Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 190 (2007) 21–32

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Measuring a Java Test Suite Coverage Using JML Specifications

F. Dadeau[a,1] [,4] Y. Ledru[a,2] L. du Bousquet[a,3]

[a] *Laboratoire Informatique de Grenoble*
BP 72, 38402 Saint Martin d'Hères, France

Abstract

We propose in this paper a way to measure the coverage of a Java test suite by considering the JML specification associed to the Java program under test. This approach is based on extracting a predicate-based graph from the JML method specifications. We then measure the coverage of this latter w.r.t. nodes of the graph that are visited by the test suite. In addition, we propose to check whether the test suite satisfies classical condition coverage criteria. We also introduce a tool, to be used as precompiler for Java, that is in charge of measuring and reporting the coverage according to these criteria.

Keywords: Specification coverage, test suite, Java, JML, condition coverage.

1 Introduction

The essence of testing consists in executing the system under test in order to find bugs [21]. Nevertheless, testing can not be a complete approach since exhaustive testing is not applicable; the validation engineer is often left with a test suite that did not detect any bug in the program. How can he/she be sure that the test suite that was run is relevant enough to be confident in the program? One solution is to evaluate the quality of the test suite.

Several works on test suite evaluation exist, such as exercising the test suite on mutations of a program. The most relevant technique is to measure the coverage of the test suite. Usually, the coverage is measured on the control-flow graph of the program [21], or on the data-flow of the program [24]. In addition, a specification coverage can possibly be measured.

[1] Email: Frederic.Dadeau@imag.fr

[2] Email: Yves.Ledru@imag.fr

[3] Email: Lydie.du-Bousquet@imag.fr

[4] This work has been partially funded by the RNTL POSE project (ANR-05-RNTL-01001), supported by the French Ministry of Research and New Technologies.

doi:10.1016/j.entcs.2007.08.003

The recent arise of annotation languages makes it possible to specify the behavior of programs (i.e. of methods) inside the source code in terms of pre- and postconditions. It provides another "vision" of what a method should do, which can also be seen as expressing low-level *requirements*. It also provides a black-box view of what a method should do, expressed in terms of a *contract* [20].

Model-based testing [2] consists in computing test suites from a model of the considered program or system. Model-based conformance testing consists in ensuring that the program does not have an unintended behavior w.r.t. its specification. This conformance can be observed through *observation* points or using a run-time assertion checking mechanism if the proximity of both the specification and the program makes it possible. In this context, the Java Modeling Language [16] (JML) has been introduced to act as a behavioral interface specification language (BISL) for Java programs. JML can be used as an oracle for testing, considering that if no JML assertion is ever violated during the program execution, then the test succeeds, otherwise, it fails.

A previous work [3] has introduced the principles of model-based testing from JML specifications. Thus, some JML-based coverage criteria were used to guide the test target definition. We came to the idea that the coverage criteria defined in the latter work could be used to evaluate test suites that would have been produced by several tools, using different approaches, such as combinatorial testing (e.g. JMLUnit [6], Tobias [18]) or random testing (e.g. Jartege [23]).

We propose an approach for evaluating test suites for Java programs w.r.t. the coverage of an associated JML specification, expressing the behavior and/or the requirements of the methods. In addition, we propose to check different condition coverage criteria, that are contained within the disjunctions of the predicates of the specification.

The paper is organized as follows. Section 2 introduces the modeling possibilities provided by the Java Modeling Language. The first coverage criterion, based on the method specifications, is presented in Sect. 3. Section 4 is dedicated to the condition coverage definition. The principles of the measure and especially the implementation and the experiments are detailed in Sect. 5. Section 6 presents the related work, before concluding and providing a glimpse of the future work in Sect. 7.

2 Java Modeling Language

The Java Modeling Language –JML– has been designed by Gary T. Leavens et al. at Iowa State University [16,17]. The modeling elements are displayed as annotations, embedded within the Java source code. JML is based on the design by contract principle (DBC) [20] which states that, at the invocation of a method the system has to fulfill a contract (by satisfying the method's precondition) and as a counterpart, the method is expected to fulfill its contract (by establishing its postcondition).

