

Measuring a Java Test Suite Coverage using JML Specifications

Frédéric Dadeau Yves Ledru Lydie du Bousquet

Laboratoire d'Informatique de Grenoble
LIG – UMR 5217

BP. 72, 38402 Saint-Martin d'Hères cedex, France

{Frederic.Dadeau,Yves.Ledru,Lydie.du-Bousquet}@imag.fr

Workshop Model-Based Testing 2007

Motivations

Testing

- aims at finding errors in a program
- not very expensive (compared to proof)
- nevertheless, not a complete method

How to know if a test suite is satisfying?

- Use mutation analysis
- Measure the test suite coverage

Motivations

Measuring the coverage of a test suite

- Can be performed using control-flow, data-flow coverage (white-box)
- or using **specification** coverage (black-box)

Recent arise of annotation languages

- Formally describe the behavior of programs within the source code
- Specification is close to the program
- Possibility of Runtime Assertion Checking

Our proposal

Use of JML ...

- Modeling language of Java
- Annotation language

... and measure its coverage by a test suite

- Written in Java
 - Using Runtime Assertion Checking mechanisms
-
- Evaluate the accuracy of a test suite
- ⇒ Implemented within a tool: `jmlCoverage`.

Outline

- 1 Java Modeling Language
- 2 Test Generation from JML Specifications
- 3 Coverage Metrics
- 4 Implementation and Experimentation
- 5 Conclusion & future work

Outline

- 1 Java Modeling Language
- 2 Test Generation from JML Specifications
- 3 Coverage Metrics
- 4 Implementation and Experimentation
- 5 Conclusion & future work

Presentation of the Java Modeling Language

Basics of JML

- Designed by Gary T. Leavens at IOWA State University
- Assertions embedded in the program comments `/*@ ... */`
- Based on the Java syntax/semantics

Design By Contract

- Introduced by Meyer with Eiffel
- Based on a contractual agreement between the system and the methods
- The system has to fulfill the method's precondition when calling it ...
- ... the method is bound to establish its postcondition

Presentation of the Java Modeling Language (cont'd)

Modelling possibilities of JML

- Class specifications: invariants, history constraints
- Method specifications: precondition, frame condition, normal postcondition, exceptional postcondition

Specification of an electronic purse

```

public class Demoney {
    static final SET_MAX_DEBIT = 1;
    static final SET_MAX_BAL = 2;

    /* invariant balance >= 0 &&
       balance <= maxBal;
    int balance, maxBal, maxDebit;

    boolean personalized;

    /* behavior
       requires personalized == false;
       {
           requires p1 == SET_MAX_DEBIT;
           assignable maxDebit, maxBal;
           ensures maxDebit == data &&
              maxBal == \old(maxBal);
       }
       also
           requires p1 == SET_MAX_BAL;
           assignable maxDebit, maxBal;
           ensures maxBal == data &&
              maxDebit == \old(maxDebit);
       }
    }

    public void PUT_DATA(byte p1, short data);
    ...
}

```

A closer look on the example

The first behavior ...

```
/*@ behavior
  @   requires personalized == false;
  @   {}
  @   requires p1 == SET_MAX_DEBIT;
  @   assignable maxDebit, maxBal;
  @   ensures maxDebit == data &&
  @           maxBal == \old(maxBal);
...

```

A test case covering the first behavior

```
Demoney d = new Demoney();
d.PUT_DATA(Demoney.SET_MAX_DEBIT, (short) 10000);

```

A closer look on the example

... the second behavior ...

```
/*@ behavior
@   requires personalized == false;
@   {}
@   ...
@   requires p1 == SET_MAX_BAL;
@   assignable maxDebit, maxBal;
@   ensures maxBal == data &&
@           maxDebit == \old(maxDebit);
...

```

A test case covering the second behavior

```
Demoney d = new Demoney();
d.PUT_DATA(Demoney.SET_MAX_BAL, (short) 32767);

```

A closer look on the example

... and the third behavior

```
...
@   requires personalized == true ||
@       (p1 != SET_MAX_DEBIT &&
@         p1 != SET_MAX_BAL)
@   assignable maxDebit, maxBal;
@   ensures false;
@   signals (CardException e)
@       maxDebit == \old(maxDebit)
@       && maxBal == \old(maxBal);
@*/
```

A test case covering the third behavior

```
Demoney d = new Demoney();
d.PUT_DATA(Demoney.SET_MAX_DEBIT, (short) 10000);
d.PUT_DATA(Demoney.SET_MAX_BAL, (short) 32767);
d.STORE_DATA(); // ends personalization phase
d.PUT_DATA(Demoney.SET_MAX_BAL, (short) 20000);
```

Presentation of the Java Modeling Language (cont'd)

Use of JML

- Help the proof of the program (ESC/Java2, JACK)
- Test oracle using the Runtime Assertion Checking

But, also used for Model-Based Testing!

