Runtime assertion checking with JML

Erik Poll

Radboud University Nijmegen

Overview

Program verification is hard, let's just test!

- **Runtime assertion checking for JML**
- **Runtime assertion checking vs testing**
- **Demo: `jmlc/jmlrac` and `jmlunit`**
- **Runtime assertion checking vs program verification**
- **Semantics of `invariant` in JML**
- **Case study in using runtime assertion checking: Transacted Memory**

Runtime assertion checking

- Normally
  - Compile using `javac` to get `A.class`
  - Execute `A.class` using `javac`
- To do JML runtime assertion checking
  - Compile using `jmlc` to get `A.class`
  - Execute `A.class` using `jmlrac`

Actually, `jmlc` is a preprocessor for `javac`, and `jmlrac` a wrapper for `java`.

Runtime assertion checking

Only observable difference between using `jmlc/jmlrac` instead of `javac/java` (because JML annotations do not have side effects):

- **Program runs slower and uses more memory**
- **Program halts when any JML assertion (precondition, postcondition, invariant, . . .) is violated**

Typically, you use `jmlc/jmlrac` when testing code.

Benefit (we hope) : More errors detected, with less effort

jmlrac vs conventional testing (1)

- **More properties are tested, at more points in time, providing better feedback**

  Eg. "**`Invariant violated in line 20000`**"

  after 1 minute instead of

  "**`NullPointerException in line 60000`**"

  after 4 minutes

  Information about cause of problem, rather than the consequence.

- **Some testcode generated automatically by `jmlc`.**

  Eg. when you use **`\old`** in postcondition.

jmlrac vs conventional testing (2)

- **Less time needed to think about what to test.**
  If you have rich specs, to test you only need to provide inputs and not the expected response.
- **Investing in assertions can be better than investing in test-code:**
  - assertions can be developed earlier,
  - assertions are easier to maintain,
  - assertions also useful for other purposes (esp. documentation)
- **Writing JML assertions make you think about testability of the code in an early stage**
  eg. by adding pure method to use in specs

Demo

jmlrac vs escjava

Essentially, the pros and cons of 'testing' compared to 'program verification'

- **– need for executable code**
- **– need for testcode and testcases**
- **– less confidence**

- **+ no need to be complete in your specs**
  - **no need for API specs**
  - **no need for assignable specs**
- **+ fewer false negatives**

But still some 'false negatives': jmlrac may still complain where code is 'ok', but escjava will too in these cases.
Cause: the (strong) semantics of `invariant`

Semantics of `invariant`

The semantics of `invariant` in JML is more complicated ('stronger') than expected:

All invariants have to hold

- **at the end of constructor**
- **at the beginning and end of methods**
- **in methods and constructors, at point of method call**

  This is needed because of possible callbacks

NB all invariants of all objects, not just the invariants of the current object.

NB impossible to check or prove this exhaustively:

- **jmlrac only checks invariants of some objects,**
- **escjava only proves invariants of some objects**

jmlrac ‘false negatives’

Typical cases where the strong notion of invariant causes problems:

- **method called when invariant is temporarily broken**
  marking method as `helper` method can help
- **method called in constructor, ie. *before* invariants hold**

- **invariant involving multiple objects**

The same cases can cause problems when using ESC/Java.