JML makes it possible to express the static part of the system, such as invariants, and the dynamic part of the system, using postconditions and history con-

```
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;
    @     requires data >= 0;
    @     assignable maxDebit, maxBal;
    @     ensures maxDebit == data &&
    @             maxBal == \old(maxBal);
    @   also
    @     requires p1 == SET_MAX_BAL;
    @     requires data >= balance;
    @     assignable maxDebit, maxBal;
    @     ensures maxBal == data &&
    @         maxDebit == \old(maxDebit);
    @   |}
    @ also
    @   requires personalized == true ||
    @            ((p1 != SET_MAX_DEBIT ||
    @              data < balance) &&
    @             (p1 != SET_MAX_BAL ||
    @              data < 0));
    @   assignable maxDebit, maxBal;
    @   ensures false;
    @   signals (CardException e)
    @         maxDebit == \old(maxDebit)
    @         && maxBal == \old(maxBal);
    @*/
  public void PUT_DATA(byte p1, short data);
  ...
}
```

Figure 1. An example of a JML specification

straints. The model is expressed through several *clauses*, identified by a keyword, and followed by a predicate. The `invariant` and `constraint` clauses respectively designate the invariant and the history constraints that apply to a class. The general behaviors of the methods are specified through method specifications, which contain specification blocks, separated by the `also` keyword. Each specification block displays preconditions (`requires` clause), normal postconditions –established when the method terminates without throwing an exception– (`ensures` clause), or exceptional postconditions –established when the method terminates by throwing an exception– (`signals` clause). The `assignable` clause gives the frame of the method. The JML predicate syntax is similar to the Java predicate syntax enriched with special keywords, prefixed by \, notably introducing quantifiers (`\forall`, `\exists`).

An example of a JML specification is displayed in Fig. 1. This specification presents a simplified version of the Demoney electronic purse [19]. Attributes `balance`, `maxBal`, and `maxDebit` respectively represent the amount of money on the purse, the maximal amount of money, and the maximal debit authorized. Finally, attribute `personalized` states whether or not the purse has been configured, by defining the values of the latter two attributes. The method displayed here makes it possible to configure the purse, by setting either the maximal value of balance, or the maximal debit. The method may throw an exception either if the parameter `p1` is wrong of if the card is already personalized.

The JML Runtime Assertion Checker (RAC) [5] has been developed to check the JML specification clauses when running the program. This tool, provided in the JML releases, acts as a precompiler which modifies the source of the program to add the following verifications on the JML model: (*i*) checking of the preconditions and the invariant when a method is entered, (*ii*) catching exceptions that may be thrown and checking of the exceptional postcondition related to the considered exception before throwing the exception again, (*iii*) checking of the normal postconditions if the method terminates normally. Notice that the invariant and history constraints are also checked during steps (*ii*) and (*iii*).

The next section introduces our proposal to decompose the JML method specifications into *behaviors*, whose coverage are the measuring unit of our approach. In addition, we will consider condition coverage criteria that will add more granularity to the measure.

3 Coverage of the Method Specifications

The method specifications describe *behaviors* of the Java methods. A behavior is either *normal* if the method terminates normally (without throwing an exception) or *exceptional* if the method terminates by throwing an exception. Our technique is to extract a predicate-based graph from the method specifications, that gives a representation of the behaviors of the method. Traversing the graph is equivalent to create a conjunction of the label predicates on its edges.

We represent each JML method specification by a graph, as shown in Fig. 2. In this figure, $P_k (k \in 1..N)$ are the precondition predicates, A gives the frame condition, $Q_k (k \in 1..N)$ are the normal postconditions, $S_p (p \in 1..M)$ are the exceptional postconditions related to the exceptions E_p. The terminations are distinguished by T, which might be either *no_exception* indicating a normal behavior, or any of the declared exceptions E_p. Invariants and history constraints are (currently) not considered.

```
/*@ 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) $S_{NP}$;
  @*/
$T$ $meth(T_1\ p_1, \ldots)$
          throws $E_{11}, \ldots, E_{NP}$ { ... }
```

