Test Generation with JML

Part I – JMLUnitNG

Wojciech Mostowski, Gabriele Paganelli

http://wwwhome.ewi.utwente.nl/~mostowskiwi/
http://www.cse.chalmers.se/~gabpag/

University of Twente
Chalmers University of Technology

Overview

Part I

- JML as test oracle

Overview

Part I

- JML as test oracle
- Test data generation

Overview

Part I

- JML as test oracle
- Test data generation
- JMLTestNG

Overview

Part I

- JML as test oracle
- Test data generation
- JMLTestNG
- Good specifications for testing

Overview

Part I

- JML as test oracle
- Test data generation
- JMLTestNG
- Good specifications for testing

Part II

- Provide even better test data with KeY

The Basic Idea

Use JML specs to check I/O behaviour of methods

The Basic Idea

Use JML specs to check I/O behaviour of methods

- Take the input test data, evaluate the precondition
  - if true run the method with input data
  - if false skip – meaningless test

The Basic Idea

Use JML specs to check I/O behaviour of methods

- Take the input test data, evaluate the precondition
  - if true run the method with input data
  - if false skip – meaningless test
- After the execution of the method evaluate the postcondition
  - if true – test passed
  - if false – test fail, quote the values of the input data

Running Example

```
/*@ public normal_behavior
      requires param >= 0;
      ensures \result >= 10;

    also public normal_behavior
      requires param < 0;
      ensures \result < -10;
  @*/
public int makeHole(int param) {
  if(param >= 0) {
    return param + 10;
  }else{
    return param - 10;
  }
}
```

Running Example

```
/*@ public normal_behavior
      requires param >= 0;
      ensures \result >= 10;

    also public normal_behavior
      requires param < 0;
      ensures \result < -10;
  @*/
public int makeHole(int param) {
  if(param >= 0) {
    return param + 10;
  }else{
    return param - 10;
  }
}
```

- Take `param==0`, first precondition true, the first spec defines a meaningful test

Running Example

```
/*@ public normal_behavior
      requires param >= 0;
      ensures \result >= 10;

    also public normal_behavior
      requires param < 0;
      ensures \result < -10;
  @*/
public int makeHole(int param) {
  if(param >= 0) {
    return param + 10;
  }else{
    return param - 10;
  }
}
```

- Take `param==0`, first precondition true, the first spec defines a meaningful test
- The second does not, skip further checks

Running Example

```
/*@ public normal_behavior
      requires param >= 0;
      ensures \result >= 10;

    also public normal_behavior
      requires param < 0;
      ensures \result < -10;
  @*/
public int makeHole(int param) {
  if(param >= 0) {
    return param + 10;
  }else{
    return param - 10;
  }
}
```

- Take `param==0`, first precondition true, the first spec defines a meaningful test
- The second does not, skip further checks
- Execute method, check the postcondition(s) (true), test pass

Running Example

```
/*@ public normal_behavior
      requires param >= 0;
      ensures \result >= 10;

    also public normal_behavior
      requires param < 0;
      ensures \result < -10;
  @*/
public int makeHole(int param) {
  if(param >= 0) {
    return param + 10;
  }else{
    return param - 10;
  }
}
```

- Take `param==0`, first precondition true, the first spec defines a meaningful test
- The second does not, skip further checks
- Execute method, check the postcondition(s) (true), test pass
- Test with other inputs...

Running Example

```
/*@ public normal_behavior
      requires param >= 0;
      ensures \result >= 10;

    also public normal_behavior
      requires param < 0;
      ensures \result < -10;
  @*/
public int makeHole(int param) {
  if(param >= 0) {
    return param + 10;
  }else{
    return param - 10;
  }
}
```

- Take `param==0`, first precondition true, the first spec defines a meaningful test
- The second does not, skip further checks
- Execute method, check the postcondition(s) (true), test pass
- Test with other inputs...
- What inputs?!

