

Java Modelling Language (JML) References

- G. T. Leavens and Y. Cheon. Design by Contract with JML, August 2005.
- L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An Overview of JML Tools and Applications. In *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.
- C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa Tool for Certification of Java/JavaCard Programs Annotated in JML. In *Journal of Logic and Algebraic Programming*, 58(1-2):89-106, 2004.
- www.jmlspecs.org
- G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. Department of Computer Science, Iowa State University, TR #98-06-rev28, July 2005.
- P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *Formal Methods for Components and Objects 2005, Revised Lectures*, pages 342-363, Springer Verlag LNCS Volume 4111, 2006.
- krakatoa.lri.fr

JML as a Design by Contract (DBC) Tool

- A “contract” between a program (Java class) and its clients.
- A *precondition* specifies the client’s (user of the program) obligation (i.e., guarantees that must be met before calling a method).
- A *postcondition* specifies the implementor’s obligation (i.e., guarantees properties that hold after execution).
- Contracts are “executable”, i.e., can be checked by tools.

Example Java Class

```
public class Person {
    private String name;
    private int weight;

    public String toString() {
        return "Person(\"" + name + "\", " + weight + ")";
    }

    public int getWeight() { return weight; }

    public void addKgs(int kgs) {
        if (kgs >= 0) { weight += kgs; }
        else { throw new IllegalArgumentException(); }
    }

    public Person(String n) { name = n; weight = 0; }
}
```

Specifying the addKgs Method

```
/*@ requires kgs >= 0;  
   @ requires weight + kgs >= 0;  
   @ ensures weight == \old(weight) + kgs;  
   @*/  
public void addKgs(int kgs) { weight += kgs; }
```

Alternate Specification and Implementation of addKgs

```
/*@ requires weight + kgs >= 0;
   @ ensures kgs >=0 && weight == \old(weight) + kgs;
   @ signals_only IllegalArgumentException;
   @ signals (IllegalArgumentException) kgs < 0;
   @*/
public void addKgs(int kgs) {
    if (kgs >= 0) { weight += kgs; }
    else { throw new IllegalArgumentException(); }
}
```

Information Hiding and Invariants

```
public class Person {
    private /*@ spec_public non_null @*/
        String name;
    private /*@ spec_public @*/
        int weight;

    /*@ public invariant !name.equals("")
        @          && weight >= 0; @*/
```

Remaining Code and Specifications of Person Class

```
//@ ensures \result != null;
public String toString() {
    return "Person(\"" + name + "\", " + weight + ")";
}

//@ ensures \result == weight;
public /*@ pure @*/ int getWeight() { return weight; }

/*@ requires n != null && !n.equals("");
   @ ensures n.equals(name)
   @   && weight == 0; @*/
public Person(String n) { name = n; weight = 0; }
```

Model Fields

If we want to change the code:

```
public class Person {  
    private /*@ spec_public non_null @*/ String name;
```

to become:

```
public class Person {  
    private /*@ non_null @*/ String fullName;
```

then we can add:

```
/*@ public model non_null String name;  
/*@ private represents name <- fullName;
```

Dutch National Flag Example

```
public class Flag {  
  
    public static final int BLUE = 1, WHITE = 2, RED = 3;  
  
    //@ public normal_behavior  
    //@     ensures \result <==>  
    //@         (i == BLUE || i == WHITE || i == RED);  
    public static /*@ pure @*/ boolean isColor(int i);  
  
    public int t[];  
    //@ public invariant  t != null &&  
    //@     (\forall int k;  
    //@         0 <= k && k < t.length; isColor(t[k]));  
}
```

```

/*@ public normal_behavior
   @   requires 0 <= i && i <= j && j <= t.length ;
   @   ensures \result <==>
           (\forall int k; i <= k && k < j; t[k] == c)
   @*/
private /*@ pure @*/ boolean
    isMonochrome(int i, int j, int c);

/*@ public normal_behavior
   @   requires    0 <= i && i < t.length &&
           0 <= j && j < t.length;
   @   modifiable t[i],t[j];
   @   ensures    t[i] == \old(t[j]) && t[j] == \old(t[i])
   @*/
private void swap(int i, int j);