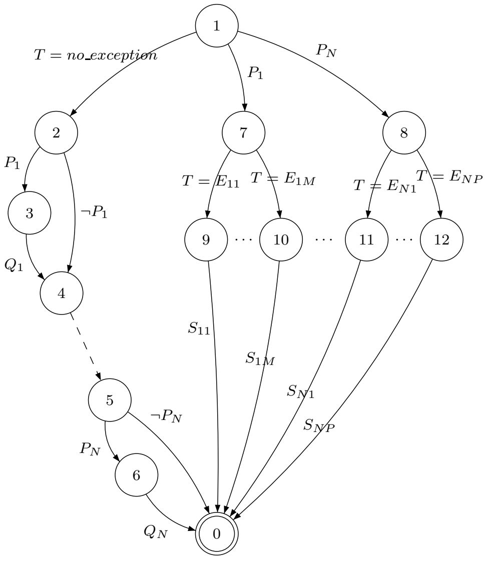

Figure 2. Extraction of the behaviors from a JML method specification

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

Figure 3. Desugaring of JML method specification blocks

The extraction of the graph is done as follows. A first branch containing the normal behavior is built. According to the semantics of JML, when the precondition of a method specification block is satisfied then the normal postcondition has to be established. Otherwise, if the precondition is not satisfied, anything may happen. This desugaring [25] can be expressed by Fig. 3. As a consequence, conditional branchings are created, in order to re-create the implication. One branching is done for each method specification block. In case of exceptional termination, if an exception is thrown, then, depending on the precondition, an exceptional postcondition has to be established. An example of such a graph is given in Fig. 4. Notice that frame conditions are not considered in the graph.

The coverage of the method specifications is achieved by covering the method specification graph. Since this graph is directed and acyclic, we do not have to cover loops. Thus, the following options can be proposed.

- *all nodes.* Achieved when a test suite activates all the nodes of the graph.
- *all edges.* Achieved when a test suite activates all the edges of the graph.
- *all paths.* Achieved when a test suite activates all the paths of the graph going from node 1 to node 0.

The hierarchy between these options is the following:

$$\textit{all nodes} \subseteq \textit{all edges} \subseteq \textit{all paths}$$

We assume that only consistent paths are computed/measured. This is ensured either by writing a comprehensive JML method specification, or by using a dedicated tool, such as JML-Testing-Tools constraint solving engine [4] or a theorem prover, such as Simplify [9] or haRVey [8] to prune the inconsistent paths in the method specification graph.

Example 3.1 [Extraction of a Graph from a Method Specification] Consider the method specification example given in Fig. 1. The corresponding graph is given in Fig. 4. A path is read as a conjunction of the statements appearing on its edges. On this example, path *[*$1 \rightarrow 9 \rightarrow 10 \rightarrow 0$*]* is equivalent to $X \wedge T = CardException \wedge maxDebit = \backslash old(maxDebit) \wedge maxBal = \backslash old(maxBal)$. This graph presents several inconsistent paths, among all the possible paths leading from node 1 to node 0. On the example, only paths *[*$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 0$*]*, *[*$1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 0$*]*, and *[*$1 \rightarrow 9 \rightarrow 10 \rightarrow 0$*]* are consistent.