Test Data Generation

- Primitive types:
  - Choose characteristic and borderline values
    e.g. `Integer.MIN_VALUE`, `-1`, `0`, `1`, `Integer.MAX_VALUE`

Test Data Generation

- Primitive types:
  - Choose characteristic and borderline values
    e.g. `Integer.MIN_VALUE`, `-1`, `0`, `1`, `Integer.MAX_VALUE`
  - Provide manual input: `-872463`, `123316`

Test Data Generation

- Primitive types:
  - Choose characteristic and borderline values
    e.g. `Integer.MIN_VALUE`, `-1`, `0`, `1`, `Integer.MAX_VALUE`
  - Provide manual input: `-872463`, `123316`
  - Still may not be enough – KeY can do better

Test Data Generation

- Primitive types:
  - Choose characteristic and borderline values
    e.g. `Integer.MIN_VALUE`, `-1`, `0`, `1`, `Integer.MAX_VALUE`
  - Provide manual input: `-872463`, `123316`
  - Still may not be enough – KeY can do better
- Reference types:
  - Objects created with default constructors

Test Data Generation

- Primitive types:
  - Choose characteristic and borderline values
    e.g. `Integer.MIN_VALUE`, `-1`, `0`, `1`, `Integer.MAX_VALUE`
  - Provide manual input: `-872463`, `123316`
  - Still may not be enough – KeY can do better
- Reference types:
  - Objects created with default constructors
  - Manual object factories. . .

Test Data Generation

- Primitive types:
  - Choose characteristic and borderline values
    e.g. `Integer.MIN_VALUE`, `-1`, `0`, `1`, `Integer.MAX_VALUE`
  - Provide manual input: `-872463`, `123316`
  - Still may not be enough – KeY can do better
- Reference types:
  - Objects created with default constructors
  - Manual object factories. . .
    We are getting close to regular, laborous test case writing

Test Data Generation

- Primitive types:
  - Choose characteristic and borderline values
    e.g. `Integer.MIN_VALUE`, `-1`, `0`, `1`, `Integer.MAX_VALUE`
  - Provide manual input: `-872463`, `123316`
  - Still may not be enough – KeY can do better
- Reference types:
  - Objects created with default constructors
  - Manual object factories. . .
    We are getting close to regular, laborous test case writing
  - Use reflection to create objects with arbitrary constructors

Test Data Generation

- Primitive types:
  - Choose characteristic and borderline values
    e.g. `Integer.MIN_VALUE`, `-1`, `0`, `1`, `Integer.MAX_VALUE`
  - Provide manual input: `-872463`, `123316`
  - Still may not be enough – KeY can do better
- Reference types:
  - Objects created with default constructors
  - Manual object factories. . .
    We are getting close to regular, laborous test case writing
  - Use reflection to create objects with arbitrary constructors
  - Create “empty” objects (Objenesis library)

Test Data Generation

- Primitive types:
  - Choose characteristic and borderline values
    e.g. `Integer.MIN_VALUE`, `-1`, `0`, `1`, `Integer.MAX_VALUE`
  - Provide manual input: `-872463`, `123316`
  - Still may not be enough – KeY can do better
- Reference types:
  - Objects created with default constructors
  - Manual object factories. . .
    We are getting close to regular, laborous test case writing
  - Use reflection to create objects with arbitrary constructors
  - Create “empty” objects (Objenesis library)
- Regardless of the method, created objects are only useful, when they satisfy some (JML) condtion

Test Data Generation

- Primitive types:
  - Choose characteristic and borderline values
    e.g. `Integer.MIN_VALUE`, `-1`, `0`, `1`, `Integer.MAX_VALUE`
  - Provide manual input: `-872463`, `123316`
  - Still may not be enough – KeY can do better
- Reference types:
  - Objects created with default constructors
  - Manual object factories. . .
    We are getting close to regular, laborous test case writing
  - Use reflection to create objects with arbitrary constructors
  - Create “empty” objects (Objenesis library)
- Regardless of the method, created objects are only useful, when they satisfy some (JML) condtion
- Collect objects after successful tests of constructors
  JMLUnitNG

JMLUnit New Generation

- Comprehensive JML based testing framework by Daniel Zimmerman and Joe Kiniry

JMLUnit New Generation

- Comprehensive JML based testing framework by Daniel Zimmerman and Joe Kiniry
- Core test generator:
  - Collect classes and methods with JML specifications