- JML specification considered only (no Java code required!)
- compute test targets + compute test cases
- published at Formal Methods 2006^a
 - Implemented within a tool: JML-Testing-Tools Test Generator (<http://lifc.univ-fcomte.fr/~jmltt>)

^aF. Bouquet, F. Dadeau, B. Legnard. *Automated Boundary Test Generation from JML Specifications*. Proceedings of the 14th Symposium on Formal Methods (FM'06)

Outline

- 1 Java Modeling Language
- 2 Test Generation from JML Specifications**
- 3 Coverage Metrics
- 4 Implementation and Experimentation
- 5 Conclusion & future work

How the story begins ...

First, animation of a JML specification

- Simulate the functional behavior of the modelled Java program
 - Using the JML specification only!
 - Methods' executions are abstracted by considering their JML behaviors
- ⇒ Provides a way to validate the behavior of the JML model

How the story begins ...

First, animation of a JML specification

- Simulate the functional behavior of the modelled Java program
 - Using the JML specification only!
 - Methods' executions are abstracted by considering their JML behaviors
- ⇒ Provides a way to validate the behavior of the JML model

Second, test generation using the JML specifications

Testing methods ⇔ activating behaviors expressed in the JML method specifications

1 Extraction of test targets

activation of a method's behavior

+ some additional constraint on the data values

2 Building of the test cases using the JML model animation

3 Reification & execution on an IUT (with RAC)

How the story begins ...

Extraction of the test targets

- Test target \Leftrightarrow activation condition of the behaviors of Java method
- Behavior given by the JML method specification
- Test targets expanded, by rewriting disjunctions

How the story begins ...

Extraction of the test targets

- Test target \Leftrightarrow activation condition of the behaviors of Java method
- Behavior given by the JML method specification
- Test targets expanded, by rewriting disjunctions

What if ... we reverse our way of thinking?

- (Accurate) test targets are obtained by specification coverage

How the story begins ...

Extraction of the test targets

- Test target \Leftrightarrow activation condition of the behaviors of Java method
- Behavior given by the JML method specification
- Test targets expanded, by rewriting disjunctions

What if ... we reverse our way of thinking?

- (Accurate) test targets are obtained by specification coverage
- ⇒ Let's check whether other approaches are able to perform such a coverage!

Outline

- 1 Java Modeling Language
- 2 Test Generation from JML Specifications
- 3 Coverage Metrics**
- 4 Implementation and Experimentation
- 5 Conclusion & future work

2 parts for coverage metrics

① Behavioral Coverage

② Condition Coverage

2 parts for coverage metrics

1 Behavioral Coverage

- Extract the behaviors from a JML method specification
- ⇒ JML method spec. represented as a Directed Acyclic Graph
- ⇒ Edges are labelled by before/after predicates
- Measure the coverage of the graph (nodes, edges, paths, etc.)

2 Condition Coverage

2 parts for coverage metrics

1 Behavioral Coverage

- Extract the behaviors from a JML method specification
- ⇒ JML method spec. represented as a Directed Acyclic Graph
- ⇒ Edges are labelled by before/after predicates
- Measure the coverage of the graph (nodes, edges, paths, etc.)

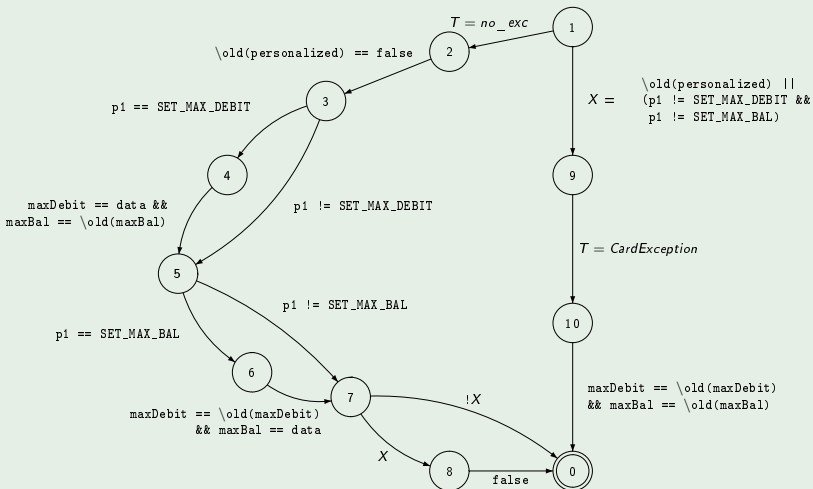
2 Condition Coverage

- Rewrite disjunctions in the conditions
- Measure how the disjunction is covered

