Formal Methods for Java

Lecture 30: Conclusion

Jochen Hoenicke

Software Engineering
Albert-Ludwigs-University Freiburg

Feb 17, 2012

Topics

Lecture	Topics
1	Introduction to JML and JLS
2–3	Operational Semantics
4–5	JML
6–7	ESC/Java
8–10	Ownership/Friendship and Invariants
11–12	Sequent Calculus and Dynamic Logic
13–18	Proving with KeY
19–22	Jahob
23–29	Java Pathfinder

Motivations

Quality

- Leads to better understood code.
- Different view point reveals bugs.
- Formal proof can rule out bugs entirely.

Productivity

- Error detection in early stages of development.
- Modular specifications allow reuse of components.
- Documentation, maintenance.
- Automatic test case generation.
- Clearer specification leads to better software.

Operational Semantics for Java

Idea: define transition system for Java

Definition (Transition System)

A transition system (TS) is a structure $TS = (Q, Act, \rightarrow)$, where

- Q is a set of states,
- Act a set of actions,
- $\rightarrow \subseteq Q \times Act \times Q$ the transition relation.

- Q reflects the current dynamic state (heap and local variables).
- Act is the executed code.
- Idea from: D. v. Oheimb, T. Nipkow, Machine-checking the Java specification: Proving type-safety, 1999

State of a Java Program

The state of a Java program consists of a flow component and valuations for local and global (heap) variables.

- $Q = Flow \times Heap \times Local$
- $Flow ::= Norm | Ret | Exc\langle\langle Address \rangle\rangle$
- $Heap = Address \to Class \times \text{seq}\ Value$
- $Local = Identifier \to Value$
- $Value = \mathbb{Z}, Address \subseteq \mathbb{Z}$

A state is denoted as $q = (flow, heap, lcl)$, where $flow : Flow$, $heap : Heap$ and $lcl : Local$.

Rules of Operational Semantics

$$\frac{(Norm, heap, lcl) \xrightarrow{e_1 \triangleright v_1} q \quad q \xrightarrow{e_2 \triangleright v_2} q'}{(Norm, heap, lcl) \xrightarrow{e_1 * e_2 \triangleright (v_1 \cdot v_2) \bmod 2^{32}} q'}$$

$$\frac{(Norm, heap, lcl) \xrightarrow{st_1} q \quad q \xrightarrow{st_2} q'}{(Norm, heap, lcl) \xrightarrow{st_1 ; st_2} q'}$$

$$\frac{(Norm, heap, lcl) \xrightarrow{e \triangleright v} q \quad q \xrightarrow{bl_1} q'}{(Norm, heap, lcl) \xrightarrow{\textbf{if}(e)\, bl_1 \,\textbf{else}\, bl_2} q'}, \text{ where } v \neq 0$$

. . . and many more.

Rules for Exceptions

$$\frac{(Norm, heap, lcl) \xrightarrow{e \triangleright v} (Norm, heap', lcl')}{(Norm, heap, lcl) \xrightarrow{\textbf{throw}\ e} (Exc(v), heap', lcl')}$$

A null-pointer dereference works like a throw statement:

$$\frac{\begin{array}{c}(Norm, heap, lcl) \xrightarrow{e \triangleright 0} q' \\ q' \xrightarrow{\textbf{throw new}\ NullPointerException()} q''\end{array}}{(Norm, heap, lcl) \xrightarrow{e.fld \triangleright v} q''}$$

, where v is some arbitrary value

Propagating exceptions:

$$(flow, heap, lcl) \xrightarrow{\alpha} (flow, heap, lcl), \text{ where } flow \neq Norm$$

JML

Web pages

jmldoc

JML Annotated Java

```
public class ArrayOps {

  private /*@ spec_public @*/ Object[] a;

  //@ public invariant 0 < a.length;

  /*@ requires 0 < arr.length;
    @ ensures this.a == arr;
    @*/
  public void init(Object[] arr) {
    this.a = arr;
  }
}
```

Warnings

ESC/Java2

Daikon

Data trace file

jmlunit

Unit tests

jmlc

Class file

XVP

Bogor

Model checking

JACK, Jive, Krakatoa,
KeY, LOOP

Correctness proof

The Java Modelling Language (JML)

JML is a behavioral interface specification language (BISL) for Java

- Proposed by G. Leavens, A. Baker, C. Ruby:
  JML: A Notation for Detailed Design, 1999
- It combines ideas from two approaches:
  - Eiffel with it's built-in language for Design by Contract (DBC)
  - Larch/C++ a BISL for C++

Tools for JML

- `http://www.jmlspecs.org/`
- Release can be downloaded from `http://sourceforge.net/projects/jmlspecs/files`
- JML compiler (`jmlc`)
- JML runtime assertion checker (`jmlrac`)

External Tools:

- ESC/Java
- KeY
- and many more ...

