The Java Modeling Language (Part 1)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.jku.at

Overview
Since 1999 by Gary T. Leavens et al. (Iowa State University).
www.jmlspecs.org openjml.org
A behavioral interface specification language.
Syntactic interface and visible behavior of a Java module (interface/class).
Tradition of VDM, Eiffel, Larch/C++.
Fully embedded into the Java language.
Java declaration syntax and (extended) expression syntax.
Java types, name spaces, privacy levels.
JML annotations disguised as Java comments.
//@
/*@...*/

Related Work
Related to/influenced by/derived from JML (selection).
- C#: Spec# (Spec Sharp).
- Static checking (non-null types), runtime assertion checking.
- Verification condition generator (Boogie) for various prover backends.
- C: VCC and ACSL (ANSI C Specification Language).
  http://frama-c.com/acs1.html
  Microsoft VCC with SMT solver Z3 as backend.
  Frama-C ACSL framework with various prover backends.
- Ada: SPARK.
  http://www.adacore.com/sparkpro
  https://www.adacore.com/community
  VC generator and prover (GNATprove with CVC4, Z3, Alt-Ergo).

1. Basic JML
2. JML Tools
3. More Realistic JML
Basic JML

JML as required for the basic Hoare calculus.

- **Assertions.**
  - assume, assert.
- **Loop assertions.**
  - loop.invariant, decreases.
- **Method contracts.**
  - requires, ensures.
- **The JML expression language.**
  - \forall, \exists, ...
Pre- and post-condition define a **contract** between a method (i.e. its implementor) and its caller (i.e. the user).

- The method (the implementor) may **assume** the precondition and must **ensure** the postcondition.
- The caller (the user) must **ensure** the precondition and may **assume** the postcondition.
- Any method documentation must describe this contract (otherwise it is of little use).

The legal use of a method is determined by its contract (not by its implementation)!

---

**Method Contracts**

```java
/*@ requires y >= 0;
@ ensures \result >= 0
@ && \result*\result <= y
@ && y < (\result+1)*(\result+1); @*/
static int isqrt(int y)
{
    return (int) Math.sqrt(y);
}
```

- **requires** specifies the method precondition
- **ensures** specifies the method postcondition

Higher-level specification of a method.

---

**The JML Expression Language**

- **Atomic Formulas**
  - Any Java expression of type boolean: `a+b == c`
  - Primitive operators and pure program functions (later).
  - Informal property expression: `(* sum of a and b equals c *)`
  - Does not affect truth value of specification.
- **Connectives**: `!P, P&&Q, P || Q, P==Q, P<=Q, P<==Q, P<==!=Q`
- `negP, P && Q, P || Q, P == Q, P => Q, P <= Q, -(P <== Q)`
- **Universal quantification**: `\forall x \in T : P 
  \exists x \in T : P \land Q`
- **Existential quantification**: `\exists x \in T : P \land Q`

Strongly typed first-order predicate logic with equality.

---

**Postcondition and Pre-State**

```java
// swap a[i] and a[j], leave rest of array unchanged
/*@ requires
@ a != null &&
@ 0 <= i && i < a.length && 0 <= j && j < a.length;
@ ensures
@ a[i] = \old(a[j]) && a[j] == \old(a[i]) &&
@ (* all a[k] remain unchanged where k != i and k != j *) @*/
static void swap(int[] a, int i, int j)
{
    int t = a[i];
    a[i] = a[j];
    a[j] = t;
}
```

- **Variable values in postconditions**: `x...value of x in post-state (after the call).`
- Except for parameters which are always evaluated in the pre-state.
- `\old(x)...value of x in post-state (before the call).`
- `\old(E)...expression E evaluated with the value of every variable x in E taken from the pre-state.

Variable values may change by the method call (more on this later).
The JML Expression Language (Contd)

- **Sum**: \( \sum_{x \in T} x \land P \land U \)
- **Product**: \( \prod_{x \in T} x \land P \land U \)
- **Minimum**: \( \min \{ U : x \in T \land P \} \)
- **Maximum**: \( \max \{ U : x \in T \land P \} \)
- **Number**: \( | \{ x \in T : P \land Q \} | \)
- **Set**: new JMLObjectSet \( \{ T x | P \} \)

Examples

// sort array a in ascending order
/*@ requires a != null; @*/
static void sort(int[] a) {
    // ... }

// return index of first occurrence of x in a, -1 if x is not in a
/*@ requires a != null; @*/
static int findFirst(int[] a, int x) {
    // ... }

// swap a[i] and a[j], leave rest of array unchanged
/*@ requires a != null \&\& 
    0 <= i \&\& i < a.length \&\& 0 <= j \&\& j < a.length; 
@*/
static void swap(int[] a, int i, int j) {
    // ... }

1. Basic JML
2. JML Tools
3. More Realistic JML
Common JML Tools

- **Type checker jml**
  - Checks syntactic and type correctness.
- **Runtime assertion checker compiler jmlc**
  - Generates runtime assertions from (some) JML specifications.
- **Executable specification compiler jmle**
  - Generates executable code from (some) JML specifications.
- **JML skeleton specification generator jmlspec**
  - Generates JML skeleton files from Java source files.
- **Document generator jmldoc**
  - Generates HTML documentation in the style of javadoc.
- **Unit testing tool junit**
  - Generates stubs for the JUnit testing environment using specifications as test conditions.