Outline

- 1 Java Modeling Language
- 2 Test Generation from JML Specifications
- 3 Coverage Metrics**
 - Behavioral Coverage
 - Condition Coverage
- 4 Implementation and Experimentation
- 5 Conclusion & future work

Graph of the PUT_DATA method specification



According to JML Semantics

```

/*@ requires P1;
   @ assignable A;
   @ ensures Q1;
   @ also
   @ requires P2;
   @ assignable A;
   @ ensures Q2;
   @*/

```

 \implies

```

/*@ requires P1 || P2;
   @ assignable A;
   @ ensures \old(P1) ==> Q1;
   @ ensures \old(P2) ==> Q2;
   @*/

```

According to JML Semantics

```

/*@ requires P1;
   @ assignable A;
   @ ensures Q1;
   @ also
   @ requires P2;
   @ assignable A;
   @ ensures Q2;
   @*/

```

 \implies

```

/*@ requires P1 || P2;
   @ assignable A;
   @ ensures \old(P1) ==> Q1;
   @ ensures \old(P2) ==> Q2;
   @*/

```

```

/*@ requires P1 || P2;
   @ assignable A;
   @ ensures \old(P1) ==> Q1;
   @ ensures \old(P2) ==> Q2;
   @*/

```

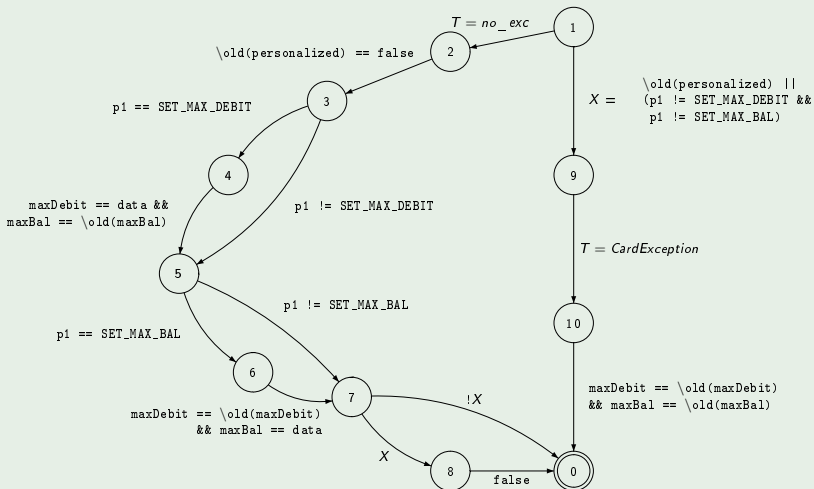
 \implies

```

/*@ requires P1 || P2;
   @ assignable A;
   @ ensures (\old(P1) && Q1)
   @         || \old(!P1);
   @ ensures (\old(P2) && Q2)
   @         || \old(!P2);
   @*/

```

Graph of the PUT_DATA method specification

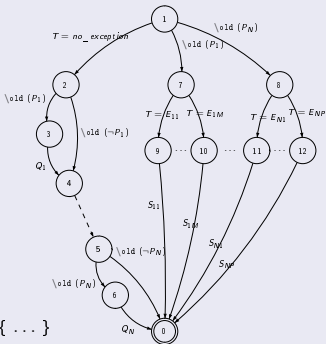


Extraction of a graph from a method specification (generalized)

```

/*@ behavior
  @ requires P1;
  @ assignable A;
  @ ensures Q1;
  @ signals (E11 e11) S11;
  @ ...
  @ signals (E1M e1M) S1M;
  @ also
  @ ...
  @ also
  @ requires PN;
  @ assignable A;
  @ ensures QN;
  @ signals (EN1 eN1) SN1;
  @ ...
  @ signals (ENP eNP) SNP;
  @*/
Type meth(T1 p1, ...) throws E11, ..., ENP { ... }