JMLUnit New Generation

- Comprehensive JML based testing framework by Daniel Zimmerman and Joe Kiniry
- Core test generator:
  - Collect classes and methods with JML specifications
  - Data generators with templates for manual input

JMLUnit New Generation

- Comprehensive JML based testing framework by Daniel Zimmerman and Joe Kiniry
- Core test generator:
  - Collect classes and methods with JML specifications
  - Data generators with templates for manual input
  - Create testing structure for everything

JMLUnit New Generation

- Comprehensive JML based testing framework by Daniel Zimmerman and Joe Kiniry
- Core test generator:
  - Collect classes and methods with JML specifications
  - Data generators with templates for manual input
  - Create testing structure for everything
- Runtime Assertion Checker (RAC) compiler:
  - Embed JML checks into the compiled Java code
  - Report results of evaluating JML expressions to the testing framework

JMLUnit New Generation

- Comprehensive JML based testing framework by Daniel Zimmerman and Joe Kiniry
- Core test generator:
  - Collect classes and methods with JML specifications
  - Data generators with templates for manual input
  - Create testing structure for everything
- Runtime Assertion Checker (RAC) compiler:
  - Embed JML checks into the compiled Java code
  - Report results of evaluating JML expressions to the testing framework
- Result: a standalone test suite based on the TestNG engine – `http://testng.org`
- Very efficient (compared to the older, non-NG version)

Some Technicalities – Process

1. Annotate your code with JML specs

Some Technicalities – Process

1. Annotate your code with JML specs
2. Run JMLUnitNG to generate tests

Some Technicalities – Process

1. Annotate your code with JML specs
2. Run JMLUnitNG to generate tests
3. (Fill in test data generators)

Some Technicalities – Process

1. Annotate your code with JML specs
2. Run JMLUnitNG to generate tests
3. (Fill in test data generators)
4. Compile the SUT with a JML-RAC enabled Java compiler

Some Technicalities – Process

1. Annotate your code with JML specs
2. Run JMLUnitNG to generate tests
3. (Fill in test data generators)
4. Compile the SUT with a JML-RAC enabled Java compiler
5. Compile the test suite with regular Java compiler

Some Technicalities – Process

1. Annotate your code with JML specs
2. Run JMLUnitNG to generate tests
3. (Fill in test data generators)
4. Compile the SUT with a JML-RAC enabled Java compiler
5. Compile the test suite with regular Java compiler
6. Run the test suite

Some Technicalities – Libraries

- `jmlunitng.jar` (steps 2, 5 & 6): test generator & runner and libraries it needs (OpenJML, antlr, TestNG, etc.)

Some Technicalities – Libraries

- `jmlunitng.jar` (steps 2, 5 & 6): test generator & runner and libraries it needs (OpenJML, antlr, TestNG, etc.)
- `jml4c.jar` (step 4), `jml4rt.jar` (step 6): one of the two possible JML RAC compilers

Some Technicalities – Libraries

- `jmlunitng.jar` (steps 2, 5 & 6): test generator & runner and libraries it needs (OpenJML, antlr, TestNG, etc.)
- `jml4c.jar` (step 4), `jml4rt.jar` (step 6): one of the two possible JML RAC compilers
  (The other has serious limitations)

Some Technicalities – Libraries

- `jmlunitng.jar` (steps 2, 5 & 6): test generator & runner and libraries it needs (OpenJML, antlr, TestNG, etc.)
- `jml4c.jar` (step 4), `jml4rt.jar` (step 6): one of the two possible JML RAC compilers
  (The other has serious limitations)
- `openjmlboot.jar` (step 2): might be needed in boot class path depending on the JVM used for test generation

Some Technicalities – Libraries

- `jmlunitng.jar` (steps 2, 5 & 6): test generator & runner and libraries it needs (OpenJML, antlr, TestNG, etc.)
- `jml4c.jar` (step 4), `jml4rt.jar` (step 6): one of the two possible JML RAC compilers
  (The other has serious limitations)
- `openjmlboot.jar` (step 2): might be needed in boot class path depending on the JVM used for test generation
- For Macs authors recommend OpenJDK ver. 7