Simple GUI launched by jml-launcher.

Example

```java
class Account {
    private /*@ spec_public @*/ int bal;
    ...
    //@ public invariant bal >= 0;
    //@ requires amt > 0 && amt <= bal;
   /*@ assignable bal; @*/
    public void withdraw(int amt) {
        bal -= amt;
    }
}
```

Runtime Assertion Checking

```bash
> jml -Q Account.java
> jmlc -Q Account.java
> jmlrac Account
Exception in thread "main"
    org.jmlspecs.jmlrac.runtime.JMLInternalPreconditionError: by method Account.withdraw
        at jmlrac.main(Account.java:1486)
```

A bit little information.

OpenJML

A replacement for the JML toolset (under development).

```bash
amir!27> openjmlrac Account.java
> java -jar /software/openjml/openjml.jar -rac Account.java
Note: /software/openjml/openjml.jar(specs17/java/util/Arrays.jml)
uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

```bash
amir!28> openjmlrun Account
java -cp /software/openjml/jmlruntime.jar:. Account
Account.Account.java:48: JML precondition is false
    acc.withdraw(200);

    //@ requires amt > 0 && amt <= bal;

    ...
Balance after withdrawal: -100
```

Other Third Party JML Tools

A large number of tools uses/supports JML.
- **Mobius Program Verification Environment**
  - Based on Eclipse, integrates common JML tools and ESC/Java2.
- **Sireum/Kiasan for Java**
  - Automatic verification and test case generation toolset.
- **Modern Jass**
  - Design by contract tool.
- **JMLUnitNG**
  - Test generation tool.
- **ESC/Java2**
  - Extends static checking (later).
- **KeY Verifier**
  - Computer-assisted verification (later).
- ...  

Support different versions of JML/Java, for current state, see http://www.jmlspecs.org/download.shtml

Practical Use

Recommended use with JML-annotated Java files.
- First compile with `javac`.
  - Check syntactic and type correctness of Java source.
- Then compile with `jml` (or `openjml`).
  - Check syntactic and type correctness of JML annotations.
- Then compile with `escjava2` (or `openjml -esc`).
  - Check semantic consistency of JML annotations.
  - More on ESC/Java2 later.

Errors can be made at each level.

More Realistic JML

JML for procedural programs with side-effects and errors.
- Side-effects
  - assignable, pure
- Exceptions
  - signals

We also have to deal with the less pleasant aspects of programs.
## Side Effects

```
static int q, r, x;
/*@ requires b != 0;
@ assignable q, r;
@ ensures a == b*q + r && sign(r) == sign(a) &&
@   (\forall int r0, int q0; a == b*q0+r0 && sign(r0) == sign(a);
@   abs(r) <= abs(r0)) */
static void quotRem(int a, int b)
{ q = a/b; r = a%b; }
```

- **assignable** specifies the variables that method may change.
- **Default:** assignable /\everything.
- **Possible:** assignable /\nothing.
- **No effect on any variable.**

## Pure Program Functions

```
static /*@ pure @*/ int sign(int x)
{
if (x == 0)
return 0;
else if (x > 0)
return 1;
else
return -1;
}
static /*@ pure @*/ int abs(int x)
{ if (x >= 0) return x; else return -x; }
```

Pure program functions may be used in specification expressions.

- **pure** implies assignable /\nothing.

JML considers pure program functions as mathematical functions.

## Arrays and Side Effects

```
ext static int[] a = new int[10];

@ assignable a;
- The pointer a may change.
  a = new int[20];

@ assignable a[\*];
- The content of a may change.
  a[1] = 1;
  // swap a[i] and a[j], leave rest of array unchanged
/*@ requires
  @ a != null &&
  @ 0 <= i && i < a.length && 0 <= j && j < a.length;
  @ assignable a[\*];
  @ ensures
    @ a[i] = \old(a[j]) && a[j] == \old(a[i]) &&
    @ (\forall int k; 0 <= k && k < a.length;
    @   (k != i && k != j) ==> a[k] == \old(a[k]));
*/
static void swap(int[] a, int i, int j) { ... }
```

## Exceptions