```

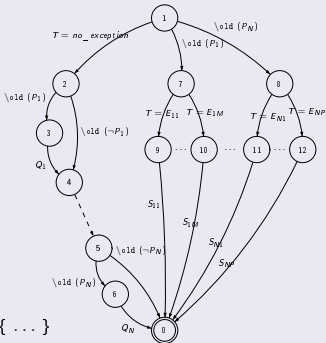


Extraction of a graph from a method specification (generalized)

```

/*@ behavior
  @ requires P1;
  @ assignable A;
  @ ensures Q1;
  @ signals (E11 e11) S11;
  @ ...
  @ signals (E1M e1M) S1M;
  @ also
  @ ...
  @ also
  @ requires PN;
  @ assignable A;
  @ ensures QN;
  @ signals (EN1 eN1) SN1;
  @ ...
  @ signals (ENP eNP) SNP;
  @*/
Type meth(T1 p1, ...) throws E11, ..., ENP { ... }

```



Graph coverage criteria

all nodes \subseteq all edges \subseteq all paths

Paths for the PUT_DATA graph

Among the 9 possible paths, only 3 are consistent

[1 → 2 → 3 → 4 → 5 → 7 → 0]

[1 → 2 → 3 → 5 → 6 → 7 → 0]

[1 → 9 → 10 → 0]

Consistent path computation/detection

- Detected using the JML-Testing-Tools constraint solving engine
- Checked by theorem proving (Simplify, haRVey, ...)

Outline

- 1 Java Modeling Language
- 2 Test Generation from JML Specifications
- 3 Coverage Metrics**
 - Behavioral Coverage
 - Condition Coverage
- 4 Implementation and Experimentation
- 5 Conclusion & future work

Why an additional criterion?

- More thinner coverage of the specification predicates
 - Most people write JML method specification in only one block
- ⇒ One single path in the graph!

Why an additional criterion?

- More thinner coverage of the specification predicates
 - Most people write JML method specification in only one block
- ⇒ One single path in the graph!

Example with multiple specification block ...

```

/*@ behavior
  @ requires personalized == false;
  @ {
  @   requires p1 == SET_MAX_DEBIT;
  @   assignable maxDebit, maxBal;
  @   ensures maxDebit == data && maxBal == \old(maxBal);
  @   also
  @   requires p1 == SET_MAX_BAL;
  @   assignable maxDebit, maxBal;
  @   ensures maxBal == data && maxDebit == \old(maxDebit);
  @ }
  @ ...
  @*/

```

Why an additional criterion?

- More thinner coverage of the specification predicates
 - Most people write JML method specification in only one block
- ⇒ One single path in the graph!

Example with a single specification block

```

/*@ behavior
  @ requires personalized == false;
  @ requires p1 == SET_MAX_DEBIT ||
  @       p1 == SET_MAX_BAL;
  @ assignable maxDebit, maxBal;
  @ ensures \old(p1)== SET_MAX_DEBIT ==>
  @         maxDebit == data && maxBal == \old(maxBal);
  @ ensures \old(p1)== SET_MAX_BAL ==>
  @         maxBal == data && maxDebit == \old(maxDebit);
  @ ...
  @*/

```

Example with a single specification block

```
/*@ behavior
@   requires personalized == false;
@   requires p1 == SET_MAX_DEBIT ||
@       p1 == SET_MAX_BAL;
@   assignable maxDebit, maxBal;
@   ensures  \old(p1) == SET_MAX_DEBIT ==>
@       maxDebit == data && maxBal == \old(maxBal);
@   ensures  \old(p1) == SET_MAX_BAL ==>
@       maxBal == data && maxDebit == \old(maxDebit);
@   ...
@*/
```

Example with a single specification block

```

/*@ behavior
  @   requires personalized == false;
  @   requires p1 == SET_MAX_DEBIT ||
  @       p1 == SET_MAX_BAL;
  @   assignable maxDebit, maxBal;
  @   ensures  \old(p1) == SET_MAX_DEBIT ==>
  @           maxDebit == data && maxBal == \old(maxBal);
  @   ensures  \old(p1) == SET_MAX_BAL ==>
  @           maxBal == data && maxDebit == \old(maxDebit);
  @   ...
  @*/

```

Behavior coverage no more sufficient!

- One single path for the normal termination behavior
- Indifferently covered by

```

Demoney d = new Demoney();
d.PUT_DATA(Demoney.SET_MAX_DEBIT, (short) 10000);

```

```

or Demoney d = new Demoney();
d.PUT_DATA(Demoney.SET_MAX_BAL, (short) 32767);

```

Rewritings

Coverage achieved by rewriting disjunctions ...

Rewriting	Set of predicates to evaluate for $P_1 \ \ P_2$	Coverage Criteria
RW1	$\{P_1 \ \ P_2\}$	
RW2	$\{P_1, P_2\}$	CC
RW3	$\{P_1 \ \&\& \ !P_2, !P_1 \ \&\& \ P_2\}$	FPC
RW4	$\{P_1 \ \&\& \ !P_2, !P_1 \ \&\& \ P_2, P_1 \ \&\& \ P_2\}$	MCC

... and checking that all literals are covered at least once by the test suite.