Some Technicalities – Libraries

- `jmlunitng.jar` (steps 2, 5 & 6): test generator & runner and libraries it needs (OpenJML, antlr, TestNG, etc.)
- `jml4c.jar` (step 4), `jml4rt.jar` (step 6): one of the two possible JML RAC compilers
  (The other has serious limitations)
- `openjmlboot.jar` (step 2): might be needed in boot class path depending on the JVM used for test generation
- For Macs authors recommend OpenJDK ver. 7
- I used OpenJDK ver. 6 on Ubuntu

Example Again – Demo

```
/*@ public normal_behavior
      requires param >= 0;
      ensures \result >= 10;

    also public normal_behavior
      requires param < 0;
      ensures \result < -10;
  @*/
public int makeHole(int param) {
  if(param >= 0) {
    return param + 10;
  }else{
    return param - 10;
  }
}
```

Specifications for Testing

Observation I

Input data that does not satisfy the precondition is considered meaningless.

Specifications for Testing

Observation I

Input data that does not satisfy the precondition is considered meaningless. **The test is skipped!**

Specifications for Testing

Observation I

Input data that does not satisfy the precondition is considered meaningless. **The test is skipped!**

Consequence

Try to have the overall precondition *wide*, ideally the disjunction (*n* number of spec cases):

$$\texttt{pre}_1 \texttt{ || } \texttt{pre}_2 \texttt{ || } \ldots \texttt{ || } \texttt{pre}_n$$

should be equivalent to **`true`**.

Specifications for Testing

Observation II

The more non-overlapping specification cases the better – test data partitioning.

Specifications for Testing

Observation II

The more non-overlapping specification cases the better – test data partitioning.

Consequence

Make sure that there is input test data for every specification case!

Specifications for Testing

Observation II

The more non-overlapping specification cases the better – test data partitioning.

Consequence

Make sure that there is input test data for every specification case!

Observation IIa

The tested code may contain more information for data partitioning (e.g. **`if`** branches), KeY can help here, see Part II

Specifications for Testing

Observation III

Postconditions contain what is actually checked, the oracle.

Specifications for Testing

Observation III

Postconditions contain what is actually checked, the oracle.

Consequence

Given the information in the precondition and values that can be accessed, check what you can.

Specifications for Testing

Observation III

Postconditions contain what is actually checked, the oracle.

Consequence

Given the information in the precondition and values that can be accessed, check what you can.

Simple **`true`** is not a good postcondition, trivial test case. . .

Specifications for Testing

Observation III

Postconditions contain what is actually checked, the oracle.

Consequence

Given the information in the precondition and values that can be accessed, check what you can.

Simple **`true`** is not a good postcondition, trivial test case. . .

Well, it still specifies absence of exceptions.

Specifications for Testing

Observation III

Postconditions contain what is actually checked, the oracle.

Consequence

Given the information in the precondition and values that can be accessed, check what you can.
Simple **`true`** is not a good postcondition, trivial test case. . .
Well, it still specifies absence of exceptions.

Observation IIIa

Not all JML expressions are RAC checkable, e.g. some quantifications.
Such specifications are quitely accepted as true!

Specifications for Testing

Observation IV

Some JML tools do not deal well with multiple spec cases.

Specifications for Testing

Observation IV

Some JML tools do not deal well with multiple spec cases.

Solution

It is always possible to combine multiple spec cases into one:

```
public normal_behavior
  requires pre1;
  ensures post1;
also public behavior
  requires pre2;
  ensures post2;
  signals (Exception e)
    expost2;
```

Specifications for Testing

Observation IV

Some JML tools do not deal well with multiple spec cases.

