Program Verification using JML and ESC/Java2

Erik Poll

Radboud University Nijmegen

Outline of this tutorial

- **formal specifcation language JML**
- **program verification using ESC/Java2**

The Java Modeling Language
JML

`www.jmlspecs.org`

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:

- **JML assertions are added as comments in .java file, between `/*@` ... `@*/`, or after `//@`,**
- **Properties are specified as Java boolean expressions, extended with a few operators (\old, \forall, \result, ...).**
- **using a few keywords (`requires`, `ensures`, `signals`, `assignable`, `pure`, `invariant`, `non_null`, ...)**

requires, ensures

Pre- and post-conditions for method can be specified.

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

Here `\old(balance)` refers to the value of `balance` before execution of the method.

requires, ensures

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

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

This default postcondition “`ensures true`” can be omitted.

Design-by-Contract

Pre- and postconditions 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.

signals

Exceptional postconditions can also be specified.

```
/*@ requires amount >= 0;
    ensures  true;
    signals (BankException e)
              amount > balance          &&
              balance == \old(balance) &&
              e.getReason().equals("Amount too b
  @*/
 public int debit(int amount) throws BankExcepti
  ...
 }
```

signals

Exceptions mentioned in throws clause are allowed by default, i.e. the default signals clause is

```
signals (Exception) true;
```

To rule them out, add an explicit

```
signals (Exception) false;
```

or use the keyword normal_behavior

```
/*@ normal_behavior
      requires ...
      ensures  ...
  @*/
```

(class) invariant

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

```
public class  Wallet {
  public static final short MAX_BAL = 1000;
  private short 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!

(class) invariant

Invariants document design decisions, e.g.,

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

Making them explicit helps in understanding the code.

loop invariant

There are also loop invariants and variant functions:

```
 ...
//@ loop_invariant  0 <= i && i<= n;
//@ decreasing  n-i;
    for(int i=0; i<0; i++) {
          ...
    }
```

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);
  @*/
```

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.

Default assignable clause: assignable \everything.

pure

A method without side-effects is called pure.

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

Directory /*@ pure non_null @*/ getParent(){...}
```

Pure method are implicitly `assignable \nothing`.

Pure methods, and only pure methods, can be used *in* specifications, eg.

```
//@ invariant 0<=getBalance() && getBalance()<=MAX_BALANCE
```

JML recap

The JML keywords discussed so far:

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

This is all you need to know to get started!

Tools for JML

tools for JML

- parsing and typechecking
- runtime assertion checking:
  test for violations of assertions during execution
  jmlrac
- extended static checking ie. automated program verification:
  prove that contracts are never violated at compile-time
  ESC/Java2
  This is program verification, not just testing.

runtime assertion checking

jmlrac compiler by Gary Leavens, Yoonsik Cheon, et al. at Iowa State Univ.

- **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 and better feedback, because more properties are tested, at more places in the code**

  Eg, “Invariant violated in line 8000” after 1 minute instead of “NullPointerException in line 2000” after 4 minutes

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)

- extended static checking = fully automated program verification, with some compromises to achieve full automation

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.

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,
  jml only tests correctness of specs.
  Hence
  - ESC/Java2 independent of any test suite,
    results of runtime testing only as good as the test suite,
  - ESC/Java2 provides higher degree of confidence.

  The price for this: you have to specify all pre- and postconditions of methods (incl. API methods) and invariants needed for modular verification

The ESC/Java2 tool

Running ESC/Java2

- Download the binary distribution from http://secure.ucd.ie/products/opensource/ESCJava2
- Untar the distribution and follow the instructions in README.release about setting environment variables.
- Run the tool by doing one of the following:
  - Run a script in the release: escjava2 or escj.bat, or
  - Run a GUI version of the tool by double-clicking the release version of esctools2.jar

Command-line options

The items on the command-line are either options and their arguments or input entries. Some commonly used options (see the documentation for more):

- **-classpath - sets the path to find referenced classes [best if it contains '.']**
- **-nocheck - parse and typecheck but no verification**
- **-routine - restricts checking to a single routine**
- **-suggest - gives suggestion on how to fix problem**
- **-loopsafe - do verification of loops ; requires loop-invariants to be provided**

modular reasoning

ESC/Java2 reasons about every method individually. So in

```
class A{
 byte[] b;
 public void n() { b = new byte[20]; }
 public void m() { n();
                   b[0] = 2;
                   ...        }
```

ESC/Java2 warns that `b[0]` may be a null dereference here,

even though you can see that it won't be.

modular reasoning

To stop ESC/Java2 complaining: add a postcondition

```
class A{
 byte[] b;
 //@ ensures b != null && b.length = 20;
 public void n() { b = new byte[20]; }
 public void m() { n();
                    b[0] = 2;
                    ...          }
```

So: property of method that is relied on has to be made explicit.
Also: subclasses that override methods have to preserve these.

modular reasoning

Similarly, ESC/Java will complain about `b[0] = 2` in

```
class A{
 byte[] b;
 public void A() { b = new byte[20]; }
 public void m() { b[0] = 2;
                   ...  }
```

Maybe you can see that this is a spurious warning, though this will be harder than in the previous example: you'll have to inspect *all* constructors and *all* methods.

modular reasoning

To stop ESC/Java2 complaining here: add an invariant

```
class A{
 byte[] b;
  //@ invariant b != null && b.length == 20;
      // or weaker property for b.length ?
 public void A() { b = new byte[20]; }
 public void m() { b[0] = 2;
                   ...  }
```

So again: properties you rely on have to be made explicit.

And again: subclasses have to preserve these properties.

assume

Alternative to stop ESC/Java2 complaining: add an assumption:

```
...
//@ assume b != null && b.length > 0;
b[0] = 2;
...
```

Especially useful during development, when you're still trying to discover hidden assumptions, or when ESC/Java2's reasoning power is too weak.

(`requires` can be understood as a form of `assume`.)

ESC/Java is not complete

ESC/Java may produce warnings about correct programs.

```
/*@ requires 0 < n;
  @ ensures \result ==
  @              (\exists int x,y,z;
  @                pow(x,n)+pow(y,n) == pow(z,n));
  @*/
public static boolean fermat(double n) {
  return (n==2);
}
```

Warning: *postcondition possibly not satisfied*
(Typically, the theorem prover times out in complicated cases.)

ESC/Java is not sound

ESC/Java may fail to produce warning about incorrect program.

```
public class Positive{
    private int n = 1;  //@ invariant n > 0;

    public void increase(){ n++; }
}
```

ESC/Java(2) produces no warning, but `increase` may break the invariant, namely if n is $2^{32} - 1$.

This can be fixed by improved model of Java arithmetic, but this does come at a price (both in specs and in code).