Figure 3 illustrates the desugaring of JML method specifications, rewriting sev-

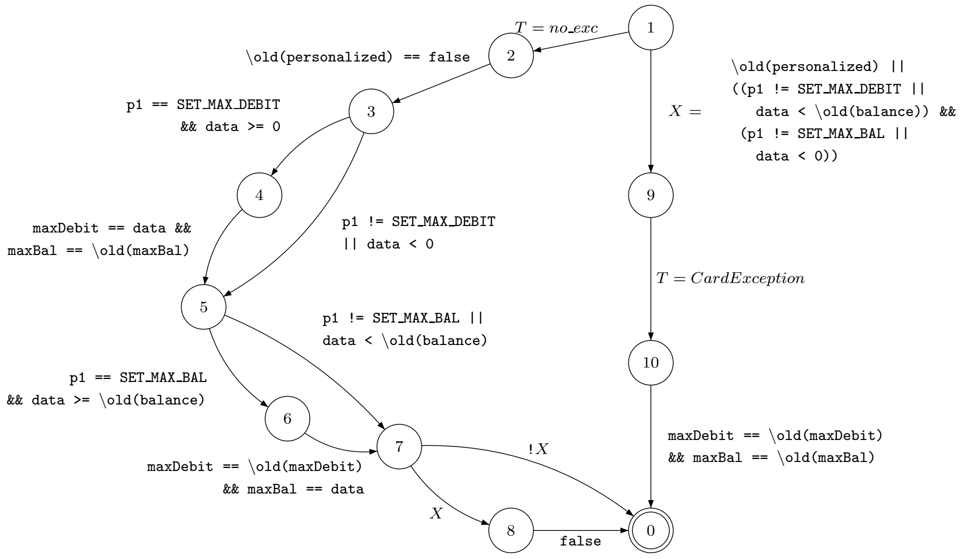
$T = no\_exc$
1
2
\old(personalized) == false
3
p1 == SET_MAX_DEBIT && data >= 0
4
maxDebit == data && maxBal == \old(maxBal)
p1 != SET_MAX_DEBIT || data < 0
5
p1 != SET_MAX_BAL || data < \old(balance)
p1 == SET_MAX_BAL && data >= \old(balance)
6
7
maxDebit == \old(maxDebit) && maxBal == data
$!X$
X
8
false
0
$X =$ \old(personalized) || ((p1 != SET_MAX_DEBIT || data < \old(balance)) && (p1 != SET_MAX_BAL || data < 0))
9
$T = CardException$
10
maxDebit == \old(maxDebit) && maxBal == \old(maxBal)

Figure 4. Graph extracted from the PUT_DATA method specification

eral blocks into a single one. The graph construction is based on the hypothesis that specifications are divided, as much as possible, into several blocks (left part of the figure). Nevertheless, most of the JML specification writers are not used to split the specification like that and only write one huge postcondition into which case-based postconditions are guarded (middle part of the figure). Thus, a graph for such a method specification would only consider one precondition and one postcondition. Measuring the coverage of this kind of graph is not really relevant. That is why, in addition to the coverage of the predicate graph, we also consider condition coverage criteria, for the predicates of the graph. If the method specification is divided into blocks, then this additional coverage increases the granularity of our measure.

4 Condition Coverage

Condition coverage is achieved by rewriting the disjunctions embedded within the predicates on the edges composing a path. We distinguish 4 rewritings, each one of them representing a specific condition coverage criterion. These rewritings and their associated coverage criteria are given by Table 1.

Rewriting 1 consists in checking the disjunction without any changes. This is the most basic way to verify a disjunction, by choosing the first positive literal.

Rewriting 2 consists in considering each literal independently. This rewriting satisfies the Condition Coverage criterion (CC).

Rewriting 3 considers each literal in an exclusive manner, by evaluating each literal and the negation of the others. Thus, this rewriting satisfies the Full Predicate Coverage criterion (FPC) [22].

Finally, the last rewriting evaluates each possibility to satisfy the disjunction. This allows to satisfy the Multiple Condition Coverage (MCC).

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

Table 1
Disjunctions Rewritings and Coverage Criteria

Here again, we can establish the following hierarchy between the rewriting and the condition coverage criteria.

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

Notice that measuring the coverage of a precondition can be reduced to measuring the satisfaction of the precondition, whereas usually the unsatisfaction of the precondition is also measured. Since we independantly consider the negation of the precondition, by construction of the graph, this step is implicitely performed.

For practical reasons, all these rewritings are only applied on the positive preconditions of the method specification blocks. Indeed, the application of these rewritings on negations of the preconditions would lead to a combinatorial explosion of the number of cases. Nevertheless, it is possible to apply RW1 or RW2, which may be an indicator of whether the test suite tries to perform unauthorized actions, and the contexts these actions are tried to be activated.

5 Performing measurements

First, we introduce the principles used to perform the coverage measurement. Second, we present a tool implementing these principles.

5.1 Principles

The principle of checking the coverage of a JML specification is similar to the runtime checking of the assertions as performed in the RAC. It is presented as a preprocessing which enriches the original Java code with the verification of the JML predicates. In addition, we need to setup a *Coverage Report Manager* (CRM) dedicated to the measures must be performed.