```

```
/*@ public normal_behavior
   @   modifiable t[*];
   @   ensures
   @     (\exists int b,r; isMonochrome(0,b,BLUE) &&
   @     isMonochrome(b,r,WHITE) &&
   @     isMonochrome(r,t.length,RED));
   @*/
public void flag()
```

```

{ int b = 0, i = 0, r = t.length;
  /*@ loop_invariant
    @   (\forall int k;
    @     0 <= k && k < t.length; isColor(t[k])) &&
    @   0 <= b && b <= i && i <= r && r <= t.length &&
    @   isMonochrome(0,b,BLUE) &&
    @   isMonochrome(b,i,WHITE) &&
    @   isMonochrome(r,t.length,RED);
    @ decreases r - i;
    @*/
  while (i < r) { switch (t[i]) {
    case BLUE:  swap(b++, i++); break;
    case WHITE: i++; break;
    case RED:   swap(--r, i); break;
  }}}

```

Advantages of JML

- good documentation
- assigns responsibility
 - ▶ For example, if a precondition doesn't hold, then client is responsible for fixing method call.
- modularity of reasoning
 - ▶ In flag example, specifications of called methods are used to prove correctness of main method.
 - ▶ Reading specifications helps understand code (but only covers what is in specification).
- specify intent and allow freedom in implementation details

Tools

[Burdy et. al.] discusses 4 categories

1. Runtime Assertion Checking and Testing

- *jmlc*

- ▶ most widely used tool
- ▶ compiles and inserts code for checking preconditions, postconditions, and invariants
- ▶ goal is to debug both specifications and programs
- ▶ suggested use involves several steps
 - 1 specify preconditions for normal behavior
 - 2 define class invariants
 - 3 describe normal postconditions
 - 4 document exceptions and add exceptional postconditions

- *jmlunit*

- ▶ generates JUnit test cases that rely on the JML runtime assertion checker
- ▶ violating precondition not necessarily an error
- ▶ satisfying precondition, but violating postcondition indicates error

2. Static Checking and Verification

- *ESC/Java*
 - ▶ automatic checks of simple assertions and common errors such as dereferencing null, array index out of bounds, casting errors
 - ▶ uses automatic theorem prover called *Simplify*
- *ESC/Java2*: updated version that handles current version of JML and increases amount of JML features that can be checked
- *LOOP*:
 - ▶ formalizes semantics of Java and JML
 - ▶ can handle a much larger class of properties
 - ▶ if automatic proof is not possible, can use PVS theorem prover interactively

- *JACK*:
 - ▶ an environment for Java and Java Card program verification using JML annotations
 - ★ Note: Java Card is a dialect of Java designed for programming smart cards, adapted to limitations and does not support floating point numbers, strings, object cloning, or threads
 - ▶ Eclipse plug-in that allows hookup to various automatic and interactive provers
- *Krakatoa* (to be discussed further later):
 - ▶ checks full specifications for most of JML
 - ▶ formalizes semantics of Java and JML
 - ▶ interactive theorem prover Coq used to complete the proofs

Notes:

- Executable code is not affected by static checks. Dynamic checks require adding code that can affect efficiency.
- If full static verification is completed (e.g., via *Krakatoa*), then dynamic checking (via *jmlc*) is not needed.

3. Generating Specifications

- *Daikon*: invariant detector, finds operational abstractions, provides assistance in creating specifications
- *Houdini*: tries to supply missing annotations to eliminate ESC/Java warnings, new annotations verified by ESC/Java

4. Documentation

- *jmldoc*: builds human-readable HTML pages from JML annotations

Case Study: Electronic Purse on Smart Card

- Reference [16] of [Burdy et. al.]. Uses *ESC/Java*.
- Debit, credit, and currency change operations were annotated, along with the methods they use. Key generation and authorization not considered.
- Specification was “lightweight”. Functional specification describes behavior, but not of more complex code involving loops.
- 42 java classes involved; 432kB of Java (736kB with JML annotations); project length was 3 months.
- Some results:
 - ▶ Lightweight specification does not allow full verification, but nevertheless, some important errors in implementation were discovered.
 - ▶ Also, parts of code never reached were discovered (and could be removed).

Case Study Conclusions

- Lightweight formal verification (in this case using ESC/Java) can provide:
 - ▶ formal specifications of an application
 - ▶ checking of implementation by tools, increasing the confidence in the correctness of the implementation
- “The problems that we have found probably also would have been found by doing thorough testing, but using theorem proving techniques one is sure not to forget some cases, without having to put much effort in developing test scenarios.”
- “Also, writing the formal specifications forces one to think very precisely about the intended behaviour of programs, which helps in finding errors.”