Introduction to JML

Erik Poll

Radboud University Nijmegen

Overview

- **The specification language JML**

  Only a subset, but this subset does cover the most used features of the language.

- **Some of the tools for JML, in particular**

  1. **runtime assertion checking using jmlc/jmlrac**
  2. **extended static checking using ESC/Java2**

- **Demo of ESC/Java2**

JML by Gary Leavens et al.

Formal specification language for Java

- to specify behaviour of Java classes
- to record design &implementation decisions

by adding **assertions** to Java source code, eg

- **preconditions**
- **postconditions**
- **invariants**

as in Eiffel (Design by Contract), but more expressive.

Goal: JML should be easy to use for any Java programmer.

JML

To make JML easy to use & understand:

- **Properties specified as comments in .java source file, between `/*@ ... @*/`, or after `//@`**
  (or in a separate file, if you don't have the source code, eg. of some API)
- **Properties are specified in Java syntax, namely as Java boolean expressions,**
  - **extended with a few operators (`\old`, `\forall`, `\result`, ...).**
  - **using a few keywords (`requires`, `ensures`, `invariant`, `pure`, `non_null`, ...)**

Example JML specification

```
public class IntegerSet {
    ...
byte[] a; /* The array a is sorted */




    ...
```

Example JML specification

```
public class IntegerSet {
    ...
byte[] a; /* The array a is sorted */
/*@ invariant
      (\forall int i; 0 <= i && i < a.length-1;
                      a[i] < a[i+1]);
  @*/
    ...
```

Informal vs Formal

The informal comment "The array a is sorted" and formal JML invariant

```
(\forall int i; 0 <= i && i < a.length-1;
                    a[i] < a[i+1])
```

document the same property, but

- **JML spec has a precise meaning. (Eg. < not <=)**
- **Precise syntax & semantics allows tool support:**
  - **runtime assertion checking: executing code and testing all assertions *for a given set of inputs***
  - **verification: proving that assertions are never violated, *for all possible inputs***

The JML specification language

Running example

```
public class BankAccount {
  final static int MAX_BALANCE = 1000;
  int balance;

  int debit(int amount) {
    balance = balance - amount;
    return balance; }
  int credit(int amount) {
    balance = balance + amount;
    return balance; }
  public int getBalance(){ return balance; }
  ...
```

requires

Pre-condition for method can be specified using requires:

```
/*@ requires amount >= 0;
  @*/
 public int debit(int amount) {
    ...
 }
```

Anyone calling `debit` has to guarantee the pre-condition.

ensures

Post-condition for method can be specified using ensures:

```
/*@ requires amount >= 0;
    ensures  balance == \old(balance)-amount &&
             \result == balance;
  @*/
public int debit(int amount) {
  ...
}
```

Anyone calling `debit` can assume postcondition (if method terminates normally, ie. does not throw exception)

`\old(...)` has obvious meaning

Design-by-Contract

Pre- and postcondition define a contract between a class and its clients:

- **Client must ensure precondition and may assume postcondition**
- **Method may assume precondition and must ensure postcondition**

Eg, in the example specs for `debit`, it is the obligation of the client to ensure that `amount` is positive. The `requires` clause makes this explicit.

requires, ensures

JML specs can be as strong or as weak as you want.

```
/*@ requires amount >= 0;
    ensures  true;
  @*/
 public int debit(int amount) {
   ...
 }
```

Default postcondition "`ensures true`" can be omitted.
Idem for default precondition "`requires true`".

invariant

Invariants (aka *class* invariants) are properties that must be maintained by all methods, e.g.,

```
public class  BankAccount {
  final static int MAX_BAL = 1000;
  int balance;
   /*@ invariant 0 <= balance &&
                   balance <= MAX_BAL;
     @*/
  ...
```

Invariants are implicitly included in all pre- and postconditions.

Invariants must *also* be preserved if exception is thrown!

invariant

Another example, from an implementation of a file system:

```
public class Directory {
private File[] files;
/*@ invariant
    files != null
    &&
    (\forall int i; 0 <= i && i < files.length;
                    files[i] != null &&
                    files[i].getParent() == this
  @*/
```

`invariant`

- **Invariants often document important design decisions.**
- **Making them explicit helps in understanding the code.**
- **Invariants often lead to pre-conditions:**
  Eg. in the `BankAccount` example, the precondition `amount <= balance` is needed to preserve the invariant `0 <= balance`

non_null

Many invariants, pre- and postconditions are about references not being `null`. non_null is a convenient short-hand for these.

```
public class Directory {

  private /*@ non_null @*/ File[] files;

  void createSubdir(/*@ non_null @*/ String name)
    ...
  Directory /*@ non_null @*/ getParent(){
    ...
```

assert

An `assert` clause specifies a property that should hold at some point in the code, e.g.,

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

assert

JML keyword `assert` now also in Java (since Java 1.4).

