Introduction to JML

David Cok, Joe Kiniry, and Erik Poll

Eastman Kodak Company, University College Dublin, and Radboud University Nijmegen
Outline of this tutorial

First

- introduction to JML
- overview of tool support for JML, esp. runtime assertion checking (using jmlrac) and extended static checking ESC/Java2

Then

- ESC/Java2: Use and Features
- ESC/Java2: Warnings
- Specification tips and pitfalls
- Advanced JML: more tips and pitfalls

interspersed with demos.
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.
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.
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, ...
Pre- and post-conditions for method can be specified.

```java
/*@ 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.

```java
/*@ requires amount >= 0; 
  ensures true; 
@*/

class Account {
  public int debit(int amount) {
    ...
  }
}

This default postcondition “ensures true” can be omitted.
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**.
Exceptional postconditions can also be specified.

```java
/*@ requires amount >= 0;
   ensures true;
   signals (BankException e)
       amount > balance && balance == \old(balance) &&
       e.getReason().equals("Amount too big");
*/

public int debit(int amount) throws BankException
{
   ...
}
```
Exceptions mentioned in throws clause are allowed by default. To change this, there are three options:

- **To rule out all exceptions, use a** `normal_behavior`
  ```java
  /** @normal_behavior
   * requires ...
   * ensures ...
   */
  ```
  
- **To rule out particular exception** `E`, add
  ```java
  signals (E) false;
  ```
  
- **To allow only some exceptions**, add
  ```java
  signals_only E1, ..., E2;
  ```
Invariants (aka class invariants) are properties that must be maintained by all methods, e.g.,

```java
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!
Invariants document design decisions, e.g.,

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

```java
public class Directory {

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

    void createSubdir(/*@ non_null */ String name) {
        ...
        /*@ non_null */ Directory getParent() {
            ...
```
An `assert` clause specifies a property that should hold at some point in the code, e.g.,

```java
if (i <= 0 || j < 0) {
    ...
} else if (j < 5) {
    //@ assert i > 0 && 0 < j && j < 5;
    ...
} else {
    //@ assert i > 0 && j > 5;
    ...
}
```
JML keyword `assert` now also in Java (since Java 1.4).

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

```java
    for (n = 0; n < a.length; n++)
        if (a[n] == null) break;

    /*@ assert (\forall int i; 0 <= i && i < n;
                a[i] != null); @*/
```
Frame properties limit possible side-effects of methods.

```java
/*@ 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`. 

assignable
A method without side-effects is called pure.

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

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

Pure methods are implicitly assignable \nothing.

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

```java
//@ invariant 0<=getBalance() && getBalance()<=MAX_BALANCE
```
The JML keywords discussed so far:

- \texttt{requires}
- \texttt{ensures}
- \texttt{signals}
- \texttt{assignable}
- \texttt{normal\_behavior}
- \texttt{invariant}
- \texttt{non\_null}
- \texttt{pure}
- \texttt{\textbackslash old, \textbackslash forall, \textbackslash exists, \textbackslash result}

This is all you need to know to get started!
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 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.
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
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 “NullPointer.Exception in line 2000” after 4 minutes*
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.
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

```java
/*@ ... 
 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
extended static checking

ESC/Java(2)

- extended static checking = fully automated program verification, with some compromises to achieve full automation
- *tries to prove correctness of specifications, at compile-time, fully automatically*
extended static checking

ESC/Java(2)

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

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

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

- extended static checking = fully automated program verification, with some compromises to achieve full automation
- *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 (e.g. Null-, ArrayIndexOutOfBoundsException-, ClassCastException-) and verifying relatively simple properties.
ESC/Java(2) credits

- **ESC/Java** originally developed at DEC SRC – later Compaq, and now HP Research – by Rustan Leino, Cormac Flanagan, Mark Lillibridge, Greg Nelson, Raymie Stata, and James Saxe.

- **ESC/Java2**, extension that supports more of JML, developed by David Cok and Joe Kiniry.
static checking vs runtime checking

One of the assertions below is wrong:

```java
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.
more JML tools

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

These tools also allow interactive verification (whereas ESC/Java2 only aims at fully automatic verification) and can therefore handle more complex properties.

- runtime detection of invariants: Daikon (Michael Ernst, MIT)

- model-checking multi-threaded programs: Bogor (Kansas State)

See www.jmlspecs.org
Related Work

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

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

- Gary Leavens leads 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 Rustan Leino, Cormac Flanagan, Mark Lillibridge, Greg Nelson, Raymie Stata, and James Saxe.

- More people at many different places are contributing to JML.
More information

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

- **JML**: [www.jmlspecs.org](http://www.jmlspecs.org)
- **mailing lists**: jmlspecs-interest@lists.sourceforge.net
  jmlspecs-developers@lists.sourceforge.net

- **mailing list**: jmlspecs-escjava@lists.sourceforge.net