Introduction to JML

Erik Poll, Joe Kiniry, David Cok

University of Nijmegen; Eastman Kodak Company

Outline of this talk

What this set of slides aims to do

- **introduction to JML**
- **provide overview of tool support for JML (jmlrac, jmlunit, escjava)**
- **explain idea of extended static checking and difference with runtime assertion checking**
- **some more ESC/Java2 tips**

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.

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 (ISOException e)
               amount > balance          &&
               balance == \old(balance) &&
               e.getReason()==AMOUNT_TOO_BIG;
 @*/
public int debit(int amount) {
 ...
}
```

signals

Exceptions 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  ...
  @*/
```

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!

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.

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`.

Only pure methods can be used *in* specifications.

visibility

JML supports the standard Java visibilities:

```
public int pub;  private int priv;

//@ requires i <= pub;
public void pub1 (int i) { ... }

//@ requires i <= pub && i <= priv;
private void priv1 (int i)  ...

//@ requires i <= pub && i <= priv; // WRONG !!
public void pub2(int i) { ... }
```

Specs of public methods may not refer to private fields.

visibility: spec_public

Keyword `spec_public` loosens visibility for specs. Private `spec_public` fields are allowed in public specs, e.g.:

```
public int pub;
private /*@ spec_public @*/ int priv;

//@ requires i <= pub && i <= priv; // OK
public void pub2(int i) { ... }
```

Exposing private details is ugly, of course. A nicer, but more advanced alternative in JML is to use public **`model`** fields to represent (abstract away from) private implementation details.

Tools for JML

tools for JML

- **parsing and typechecking**

tools for JML

- **parsing and typechecking**
- **runtime assertion checking:**
  test for violations of assertions during execution
  jmlrac

tools for JML

- **parsing and typechecking**
- **runtime assertion checking:**
  test for violations of assertions during execution
  jmlrac
- **extended static checking:**
  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 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.

runtime assertion checking

jmlrac compiler by Gary Leavens 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, because more properties are tested, at more places in the code**

runtime assertion checking

jmlrac compiler by Gary Leavens 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, 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*.

runtime assertion checking

jmlrac compiler by Gary Leavens 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, 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)

- ***tries* to *prove* correctness of specifications, at compile-time, fully automatically**

extended static checking

ESC/Java(2)

- *tries* to *prove* correctness of specifications, at compile-time, fully automatically
- *not sound*: ESC/Java may miss an error that is actually present

extended static checking

ESC/Java(2)

- *tries* to *prove* correctness of specifications, at compile-time, fully automatically
- *not sound*: ESC/Java may miss an error that is actually present
- *not complete*: ESC/Java may warn of errors that are impossible

extended static checking

ESC/Java(2)

- *tries* to *prove* correctness of specifications, at compile-time, fully automatically
- *not sound*: ESC/Java may miss an error that is actually present
- *not complete*: ESC/Java may warn of errors that are impossible
- but finds lots of potential bugs quickly

extended static checking

ESC/Java(2)

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

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.**

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.

modular reasoning (1)

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 (1)

To stop ESC/Java2 complaining: add a postcondition

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

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

modular reasoning (2)

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 (2)

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`.)

more JML tools

- javadoc-style documentation: jmldoc
- 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)

See `www.jmlspecs.org`

Acknowledgements

Many people and groups have contributed to JML and related tools.

- **Gary Leavens led the JML effort at Iowa St. Contributors include Albert Baker, Clyde Ruby, Curtis Clifton, Yoonsik Cheon, Anand Ganapathy, Abhay Bhorkar, Arun Raghavan, Kristina Boysen, David Behroozi. Katie Becker, Elisabeth Seagren, Brandon Shilling, Katie Becker, Ajani Thomas, and Arthur Thomas.**
- **The ESC project at SRC included K. Rustan M. Leino, Cormac Flanagan, Mark Lillibridge, Greg Nelson, Raymie Stata, and James Saxe.**
- **Bart Jacobs led the LOOP (now SoS) group at Nijmegen. Contributors include Erik Poll, Joachim van den Berg, Marieke Huisman, Cees-Bart Breunesse, and Joe Kiniry.**
- **David Cok is a primary contributor to JML and ESC/Java2.**

More information

These websites and mailing lists can provide more information (and have links to even more):

- **JML: www.jmlspecs.org**
- **mailing lists: jmlspecs-interest@lists.sourceforge.net jmlspecs-developers@lists.sourceforge.net**

- **ESC/Java2: www.cs.kun.nl/sos/research/escjava**
- **ESC/Java: www.research.compaq.com/SRC/esc/**
- **mailing list: jmlspecs-escjava@lists.sourceforge.net**