Still, assert in JML is more expressive, for example in

```
  ...
  for (n = 0; n < a.length; n++)
       if (a[n]==null) break;
/*@ assert (\forall int i; 0 <= i && i < n;
                                a[i] != null);
  @*/
```

signals

Exceptional postconditions can also be specified.

```
/*@ requires amount >= 0;
    ensures  true;
    signals (BankAccountException e)
              amount > balance          &&
              balance == \old(balance) &&
              e.getReason()==AMOUNT_TOO_BIG;
  @*/
  public int debit(int amount) { ...  }
```

The implementation given earlier does not meet this specification.

pure

A method without side-effects is called pure.

```
public /*@ pure @*/ int getBalance(){...
```

Pure methods – and only pure methods – can be used *in* JML specifications.

assignable

Frame properties limit possible side-effects of methods.

```
/*@   requires amount >= 0;
     assignable balance;
        ensures balance == \old(balance)-amount;
 @*/
public int debit(int amount) {
   ...
```

E.g., `debit` can *only* assign to the field `balance`.
NB this does *not* follow from the post-condition.

Assignable clauses are only needed to allow modular verification of code, by tools like ESC/Java(2).

Pure methods are `assignable \nothing`.

JML recap

The JML keywords discussed so far:

- `requires`
- `ensures`
- `signals`
- `invariant`
- `non_null`
- `pure`
- `\old`, code`\forall`, `\exists`, `\result`

This is all you need to know to get started!

Tools for JML

Tools for JML

A *formal* language allows *tool support*.

1. **Parsing and typechecking**
   Typos in JML specs are detected, typos in comments are not.
2. **Runtime assertion checking**
   test for violations of assertions during execution
   with the tool jmlrac
3. **Extended static checking ie. automated program verification**
   prove that contracts are never violated at compile-time
   with the tool ESC/Java2
4. **Interactive program verification:**
   more about that later

Runtime assertion checking

jmlrac compiler by Gary Leavens & Yoonsik Cheon

- **translates JML assertions into runtime checks:**
  during execution, *all* assertions are tested and any violation of an assertion produces an Error.
- **cheap & easy to do as part of existing testing practice**
- **better testing, because more properties are tested, at more places in the code**

Of course, an assertion violation can be an error in code or an error in specification.

The jmlunit tool combines jmlrac and unit testing.

Runtime assertion checking

jmlrac can generate complicated test-code for free. E.g., for

```
/*@ ...
    signals (Exception)
                balance == \old(balance);
  @*/
  public int debit(int amount) { ... }
```

it will test that if `debit` throws an exception, the balance hasn't changed, and all invariants still hold.

jmlrac even checks `\forall` if the domain of quantification is finite.

Extended static checking

ESC/Java(2) by Rustan Leino et al.

- ***tries* to *prove* correctness of specifications, at compile-time, fully automatically**
- ***not complete*: ESC/Java may warn of errors that can not occur, or time-out**
- ***not sound*: ESC/Java may miss an error that can occur**
- **but finds lots of potential bugs quickly**
- **good at proving absence of runtime exceptions (eg Null-, ArrayIndexOutOfBounds-, ClassCast-) and verifying relatively simple properties.**

Extended static checking vs runtime checking

Important differences:

- **ESC/Java2 checks specs at compile-time, jmlrac checks specs at run-time**
- **ESC/Java2 proves correctness of specs, jmlrac only tests correctness of specs.**
- **ESC/Java2 provides higher degree of confidence, but at a much higher price.**
- **Academics mainly interested in ESC/Java2, industrials mainly interested in jmlrac.**

Extended static checking vs runtime checking

One of the assertions below is wrong:

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

Runtime assertion checking *may* detect this with a comprehensive test suite.
ESC/Java2 *will* detect this at compile-time.

More JML tools

- javadoc-style documentation: jmldoc
- Eclipse plugin
- Other red verification tools:
  - LOOP tool + PVS (Nijmegen)
  - JACK (Gemplus/INRIA)
  - Krakatoa tool + Coq (INRIA)

  These tools (also) aim at interactive verification of complex properties, whereas ESC/Java2 aims at automatic verification of relatively simple properties.
- runtime detection of invariants: Daikon (Michael Ernst, MIT)
- model-checking multi-threaded programs: Bogor (Kansas State Univ.)

Related Work

- jContract tool for Java by Parasoft
- Spec# for C# by Microsoft
- SparkAda - subset of Ada by Praxis Critical Systems Ltd.
- OCL specification language for UML

Conclusions

- JML (relatively) easy to use and understand, using familiar syntax
- JML specs added to source code, so
  - easy to use incrementally
  - no need to construct a separate model
  - but. . . maybe lower level that other formal models

Some papers about (using) JML

- **Introduction to JML language:**
  Design by Contract with JML by Leavens and Cheon
- **An overview of JML tools and applications by lots of people**
- **Experience report about using JML:**
  Formal specification of the Java Card API in JML: the APDU class by Poll, van den Berg, and Jacobs
- **Experience report about using ESC/Java:**
  Formal specification of Gemplus's electronic purse case study by Cataño and Huisman