The Java Modeling Language JML

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

Wolfgang.Schreiner@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at/people/schreine

Table of Contents

Overview

- Since 1999 by Gary T. Leavens et al. (Iowa State)
  - `www.jmlspecs.org`
- A *behavioral interface specification language*
  - Specifies syntactic interface and visible behavior of a Java module (interface or class); tradition of VDM, Z, Larch/C++
- Fully embedded into the Java language
  - Java declaration syntax and (extended) expression syntax.
  - Types, name spaces, privacy levels.

Used by various groups, supported by various tools.

Annotation Statements

```
public class IntMathOps
{
  public static int isqrt(int y)
  {
    //@ assume y >= 0;
    int r = (int) Math.sqrt(y);
    //@ assert r >= 0 && r*r <= y && y < (r+1)*(r+1);
    return r;
  }
}
```

Obligations of caller and of implementor, respectively.

Function Specifications

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

Pre- and post-conditions.

Refinements

```
//@ refine “IntMathOps.java”

public class IntMathOps
{
  /*@ requires y >= 0;
    @ ensures
    @   0 <= \result && \result * \result <= y &&
    @   y < ((\result + 1) * (\result + 1));
    @*/
  public static int isqrt(int y);
}
```

Separate code from specifications, refine specifications.

Refinements

- Refinement chain (from least to most *active* one*)*

`Class.jml-refined`	*passive files (must not be passed to tools)*
`Class.spec-refined`	
`Class.java-refined`	
`Class.jml`	*active files (may be passed to tools)*
`Class.spec`	
`Class.java`	
`Class.refines-jml`	
`Class.refines-spec`	
`Class.refines-java`	

- When tool needs `Class`, it looks for base of chain.
  - File with most active suffix.
  - All files in sequence are loaded automatically.
  - All specifications of an entity are combined.

Procedure Specifications

```
public class Int
{
  int value, other;
  ...

  /*@ requires x != 0;
    @ assignable value;
    @ ensures value = \old(value)/x;
  public void div(int x)
  {
    value = value/x;
  }
}
```

Procedure must specify the variables it may change.

Exceptions

```
...

/*@ ensures x != 0 && value = \old(value)/x;
  @ assignable value;
  @ signals (DivByZeroException e) x == 0;
public void div(int x) signals DivByZeroException
{
  if (x == 0)
    throw new DivByZeroException();
  else
    value = value/x;
}
```

Normal behavior as well as exceptional behavior.

Heavy-Weight Specifications

```
/*@ public normal_behavior
  @   requires x != 0;
  @   assignable value;
  @   ensures value = \old(value)/x;
  @ public exceptional_behavior
  @   requires x == 0;
  @   assignable \nothing;
  @   signals (DivByZeroException e);
  @*/
  public void div(int x) signals DivByZeroException;
```

More expressive than light-weight specifications.

Specification Expressions

- Various logical and arithmetic quantifiers

```
/*@ requires
  @   a != null &&
  @   (\forall int i; 0 <= i && i < a.length-1;
  @               a[i] <= a[i+1]);
  @ ensures
  @    (\result == -1 &&
  @        (\forall int i; 0 <= i && i < a.length;
  @                   a[i] != x)) ||
  @    (0 <= \result && \result < a.length &&
  @         a[\result] == x));
  @*/
public static int binarySearch(int[] a, int x);
```

Specification Expressions

- Use of *pure* program functions

```
class IntStack
{
  ...

  public /*@ pure @/ boolean isEmpty() { ... }

  /*@ requires !isEmpty(); @*/
  public int pop() { ... }
}
```

Pure function must not change program state.

Class Specifications

- Class conditions

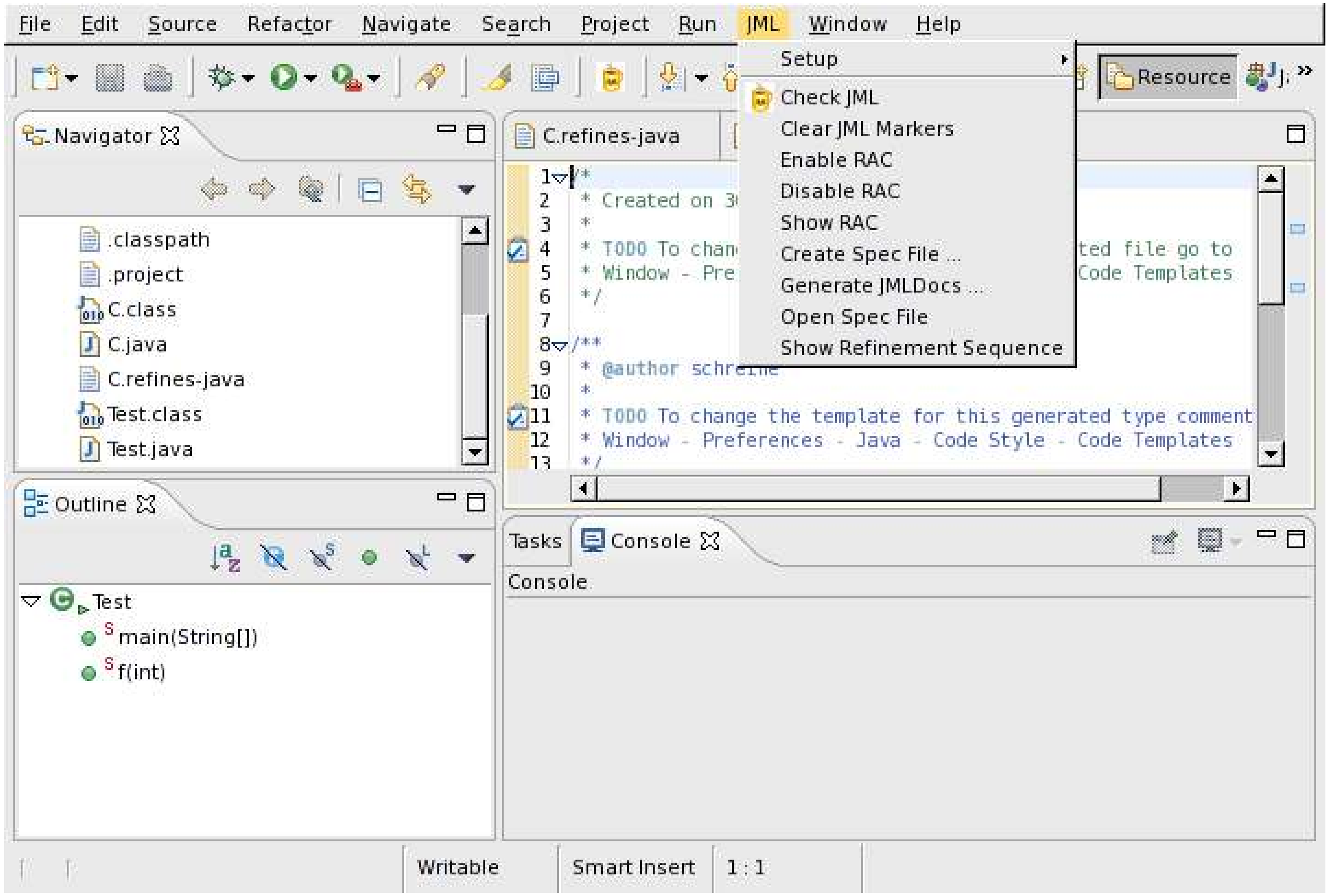
```
class IntStack
{
  int[] stack; // element container
  int n;       // number of elements in container

  //@ public initially n == 0;
  //@ public invariant 0 <= n && n <= stack.length;

  ...
}
```