The CRM keeps track of the graphs representing each method specification. Each time a predicate is checked within the source code, the report manager is informed of the edge that has been covered and the node that has been reached. In brief, the principle is illustrated on a generic method specification in Fig. 5.

It is important to notice that, as for the JML Runtime Assertion Checker, the verifications added to the Java code do not change the functional behavior of the methods and the functional behavior of the program in general.

Dedicated internal methods, within the CRM are in charge of computing the edges/nodes coverage achieved at the end of the test suite execution. It is possible to display a report or to consult them using a customized API.

5.2 The jmlCoverage Tool

The jmlCoverage tool implements the principles described before, as illustrated in Fig. 6. It acts as a precompiler that produces the Coverage Report Manager (as a Java source file) and the monitor itselft, as an AspectJ file or an instrumented Java source file, that is in charge of monitoring the execution of the observed methods.

When the main program execution is over, the Coverage Report Manager displays a table that informs of the coverage of the nodes/edges/paths of the JML method specification graph, for each condition rewriting that can be applied.

jmlCoverage has been developed, by choice, independently from the JML Runtime Assertion Checker. The tool supports the same functionalities as the RAC and, as a consequence, it requires the JML expressions to be executable (i.e., by iterating `\forall` or `\exists` over a finite range of integers). Basically, all constructs accepted by the JML RAC can be accepted by the tool. Its use is independent from the RAC and, whereas it is not its first intent, jmlCoverage is also able to detect postconditions that are not established by the implementation.

The next section reports on the use of the jmlCoverage tool within a realistic case study.

5.3 Experiments

5.3.1 Target program.

We have experimented our approach on a case study, adapted from an industrial example, named Demoney [19]. Demoney is an applet designed by Trusted Logics, implementing an electronic purse. For the experimental prupose, we have developed a simplified version of the implementation, which had been previously annotated with JML to describe its functional behavior. The classes of the application represent about 500 lines of JML spread in 4 classes.

5.3.2 Selected testing tools.

Then we have selected two (semi-)automated test generation tools, for which we wanted to evaluate the test suite generation capabilities. We have selected a random testing tool, Jartege [23], and a combinatorial testing tool, Tobias [18]. Jartege produces a given number of sequences, each sequence being of a given length, and composed of randomly selected method invocations using random inputs. On the other hand, Tobias is able to produce large combinatorial test suites from a test schema defined as a regular expression. Both tools rely on the JML method specifications to filter test cases that do not fulfill the method preconditions.

```
/*@ 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;
  @*/
T meth(T1 p1, ...) throws ... {
    body;
}
```

```
T meth(T1 p1, ...) throws ... {
   Check and report precondition
   edges predicates coverage
    try {
       body;
    }
    catch(java.lang.Error e) {
       if (e instanceof E11) {
           Check and report edges
           predicates coverage for E1
       }
       ...
       if (e instanceof ENP) {
           Check and report edges
           predicates coverage for EN
       }
       throw e;
    }
   Check and report edges predicates
   coverage for normal postcondition
}
```

Figure 5. Instrumented java source code

5.3.3 Study.
First, we ran Jartege on the Demoney class. Since Jartege is a random testing tool, we were interested in evaluating the efficiency of such a tool. Its use shows the practicability of our approach, as well as an interesting feedback on the produced test suites. Indeed, the possibility to connect Jartege with jmlCoverage to help has appeared to be an interesting option. In this context, jmlCoverage can be used to limit the number of generated test cases, by generating tests until a user-defined specification coverage rate is reached. Second, we designed 5 testing schemas that Tobias unfolded in 162 test cases. The resulting abstract tests were concretized to a Java test program. We have been able to establish the overall coverage of our testing schema. Here again, our tool can be used to master the combinatorial