Run-time Checking with jmlrac

Advantages of run-time checking:

- Easy to use.
- Supports a large sub-language of JML.
- No false warnings.

Disadvantages of run-time checking:

- Coverage only as good as test cases that are used.
- Does not prove absence of errors.

Static Checking with ESC/Java

Advantages of static checking:

- Easy to use.
- No test cases needed.
- Better coverage than runtime checking.
- Can detect missing specification.

Disadvantages of static checking:

- Only a small subset of JML supported.
- Many spurious warnings (not complete).

Theorem Proving with KeY

Advantages of static checking:

- Prove of correctness.
- Both sound and complete (modulo Peano Axioms).

Disadvantages of static checking:

- Very difficult to use.
- Can require interactive proving.

Model-checking with Java Pathfinder

Advantages of model-checking:

- Almost as easy as testing.
- More exhaustive than simple testing.

Disadvantages of model-checking:

- State explosion problem.
- Runtime vs. coverage.

Suggested order

1. Run-time checking, e.g. jmlrac and jmlunit.
2. Static checking, e.g. ESC/Java.
3. Model-checking, e.g. Java Pathfinder
4. Theorem proving, e.g. KeY.

Ensures that most bugs are already found before starting with theorem proving. Some prefer doing static checking before run-time checking (no test cases needed).

JML Keywords

Recall the meaning of the following Keywords:

- `requires`
- `ensures`
- `assignable`
- `signals`/`signals_only`
- `behavior`/`also`
- `normal_behavior`/`exceptional_behavior`
- `pure`
- `invariant`
- *`loop_invariant`*/*`decreases`*
- `nullable`/`non_null`
- `spec_public`
- `model`/`ghost`
- `represents`/`in`
- `assert`/`assume`

The Invariant Problem

```
public class SomeClass {
  /*@ invariant inv; @*/

  /*@ requires P;
    @ ensures Q;
    @*/
  public void doSomething() {

    assume(P);
    assume(inv);

    ...code of doSomething...

    assert(Q);
    assert(inv);

  }
}
```

```
public class OtherClass {
  public void caller(SomeObject o) {
    ...some other code...

    assert(P);

    o.doSomething();

    assume(Q);

  }
}
```

- Only sound if invariant cannot be invalidated.
- E.g., with the `pack`/`unpack` mechanism.

Sequent Calculus and Dynamic Logic

- $\phi_1, \phi_2 \Longrightarrow \psi_1, \psi_2$
- What are the rules of sequent calculus? Are they sound/complete?
- Hoare-Triples vs. $\phi \Longrightarrow \langle\alpha\rangle\psi$ and $\phi \Longrightarrow [\alpha]\psi$
- What is the meaning of $\langle\alpha\rangle\phi$?
- What are the rules for dynamic logic?
- What is the while_invariant_with_variant rule?

Jahob

- What is Jahob?
- Difference to ESC/Java?
- Difference to KeY?
- What does Jahob internally?
- How are the verification conditions generated?
- How are they checked?

Java Pathfinder

- What is Model-Checking?
- Difference to ESC/Java and Jahob?
- Difference to KeY?
- How can we write our own listeners?
- How can we use choice generators?
- What is partial order reduction?

What should you have learned

- How to give formal semantics to Java/JML (e.g. operational semantics).
- How to give pre-/post-conditions in JML.
- What is the relation between assume, assert and ensures, requires?
- What is run-time checking? Why is it useful? What are the limits?
- What is static checking (ESC/Java)? Why useful? What are the limits?
- What are the problems of class invariants and how to solve them.
- What is soundness and completeness? How does it apply to software verification.
- How to prove with KeY-System. How can loops be checked?
- How can verification conditions be generated from a program with assumes and asserts?
- How can these verification conditions be proven? Which tools exist?