Solution

It is always possible to combine multiple spec cases into one:

```
public normal_behavior
  requires pre1;
  ensures post1;
also public behavior
  requires pre2;
  ensures post2;
  signals (Exception e)
    expost2;
```

```
public behavior
  requires pre1 || pre2;
  ensures \old(pre1) ==> post1;
  ensures \old(pre2) ==> post2;
  signals (Exception e)
    (\old(pre2) ==> expost2)
    && !\old(pre1);
```

Specifications for Testing

Observation IV

Some JML tools do not deal well with multiple spec cases.

Solution

It is always possible to combine multiple spec cases into one:

```
public normal_behavior
  requires pre1;
  ensures post1;
also public behavior
  requires pre2;
  ensures post2;
  signals (Exception e)
    expost2;
```

```
public behavior
  requires pre1 || pre2;
  ensures \old(pre1) ==> post1;
  ensures \old(pre2) ==> post2;
  signals (Exception e)
    (\old(pre2) ==> expost2)
    && !\old(pre1);
```

Examples to Play With

- Binary search algorithm:
  - Specify two cases, element found and not found, hint:

```
requires array != null;
requires Array is sorted;
requires (\forall int i;
  i>=0 && i<array.length; array[i] != target);
ensures \result == -1;
```

  - Create and run the tests, find the bug in the code
  - File `BinSearch.java`, then

```
$ make TESTCLASS=BinSearch testgen
Fill in test data
$ make TESTCLASS=BinSearch run
```

Examples to Play With

- Binary search algorithm:
  - Specify two cases, element found and not found, hint:

```
requires array != null;
requires Array is sorted;
requires (\forall int i;
  i>=0 && i<array.length; array[i] != target);
ensures \result == -1;
```

  - Create and run the tests, find the bug in the code
  - File `BinSearch.java`, then

```
$ make TESTCLASS=BinSearch testgen
Fill in test data
$ make TESTCLASS=BinSearch run
```

- Integer division
  - A specification that cannot be RAC checked
  - File `Divide.java`

Examples to Play With

- Binary search algorithm:
  - Specify two cases, element found and not found, hint:

```
requires array != null;
requires Array is sorted;
requires (\forall int i;
  i>=0 && i<array.length; array[i] != target);
ensures \result == -1;
```

  - Create and run the tests, find the bug in the code
  - File `BinSearch.java`, then

```
$ make TESTCLASS=BinSearch testgen
Fill in test data
$ make TESTCLASS=BinSearch run
```

- Integer division
  - A specification that cannot be RAC checked
  - File `Divide.java`

Reference Material – My Makefile

```
OPENJMLBOOT=lib/openjmlboot.jar
JMLUNITNG=lib/jmlunitng.jar
JML4C=lib/jml4c.jar
JML4RT=lib/jml4rt.jar

TESTCLASS?=Example

all: testgen run

clean:
        rm -rf tests test-output
        find src/ -name "*.class" -exec rm \{\} \;

testgen: clean
        mkdir -p tests
        java -Xbootclasspath/p:$(OPENJMLBOOT) -jar $(JMLUNITNG) --dest tests --reflection src/

compile:
        java -jar $(JML4C) -cp $(JML4C):$(JMLUNITNG) src/
        javac -cp $(JML4RT):$(JMLUNITNG):src/ `find tests/ -name "*.java"`

run: compile
        java -cp $(JMLUNITNG):$(JML4RT):./tests/:./src/ org.charter.jmlunitng.$(TESTCLASS)_JML_Test

report: compile
        java -cp $(JMLUNITNG):$(JML4RT):./tests/:./src/ com.beust.testng.TestNG test_$(TESTCLASS).xml
```

References

Yoonsik Cheon and Gary Leavens. *A simple and practical approach to unit testing: The JML and JUnit way.* ECOOP'02 Proceedings, Springer LNCS 2374.

Daniel M. Zimmerman and Rinkesh Nagmoti. *JMLUnit: The Next Generation*, FoVeOOS 2010 Conference Proceedings, Springer LNCS 6528.

Software & Libraries

- OpenJML

  `http://jmlspecs.sourceforge.net/openjml.tar.gz`

- JML4C

  `http://www.cs.utep.edu/cheon/download/jml4c/download.php`

- JMLUnitNG

  `http://formalmethods.insttech.washington.edu/software/jmlunitng/`