Invariant true before and after each method invocation.

Tools

- Type checker `jml`
  - Checks syntactic correctness and type correctness.
- Runtime assertion checker compiler `jmlc`
  - Generates executable code with runtime assertions checking many specification conditions.
- Unit testing tool `junit`
  - Generates stub for *JUnit* testing environment using specification conditions as test conditions.
- Document generation `jmldoc`
  - HTML in the style of `javadoc`

Eclipse Plugin



```
/*
 * Created on 3
 *
 * TODO To chan                                    ted file go to
 * Window - Pre                                    Code Templates
 */

/**
 * @author schreine
 *
 * TODO To change the template for this generated type comment
 * Window - Preferences - Java - Code Style - Code Templates
 */
```

Model Variables

```
public class IntArray
{
  private int[] array;
  //@ public model int n;
  //@ private represents n <- array.length;

  /*@ ensures \result <=>
   *@   (\forall int i; 0 <= i; i < n)
   *@     array[i] == 0;
   *@/
  public boolean isZero() {...}
}
```

Specification-only variables.

Pure Model Functions

```
public class IntStack
{
  int[] stack; // element container
  int n;       // number of elements in container

  /*@ ensures \result <==> n == stack.length;
    @ public pure model boolean isFull();
    @/

  //@ requires !isFull();
  public void push(int x) { ... }
}
```

Specification-only functions.

Model Types

```
//@ model import org.jmlspecs.models.*;

public class ArraySet
{
  //@ public model JMLValueSet set;
  //@ public initially set != null && !set.isEmpty();
  ...

  //@ ensures set.equals(\old(set).insert(e));
  public void add(Object e) { ... } ;
}
```

Pure classes primarily used for modeling.

An Algebraically Specified Model Type

```
public /*@ pure @*/ class Stack
{
  //@ public model Stack();
  //@ public model boolean isEmpty();
  //@ public model Stack push(int e);
  //@ public model int top();
  //@ public Stack pop();

  /*@ public invariant
    @ (\forall Stack s; s != null;
    @    (\forall int e; ;
    @      new Stack().isEmpty() &&
    @      e == s.push(e).top()  &&
    @      s.equals(s.push(e).pop())));
    @*/
}
```

Further Features

- Expression syntax
  - Quantifiers, sets, ...
- Interface specifications
  - Properties of implementation classes.
- Redundant specifications
  - Properties implied by other properties:

    `ensures_redundantly`, `for_example,` ...
- Subtyping
  - Combining specifications of superclass and subclass;

    `also ensures`

Further Features

- Data groups
  - Relationship between model variable and group of program variables/locations: `//@ maps o.f \into mvar`
- History constraints
  - Constraint how variable may be changed by any method

    `//@ public constraint x = \old(x)`
- Non-functional properties of methods
  - Execution time, execution space, methods invoked, ...
- Concurrency
  - Support for MultiJava.

JML for Verification

- Extended static checker *ESC/Java2.*
  - Generates verification conditions, checks with *Simplify*.
- Program verification with *LOOP.*
  - Verification conditions for interactive *PVS* proofs.
- Static verification with *JACK.*
  - Generates verification conditions, checks with *B* toolkit.
- Invariant detection with *Daikon.*
  - Guesses program invariants from execution profiles.
  - Report: 90% correct, 90% of those needed for verification.

Literature

- Web-Site

  `www.jmlspecs.org`

- Short Paper
  - Leavens et al. Design by Contract with JML.
- Longer paper
  - Leavens et al. JML: A Notation for Detailed Design.
- Longer report
  - Leavens et al. Preliminary Design of JML: A Behavioral Interface Specification Language for Java.