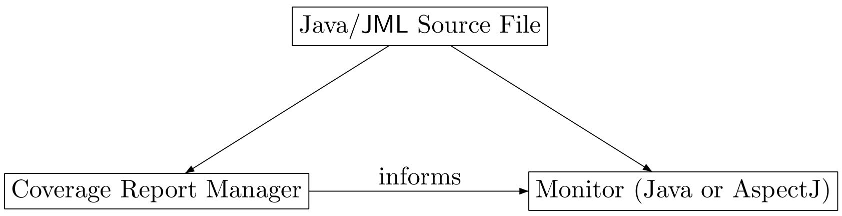

Figure 6. Principles of measuring the specification coverage

TS size × number of TS	javac	RAC	jmlCoverage	RAC+jmlCoverage
100 × 10	0.738	1.304	0.768	1.244
100 × 20	1.195	1.116	1.140	1.676
100 × 50	2.671	1.165	1.711	2.010
100 × 100	4.987	1.565	2.109	2.464

Table 2
Results of execution times on the case study (in ms.)

explosion induced by the use of schemas. For both of these tools, the specification coverage measure is a good help for a validation engineer to know whether the test suites are pertinent or not.

One interesting point is the comparison of the execution times of the test suites w.r.t. the additional annotations. Table 2 displays the execution times (in ms) of several test suites w.r.t. on programs (*i*) without any runtime verification, (*ii*) with the JML assertions checking, (*iii*) with the JML assertions coverage measure, and (*iv*) with (*ii*) and (*iii*). Notice that the test suites were automatically generated using Jartege. Notice also that execution times with (*i*) may be longer than in other cases, since the runtime checking may reveal inconclusive tests (i.e., which do not respect one method's precondition), and interrupts the execution of the corresponding test suite.

The results show that the cost of executing jmlCoverage is very little, regardless to executing the RAC. This is due to the fact that the RAC performs lot more checkings than jmlCoverage, since it systematically checks invariants and history constraints. But, the additional cost of using jmlCoverage on top of the RAC is minimal, even for larger test suites.

6 Related Work

The JML Runtime Assertion Checker [5] is already able to report a partial coverage of a JML method specification, indicating if a precondition has been covered once, more than once, or never. Nevertheless, it does not present the same granularity as our approach and can not be considered as a relevant coverage measure tool. VDMTools [11] also adopt a DBC approach. They provide coverage tools which consider pre- and postconditions as ordinary statements and measure how much of the specification has been exercised. In other words, it provides an extended notion of statement coverage which is most of the time weaker than our measures. Works have also been led on the coverage measure of UML specifications [1], especially based on the structural coverage of statecharts diagrams. Simulink Stateflow [12] is also able to perform model coverage measurement on statecharts diagrams. A complementary view of test suite measurement is the code coverage measurement, that can be achieved with tools such as JCover [14], JCoverage [15], clover [7] or EMMA [10].

The approach we have proposed is inspired of both classical control-flow graph coverage criteria [21], and classical condition coverage criteria [22]. The novelty is the application of these criteria to a predicate-based graph extracted from a JML method specification. Moreover, the interest of using a specification coverage tool instead of a code coverage is that the specification makes it possible to express properties independently from a specific implementation, and thus, allows more specific measurements, based on a black-box view of the program.

7 Conclusion and Future Works

This paper has presented an approach for measuring the coverage of JML method specifications by a Java test suite. A run-time assertion checking mechnism is employed to ensure the coverage of the graph extracted from the method specifications. The originality of this work is the application of the criteria to JML. We believe that this work can help increasing the confidence of a validation engineer in his/her test suite, even if it does not replace a code coverage analysis. From a technical point of view, the use of aspects for runtime checking of the assertions, frees us from requesting the Java source code. We only need the JML specification. As a consequence, this approach is suited to model-based testing. The work presented in this paper can be used as a basis for reducing test suites w.r.t. a defined coverage criterion so that the reduced test suite provides the same coverage as the complete one [13]. Moreover, the Java interface could be interesting for connecting Jartege, also written in Java.

One interesting point is to extend the coverage of the JML specifications to take other clauses into account, such as the class invariant or the history constraints. In addition, we would like to base the development of jmlCoverage on the RAC's architecture. This would increase the evolutions of the tool w.r.t. the evolutions of JML, and it would make it possible to reuse the assertion generation mechanisms of the RAC. Finally, the use of an annotation modeling language such as JML, leads us to consider the extension of this work to Spec# specifications, which would probably not present any difficulties.