Rewritings

Coverage achieved by rewriting disjunctions ...

Rewriting	Set of predicates to evaluate for $P_1 \ \ P_2$	Coverage Criteria
RW1	$\{P_1 \ \ P_2\}$	
RW2	$\{P_1, P_2\}$	CC
RW3	$\{P_1 \ \&\& \ !P_2, \ !P_1 \ \&\& \ P_2\}$	FPC
RW4	$\{P_1 \ \&\& \ !P_2, \ !P_1 \ \&\& \ P_2, \ P_1 \ \&\& \ P_2\}$	MCC

... and checking that all literals are covered at least once by the test suite.

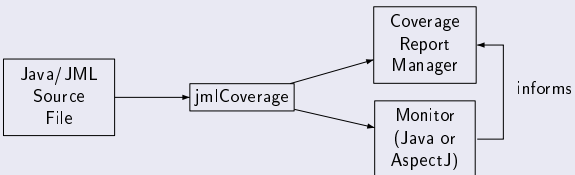
Condition Coverage Criteria hierarchy

$$RW1 \subseteq RW2 \subseteq RW3 \subseteq RW4$$

Outline

- 1 Java Modeling Language
- 2 Test Generation from JML Specifications
- 3 Coverage Metrics
- 4 Implementation and Experimentation**
- 5 Conclusion & future work

Ideas implemented within a prototype tool: jmlCoverage



Monitoring using embedded Java

Runtime checker style

```

/*@ behavior
  @   requires  $P_1$ ;
  @   assignable  $A$ ;
  @   ensures  $Q_1$ ;
  @   signals ( $E_{11}$  e11)  $S_{11}$ ;
  @   ...
  @   signals ( $E_{1M}$  e1M)  $S_{1M}$ ;
  @ also
  @   ...
  @ also
  @   requires  $P_N$ ;
  @   assignable  $A$ ;
  @   ensures  $Q_N$ ;
  @   signals ( $E_{N1}$  eN1)  $S_{N1}$ ;
  @   ...
  @   signals ( $E_{NP}$  eNP)  $S_{NP}$ ;
/*@
Type meth( $T_1$   $p_1, \dots$ ) throws  $E_{11}$  ... {
  body;
}

```

```

Type meth( $T_1$   $p_1, \dots$ ) throws  $E_{11}, \dots$  {
  try {
    Check and report precondition
edges predicates coverage
    body;
  }
  catch (java.lang.Error e) {
    if (e instanceof  $E_{11}$ ) {
      Check and report edges
predicates coverage for  $E_1$ 
    }
    ...
    if (e instanceof  $E_{NP}$ ) {
      Check and report edges
predicates coverage for  $E_N$ 
    }
    throw e;
  }
  Check and report edges predicates
coverage for normal postcondition
}

```

Experimentation

Configuration

- Target program: Demoney applet (electronic purse)
 - 4 classes
 - Annotated with 500 lines of JML
- Test suite generators:
 - Jartege – Java Random Test Generator
 - Tobias – Combinatorial test generation

Results

- Jartege experiment: feedback on the practicability of the approach
- Tobias experiment: 5 schemas unfolded into 162 test cases covering 100% of consistent paths
- Low cost in execution time

Results

- JarTEge experiment: feedback on the practicability of the approach
- Tobias experiment: 5 schemas unfolded into 162 test cases covering 100% of consistent paths
- Low cost in execution time

Some interesting ideas with jmlCoverage

- Improving JarTEge – halting criterion for random testing
- Improving Tobias – filtering/reducing test suites w.r.t. the JML coverage criteria

Conclusion & future work

We have presented

- A **black-box** coverage measure for Java programs, based on
 - the **behavioral coverage** of the JML method specifications
 - the **condition coverage** of the JML predicates
- A prototype tool implementing the idea
- An approach that can be used to complete other coverage measures

Conclusion & future work

We have presented

- A **black-box** coverage measure for Java programs, based on
 - the **behavioral coverage** of the JML method specifications
 - the **condition coverage** of the JML predicates
- A prototype tool implementing the idea
- An approach that can be used to complete other coverage measures

For the future ...

- Extension to measure the coverage of other JML clauses (invariant, history constraints, etc.)
- Improve the prototype to be a feature of the JML-RAC
- Use this coverage criteria for test suite reduction

Thanks for your attention

Questions ?