Java:

- a denotational semantics for sequential Java.
- a compiler – the LOOP tool – which translates `A.java` to `A.pvs` describing its semantics.

Semantics of a method `m` is a function

$$[\![\mathtt{m}]\!] \in OM \longrightarrow 1 + OM + (Excp \times OM)$$

on some complicated state space OM.

Reasoning about $[\![\mathtt{m}]\!]$ is horrible: we want a 'Hoare'-logic for this . . .

NB

Two important points to note:

1. A Hoare logic not at syntactic level, but at **semantic level**:

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

   However, $[\![m]\!]$ is compositional: e.g. $[\![s1; s2]\!] = [\![s1]\!] \,;\, [\![s2]\!]$.

2. Complicating factors in a Hoare logic for Java:
   - exceptions and other abrupt control flow via `return`, `break`, `continue`.
   - expressions can have side-effects

   ie. not Hoare triples but Hoare **n-tuples**,
   and for **expressions** as well as **statements**.

Java control flow: methods

Because of exceptions, instead of a Hoare triple

$$\{P\}\ m\ \{Q\} \quad \text{, in our notation:} \quad \begin{pmatrix} \text{requires} & = & P, \\ \text{statement} & = & m, \\ \text{ensures} & = & Q \end{pmatrix}$$

we need a Hoare 4-tuple

$$\begin{pmatrix} \text{requires} & = & P, \\ \text{statement} & = & m, \\ \text{ensures} & = & Q, \\ \text{signals} & = & Q_{excp} \end{pmatrix}$$

including an exceptional postcondition Q_{excp}, for a method m.

Java control flow: statements inside methods

Inside method bodies, apart from exceptions, also abrupt control flow via `return`, `break`, `continue`

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

for a statement s.
Initially Q_{break} and Q_{cont} are false; they get a value inside repetitions.
Q_{ret} will be the postcondition of the method that contains s.

Translating JML to Hoare Logic

The LOOP tool translates a JML method specification

```
/*@   requires P;
      modifiable mod;
         ensures Q
         signals (E e) Qexcp
    @*/
  public void m ...
```

to PVS:

$$\left(\begin{array}{rcl} \text{requires} & = & \lambda x.\ [\![\mathtt{P}]\!] \cdot x, \\ \text{statement} & = & [\![\mathtt{m}]\!], \\ \text{ensures} & = & \lambda x.\ [\![\mathtt{Q}]\!] \cdot x, \\ \text{signals} & = & \lambda x.\ \lambda e.\ [\![e]\!]\ \text{instanceof}\ [\![\mathtt{E}]\!] \wedge [\![\mathtt{Qexcp}]\!] \cdot x \cdot e \end{array}\right)$$

Translating JML to Hoare Logic (2)

Any invariants `I` should be included:

$$\left(\begin{array}{rcl} \text{requires} & = & \lambda x.\ [\![\mathtt{I}]\!] \cdot x \wedge [\![\mathtt{P}]\!] \cdot x, \\ \text{statement} & = & [\![\mathtt{m}]\!], \\ \text{ensures} & = & \lambda x.\ [\![\mathtt{I}]\!] \cdot x \wedge [\![\mathtt{Q}]\!] \cdot x, \\ \text{signals} & = & \lambda x.\lambda e.\ [\![\mathtt{I}]\!] \cdot x \wedge [\![e]\!]\ \text{instanceof}\ [\![\mathtt{E}]\!] \wedge [\![\mathtt{Qexcp}]\!] \cdot x \cdot e \end{array}\right)$$

Translating JML to Hoare Logic (3)

Postconditions are not predicates, but are relations (because of `old`): we need a logical variable z to record the pre-state

$$\left(\begin{array}{rcl} \text{requires} & = & \lambda x.\ [\![\mathtt{I}]\!] \cdot x \wedge [\![\mathtt{P}]\!] \cdot x \wedge z = x, \\ \text{statement} & = & [\![\mathtt{m}]\!], \\ \text{ensures} & = & \lambda x.\ [\![\mathtt{I}]\!] \cdot x \wedge [\![\mathtt{Q}]\!] \cdot x \cdot z \wedge z \approx_{mod} x, \\ \text{signals} & = & \lambda x.\lambda e.\ [\![I]\!] \cdot x \wedge [\![e]\!]\ instanceof\ [\![\mathtt{E}]\!] \wedge [\![\mathtt{Qexcp}]\!] \cdot x \cdot e \cdot z \wedge \\ & & z \approx_{mod} x \end{array}\right)$$

where $\approx_{mod}$ is given by the modifiable clauses.

Proving JML specifications

Two ways to prove $\begin{pmatrix} \text{requires} & = & P, \\ \text{statement} & = & [\![\mathtt{m}]\!], \\ \text{ensures} & = & Q, \\ \text{signals} & = & Q_{excp} \end{pmatrix}$ in PVS

1. indirectly: using Hoare rules, following structure of $[\![\mathtt{m}]\!]$
2. directly: relying on its definition:

$$\forall x.\ P(x) \Rightarrow \ (\text{CASES } [\![\mathtt{m}]\!] \cdot x \text{ OF} \quad \begin{array}{rcl} \text{norm}(x') & : & Q(x') \\ \text{excp}(x', e) & : & Q_{excp}(x', e) \quad \text{ENDCASES}) \end{array}$$

Rule for Composition

$$\frac{\begin{pmatrix} \text{requires} & = & P, \\ \text{statement} & = & s_1, \\ \text{ensures} & = & P', \\ \text{signals} & = & Q_{excp} \\ \text{return} & = & Q_{ret}, \\ & \vdots & \end{pmatrix} \qquad \begin{pmatrix} \text{requires} & = & P', \\ \text{statement} & = & s_2, \\ \text{ensures} & = & Q, \\ \text{signals} & = & Q_{excp}, \\ \text{return} & = & Q_{ret}, \\ & \vdots & \end{pmatrix}}{\begin{pmatrix} \text{requires} & = & P, \\ \text{statement} & = & s_1; s_2, \\ \text{ensures} & = & Q, \\ \text{signals} & = & Q_{excp}, \\ \text{return} & = & Q_{ret}, \\ & \vdots & \end{pmatrix}}$$

This is a lemma in PVS.