References

[1] A. Amschler Andrews, R.B. France, S. Ghosh, and G. Craig. Test adequacy criteria for uml design models. *Software Testing, Verification and Reliability*, 13(2):95–127, 2003.

[2] B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems.* John Wiley & Sons, New York, USA, 1995.

[3] F. Bouquet, F. Dadeau, and B. Legeard. Automated boundary test generation from JML specifications. In *FM'06, 14th Int. Conf. on Formal Methods*, volume 4085 of *LNCS*, pages 428–443, Hamilton, Canada, August 2006. Springer-Verlag.

[4] F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. JML-Testing-Tools: a Symbolic Animator for JML Specifications using CLP. In *Proceedings of 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, Tool session (TACAS'05)*, volume 3440 of *LNCS*, pages 551–556, Edinburgh, United Kingdom, April 2005. Springer-Verlag.

[5] Y. Cheon and G.T. Leavens. A Runtime Assertion Checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *Proceedings of the International Conference on*

Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002, pages 322–328. CSREA Press, June 2002.

[6] Y. Cheon and G.T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In Boris Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming, 16th European Conference, Máalaga, Spain, Proceedings*, volume 2374 of *LNCS*, pages 231–255, Berlin, June 2002. Springer-Verlag.

[7] The Clover web site. http://cenqua.com/clover/, 2006.

[8] D. Deharbe and S. Ranise. Light-weight theorem proving for debugging and verifying units of code, 2003.

[9] D. Detlefs, G. Nelson, and J.B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[10] The EMMA web site. http://emma.sourceforge.net/, 2006.

[11] The VDM Tool Group. VDM-SL Toolbox User Manual. Technical report, IFAD, October 2000. ftp://ftp.ifad.dk/pub/vdmtools/doc/userman_letter.pdf.

[12] G. Hamon and J.M. Rushby. An operational semantics for stateflow. In M. Wermelinger and T. Margaria, editors, *FASE*, volume 2984 of *Lecture Notes in Computer Science*, pages 229–243. Springer, 2004.

[13] M.J. Harrold, R. Gupta, and M.L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, 1993.

[14] The JCover web site. http://www.codework.com/JCover/product.html, 2006.

[15] The JCoverage web site. http://www.jcoverage.com/, 2006.

[16] G.T. Leavens, A.L. Baker, and C. Ruby. JML: a Java Modeling Language. In *Formal Underpinnings of Java Workshop (at OOPSLA '98)*, October 1998.

[17] G.T. Leavens, A.L. Baker, and C Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[18] Y. Ledru, L. du Bousquet, O. Maury, and P. Bontron. Filtering tobias combinatorial test suites. In Michel Wermelinger and Tiziana Margaria, editors, *Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004*, volume 2984 of *Lecture Notes in Computer Science*, pages 281–294, Barcelona, Spain, 2004. Springer-Verlag.

[19] R. Marlet and C. Mesnil. Demoney: A demonstrative electronic purse - card specification. Technical Report SECSAFE-TL-007, SecSafe, 2002.

[20] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2^{nd} edition, 1997. MEY b 88:1 2.P-Ex.

[21] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[22] A.J. Offutt, Y. Xiong, and S. Liu. Criteria for generating specification-based tests. In *Proceedings of the 5^{th} IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'99)*, pages 119–131, Las Vegas, USA, October 1999. IEEE Computer Society Press.

[23] Catherine Oriat. Jartege: A tool for random generation of unit tests for java classes. In *Proccedings of the Second International Workshop on Software Quality, SOQUA 2005*, volume 3712 of *Lecture Notes in Computer Science*, pages 242–256, Erfurt, Germany, September 2005. Springer-Verlag.

[24] Norbert Oster. Automated generation and evaluation of dataflow-based test data for object-oriented software: A tool for random generation of unit tests for java classes.. In *Proccedings of the Second International Workshop on Software Quality, SOQUA 2005*, volume 3712 of *Lecture Notes in Computer Science*, pages 212–226, Erfurt, Germany, September 2005. Springer-Verlag.

[25] A.D. Raghavan and G.T. Leavens. Desugaring JML method specifications. Technical Report 00-03e, Iowa State University, Department of Computer Science, May 2005.