```
ext static int balance;
/*@ assignable balance;
@ ensures \old(balance) >= amount
@   && balance = \old(balance)-amount;
@ signals(DepositException e) \old(balance) < amount
@   && balance == \old(balance); */
static void withdraw(int amount) throws DepositException
{ if (balance < amount) throw new DepositException();
balance = balance-amount;
}
```

This method has two ways to return.

- **Normal return:** the postcondition specified by ensures holds.
- **Exceptional return:** an exception is raised and the postcondition specified by signals holds.
Exceptions

- **Default**: `signals(Exception e) true;`
  - Instead of a normal return, method may also raise an exception without any guarantee for the post-state.
  - Even if no `throws` clause is present, runtime exceptions may be raised.
- **Consider**: `signals(Exception e) false;`
  - If method returns by an exception, false holds.
  - Thus the method must not raise an exception (also no runtime exception).

We also have to take care to specify the exceptional behavior of a method!

Lightweight Specifications

This is the contract format we used up to now.

```*/@ requires ...; 
  @ assignable ...; 
  @ ensures ...; 
  @ signals ...; @*/
```

- Convenient form for simple specifications.
- If some clauses are omitted, their value is unspecified.

So what does a (partially) unspecified contract mean?

Preconditions versus Exceptions

```/*@ requires (\exists int x; ; a == x*b); 
  @ ensures a == \result*b; @*/
```

```static int exactDivide1(int a, int b) { ... }```

```/*@ ensures (\exists int x; ; a == x*b) && a == \result*b; 
  @ signals(DivException e) !(!\exists int x; ; a == x*b) @*/
```

```static int exactDivide2(int a, int b) throws DivException { ... }```

- `exactDivide1` has precondition $P : \equiv \exists x : a = x \cdot b$.
  - Method must not be called, if $P$ is false.
  - It is the responsibility of the caller to take care of $P$.
- `exactDivide2` has precondition true.
  - Method may be also called, if $P$ is false.
  - Method must raise `DivException`, if $P$ is false.
  - It is the responsibility of the method to take care of $P$.

Different contracts!

Method Underspecification

If not specified otherwise, **client** should assume weakest possible contract:

- `requires false;`
  - Method should not be called at all.
- `assignable \everything;`
  - In its execution, the method may change any visible variable.
- `ensures true;`
  - If the method returns normally, it does not provide any guarantees for the post-state.
- `signals(Exception e) true;`
  - Rather than returning, the method may also throw an arbitrary exception; in this case, there are no guarantees for the post-state.

Defensive programming: for safety, client should avoid implicit assumptions.
**Method Underspecification**

If not specified otherwise, **method should implement strongest possible contract**:
- **requires true**; Method might be called in any pre-state.
- **assignable \nothing**; In its execution, the method must not change any visible variable.
- **signals(Exception e) false**; Method should not throw any exception.

Defensive programming: for safety, method should satisfy implicit client assumptions (as far as possible).

---

**Heavyweight Specifications**

`/*@ public normal_behavior`  
@ requires ...;  
@ assignable ...;  
@ ensures ...;  
@ also public exceptional_behavior  
@ requires ...;  
@ assignable ...;  
@ signals(...) ...; @*/

- A normal behavior and (one or multiple) exceptional behaviors.
- Method must implement all behaviors.
- Each behavior has a separate precondition.
  - What must hold, such that method can exhibit this behavior.
  - If multiple hold, method may exhibit any corresponding behavior.
  - If none holds, method must not be called.
- For each behavior, we can specify the visibility level (later), the assignable variables, the postcondition.

---

**Heavyweight Specification Defaults**

If not specified otherwise, we have the following defaults:
- **requires true**; Method may be called in any state.
- **assignable \everything**; In its execution, the method may change every visible variable.
- **ensures true**; After normal return, no guarantees for the post-state.
- **signals(Exception e) true**; Rather than returning, the method may also throw an arbitrary exception; then there are no guarantees for the post-state.

Method must not make assumptions on the pre-state, caller must not make assumptions on the method behavior and on the post-state.

---

**Example**

```java  
static int balance;

/*@ public normal_behavior  
@ requires balance >= amount;  
@ assignable balance;  
@ ensures balance = \old(balance)-amount;  
@ also public exceptional_behavior  
@ requires balance < amount;  
@ assignable \nothing;  
@ signals(DepositException e) true; @*/

class Account {  
    static void withdraw(int amount) throws DepositException {  
        if (balance < amount) throw new DepositException();  
        balance = balance-amount;  
    }  
}

Clearer separation of normal behavior and exceptional behavior.