Example

Omitting the invariant $0 \leq \mathtt{i} \leq \mathtt{MAX}$:

$$\begin{pmatrix} \text{requires} & = & \mathtt{j} \geq 0 \\ \text{statement} & = & [\![\quad \mathtt{if(j == 0)\{return;\};} \\ & & \quad\ \ \mathtt{if(i + j > MAX)\{throw\ new\ MyException();\};} \\ & & \quad\ \ \mathtt{i = i + j;}\ ]\!] \\ \text{ensures} & = & \mathtt{i} = \mathtt{old(i)} + \mathtt{j} \\ \text{signals} & = & \mathtt{i} + \mathtt{j} > \mathtt{MAX} \\ \text{return} & = & \mathtt{i} = \mathtt{old(i)} + \mathtt{j} \end{pmatrix}$$

$$\begin{pmatrix} \text{requires} & = & \mathtt{j} \geq 0 \\ \text{statement} & = & [\![\mathtt{if(j == 0)\{return;\}}]\!] \\ \text{ensures} & = & \mathtt{j} > 0 \\ \text{signals} & = & \mathtt{i} + \mathtt{j} > \mathtt{MAX} \\ \text{return} & = & \mathtt{i} = \mathtt{old(i)} + \mathtt{j} \end{pmatrix}$$

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

$$\left(\begin{array}{rcl} \text{requires} & = & \mathtt{j} > 0 \ \wedge\ \mathtt{i}+\mathtt{j} \leq \mathtt{MAX} \\ \text{statement} & = & [\![\mathtt{i} = \mathtt{i}+\mathtt{j}]\!] \\ \text{ensures} & = & \mathtt{i} = \mathtt{old(i)} + \mathtt{j} \\ \text{signals} & = & \mathtt{i}+\mathtt{j} > \mathtt{MAX} \\ \text{return} & = & \mathtt{i} = \mathtt{old(i)} + \mathtt{j} \end{array}\right)$$

Rule for While

$$\dfrac{\left(\begin{array}{rcl} \text{requires} & = & \text{inv} \\ \text{expression} & = & cond, \\ \text{ensures} & = & \lambda x.\lambda b. \\ & & \text{IF } b \text{ THEN } \ \text{inv}' \cdot x \\ & & \qquad\ \ \text{ELSE } \ Q \cdot x \\ \text{signals} & = & Q_{excp} \end{array}\right) \quad \left(\begin{array}{rcl} \text{requires} & = & \text{inv}', \\ \text{statement} & = & s, \\ \text{ensures} & = & \text{inv}, \\ \text{signals} & = & Q_{excp}, \\ \text{return} & = & Q_{ret}, \\ \text{break} & = & Q, \\ \text{continue} & = & \text{inv} \end{array}\right)}{\left(\begin{array}{rcl} \text{requires} & = & \text{inv}, \\ \text{statement} & = & \text{WHILE} \cdot cond \cdot s, \\ \text{ensures} & = & Q, \\ \text{signals} & = & Q_{excp}, \\ \text{return} & = & Q_{ret} \end{array}\right)}$$

Atomic statements

Properties of atomic statements, e.g. $\left(\begin{array}{rcl} \text{requires} & = & P, \\ \text{statement} & = & [\![\mathtt{obj.i} = \mathtt{e}]\!], \\ \text{ensures} & = & Q, \\ \text{signals} & = & Q_{excp}, \\ \text{return} & = & Q_{ret}, \\ & \vdots & \end{array}\right)$

are proved by going back its definition, i.e. by proving

$$\begin{array}{l} \forall x.\ P(x) \Rightarrow \ \ (\text{CASES } [\![\mathtt{obj.i} = \mathtt{e}]\!] \cdot x \text{ OF} \\ \qquad\qquad \begin{array}{rcl} \text{norm}(x') & : & Q(x') \\ \text{excp}(x', e) & : & Q_{excp}(x', e) \\ \text{ret}(x') & : & Q_{ret}(x') \\ & \vdots & \end{array} \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{ENDCASES}) \end{array}$$

using the definition of $[\![\mathtt{obj.i} = \mathtt{e}]\!]$ in PVS.

Proof rule for method invocation ?

We would like

$$\overline{\{Pre_m\}\ \texttt{o.m}(args)\ \{Post_m\}}$$

to reason about method invocation, but we get

$$\overline{\{Pre_m \wedge Inv(o)\}\ \texttt{o.m}(args)\ \{Post_m\}}$$

Wanted: some way of doing **modular verification**

Conclusions

- Logic for Java & JML, using 'Hoare 8-tuples' rather than Hoare triples
- at semantic level rather than syntactic level
- proved sound in PVS wrt. denotational semantics of sequential Java
- applied to simple case studies (AID.java from Java Card^{TM} API)

Future Work

- More case studies using this logic on Java(Card) programs
- including more of JML, notably model variables
- modular verification