The JML Tool

Néstor Cataño

ncatano@puj.edu.co

Faculty of Engineering

Pontificia Universidad Javeriana

Tools for JML

1. Parsing and type-checking
2. Checking assertions at runtime
3. Checking assertions at compile-time
4. Generating specifications
5. Documentation

JML Libraries

- JML comes with an libraries describing theories in mathematics
  - Sets, sequences, and relations
- These libraries are similar to libraries found in OCL and Z, but concepts are burnt directly as specifications

Side-Effects Free in Assertions

- JML prohibits side-effects in assertions, *e.g.*, expressions formed with the aid of **+=**, **-=**
- A method can be used in assertions only if it is declared as **pure**
- *e.g.*, if declared as `/*@ pure @*/ int getBalance() { ... }`, a `getBalance()` method can be used instead of the field `balance`

Parsing and Type-checking: `jml`

- `jml` performs parsing and type-checking of JML-specified Java code
- This provides an advantage over informal comments
  - typos, type incompatibilities, references to names that no longer exist, etc.

Generating Specifications

- The **`Daikon`** tool infers likely invariants by observing the runtime behaviour of a program
- The **`Houdini`** tool postulates annotations for code, then uses ESC/JAVA to check them
- The **`jmlspec`** tool can produce a skeleton of a specifications from Java source

Documentation: jmldoc

jmldoc produces HTML code from JML code in the style of **javadoc**

```
public class IntMathOps4 {
 /** Integer square root function.
   * @param y the number to take the root of
   * @return an integer approximating the positive square root of y
   * <pre><jml>
   *  public normal_behavior
   *    requires y >= 0;
   *    assignable \nothing;
   *    ensures 0 <= \result &&\result * \result <= y &&
   *              y < (\result + 1) * (\result + 1));
   * </jml></pre>
   **/
  public static int isqrt(int y){ return (int) Math.sqrt(y); }
}
```

Runtime Assertion Checking

- Are the specifications or the code incorrect?
  - Information on what parts of the specification were violated
  - Information on where in the program the violation was detected
  - backtrace to the source of the violation

Runtime Assertion Checking: `jmlc`

- It translates assertions into runtime checks
- It throws an exception every time an assertion is violated at runtime
- It is transparent when no assertions are violated
- It gives a warning for non-executable specifications, *e.g.*, `\fresh`

Runtime Assertion Checking: `jmlc`

- **`jmlc`** is incomplete, *i.e.*, it can fail to report assertions that are false
  - *e.g*, JML allows a way to write informal descriptions in assertions
- **`jmlc`** is sound, *i.e.*, it does not generate false reports

Runtime Assertion Checking: `jmlc`

Methodology for using `jmlc`

- To specify `normal_behavior` method preconditions
  - This ensures that all methods are called in expected states
- To add `toString()` methods to all classes
- To define invariants that define legal states of objects for each class
- To specify `normal_behavior` methods postconditions
- To specify exceptional postconditions for methods

Testing: `jmlunit`

- Runtime assertion checking can be used as part of a testing phase
- **`jmlunit`** combines runtime assertion checking and unit testing at the style of **`JUnit`**

Testing: `jmlunit`

- **`jmlc Foo.java`** produces bytecode with checks for preconditions, postconditions, invariants, etc.
- **`jmlunit Foo.java`** generates
  - **`Foo_JML_TestData.java`**, in which test data is to be placed for the types used as arguments to the methods in **`Foo.java`**
  - **`Foo_JML_Test.java`**, which implements **`JUnit`** tests for all methods in **`Foo.java`**
- Users can supply handwritten **`JUnit`** test methods if desired

Testing: `jmlunit`

- The JML assertions, as compiled by **`jmlc`**, are used to decide whether the **`JUnit`** tests, generated by **`jmlunit`**, succeed or fail

Testing: `jmlunit`

- The JML assertions, as compiled by **`jmlc`**, are used to decide whether the **`JUnit`** tests, generated by **`jmlunit`**, succeed or fail

The quality of testing provided by **`jmlunit`** is s good as the quality of the JML specifications

The JML Runtime Assertion Checker `jmlrac`

- It checks for runtime violations
  - **`Foo_JML_Test.java`** is compiled with a standard Java compiler, then
  - **`jmlrac Foo_JML_Test`** is used

Testing: `jmlunit`

- A formal specification can be viewed as a test oracle, and JML's runtime assertion checker can be used as the decision procedure for the test oracle
- **`jmlunit`** implements this idea, combining runtime assertion checking and unit testing
- **`jmlunit`** frees the programmer from writing **`JUnit`** code that decides whether unit tests pass or fail

Checking Assertions at Compile-Time

- More ambitious than testing if the code satisfies the specifications at runtime is verifying that the code satisfies its specification statically
- Verification provides more assurance on the correctness of the specification *w.r.t* the code
  - for all possible execution paths ...
- Runtime assertion checking is limited by the execution paths exercised by the test suite being used

`Purse` Demo

```
public class Purse {
  public final int MAX_BALANCE;
  public int balance;
  public byte[] pin;

  public int debit(int amount) throws PurseException {
    if (amount <= balance) { balance -= amount; return balance; }
    else { throw new PurseException("overdrawn by " + amount); }
  }

  public boolean checkPin(byte[] p) {
   boolean res = true;
   for (int i=0; i < 4; i++) { res = res && pin[i] == p[i]; }
   return res;
  }

  public Purse(int mb, int b, byte[] p) {
   MAX_BALANCE = mb; balance = b; pin = (byte[]) p.clone();
  }
}
```

`Decimal` Demo

```
public class Decimal {
 public static final short MAX_DECIMAL_NUMBER = (short) 32767;
 public static final short PRECISION = (short) 1000;

 public Decimal() {
   intPart = (short) 0;
   decPart = (short) 0;
 }

 public Decimal(short i, short d) throws DecimalException {
   try{ this.setValue(i,d); }
   catch(DecimalException e){
    System.out.println("Error initialising object of type Decimal");
   }
 }
 ...
```

`Decimal` Demo

```
  ...

  public short getIntPart(){ return intPart; }

  public short getDecPart(){ return decPart; }

  public Decimal setValue(short i, short d) throws DecimalException {
    if(i<0 || d<0 || d>=PRECISION || (i==MAX_DECIMAL_NUMBER && d!=0))
      throw new DecimalException();

    intPart = i;
    decPart = d;
    return this;
  }

  public void add(short e, short f) { ... }

  public Decimal mul(Decimal d) throws DecimalException { ... }
}
```

Checking Assertions at Compile-Time

- Correctness of a program with respect to a given specification is not decidable in general
  - Trade off between level of automation of whole checking/verification process and expressibility of specifications
- Static checking and verification JML-based tools
  - ESC/JAVA
  - Jack
  - LOOP

Static Checking Vs Runtime Checking

- Static checkers check specification at compile-time
  Static checkers prove correctness of specifications

Static Checking Vs Runtime Checking

- Static checkers check specification at compile-time
  Static checkers prove correctness of specifications
- Runtime assertion checkers check assertions at run-time
  Runtime assertion checkers test correctness of specifications

Static Checking Vs. Runtime Checking

One of these assertions is wrong:

```
if (i <= 0 || j < 0) {
  ...
}
else if (j < 5) {
//@ assert i > 0 && 0 < j && j < 5;
  ...
}
else {
//@ assert i > 0 && j > 5;
  ...
}
```

JML may detect this with a comprehensive test suite.

An static checker will (or will not) detect this at compile-time.