Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 190 (2007) 73–84

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Case Study in Matching Test and Proof Coverage

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

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

Abstract

This paper studies the complementarity of test and deductive proof processes for Java programs specified in JML (Java Modeling Language). The proof of a program may be long and difficult, especially when automatic provers give up. When a theorem is not automatically proved, there are two possibilities: either the theorem is correct and there are not enough pieces of information to deal with the proof, or the theorem is incorrect. In order to discriminate between those two alternatives, testing techniques can be used. Here, we present experiments around the use of the JACK tool to prove Java programs annotated with JML assertions. When JACK fails to decide proof obligations, we use a combinatorial testing tool, TOBIAS, to produce large test suites that exercise the unproved program parts. The key issue is to establish the relevance of the test suite with respect to the unproved proof obligations. Therefore, we use code coverage techniques: our approach takes advantage of the statement orientation of the JACK tool to compare the statements involved in the unproved proof obligations and the statements covered by the test suite. Finally, we ensure our confidence within the test suites, by evaluating them on mutant program killing exercises. These techniques have been put into practice and are illustrated by a simple case study.

Keywords: deductive proof process, combinatorial testing, test suite relevance evaluation, JML

1 Introduction

Software testing has emerged as one of the major techniques to evaluate the conformance between a specification and some implementation. Unfortunately, testing only reveals the presence of errors and conformance may only be totally guaranteed by formal proof, exhaustive testing with respect to some criteria or a combination of both.

Formal proof techniques are often very difficult to apply because most of them require a high level of mathematical skills. In order to make proof techniques

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

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

[3] Email: Frederic.Dadeau@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.2007.08.007

available to non-specialists, significant efforts have been spent in the last years to provide automatic deductive proof tools. For instance, the JACK tool [2] has been designed to automatically prove the correctness of some Java code with respect to its JML specification. JML (Java Modeling Language [9]) is a formal specification language for Java, based on assertions such as invariants, pre- and post-conditions. During the proof process, JACK firstly produces the proof obligations required to show that the implementation (i.e. the code) conforms to its specification. Then, JACK tries to automatically prove each of them.

Nevertheless a difficulty remains; when a theorem is not automatically proved, there are two possibilities: either the theorem is correct and the prover is unable to decide it, or the theorem is just simply incorrect.

In these cases, testing may contribute to exhibiting that there is an error in the code (or in the specification), or increase the confidence that the theorem is correct (meaning that the implementation conforms to its specification). To achieve that, our proposal is to produce a huge number of tests, with combinatorial testing techniques. Then, the relevance of the test suite has to be evaluated. Indeed, if the test cases are not related to the part of the program involved in the proof obligation, they cannot provide feedback about the correctness of the theorem to be proved. In this work, the relevance of the test suite is evaluated with coverage measurements.

Section 2 introduces the principles of JML, an executable model-based specification language. Section 3 gives an overview of the JACK prover. A testing process is described in Sect. 4 along with a brief presentation of TOBIAS tool. Section 5 and 6 present the evaluation of the quality of the test suite by respectively using coverage and mutation techniques. Finally Section 8 draws the conclusions and presents the perspectives of this work.

2 Using JML as a Specification Language

2.1 A Small Example

We use a simple buffer system as a running example. This system is composed of three buffers. Each buffer is modeled by an integer value, which indicates the number of elements in it. The system state is given by the three variables, `b1`, `b2` and `b3`. The maximum size of the system is 40 elements. The system has to distribute the elements among the buffers so that: buffer `b1` is smaller or equal than `b2`, which is smaller or equal than `b3`. The difference between `b1` and `b3` should not exceed 15 elements. These constraints leave some freedom in the manner the elements can be shared between the buffers. For example, 30 elements can be stored with `b1=5 b2=10 b3=15` or with `b1=8 b2=10 b3=12`.

Three methods are provided to modify the system. `Init` resets all buffers to zero. `Add(x)` increases the total number of elements of the system by `x` ($x > 0$) by adding `x` elements to the buffers; these elements are distributed in `b1`, `b2`, and `b3`. `Remove(x)` decreases the total number of elements in the system by `x` ($x > 0$) by removing `x` elements from the buffers.

```
 1 public class Buffer{
   public int b1;
   public int b2;
   public int b3;
 5
   /*@ public invariant  b1+b2+b3<=40 ; @*/
   /*@ public invariant  0<=b1 ; @*/
   /*@ public invariant  b1<=b2 ; @*/
   /*@ public invariant  b2<=b3 ; @*/
10 /*@ public invariant  b3-b1<=15; @*/

   /*@ requires true;
     @ modifies b1, b2, b3;
     @ ensures b1==0 && b2==0 && b3==0;
15 */
   public Buffer(){
       b1 = 0;
       b2 = 0;
       b3 = 0;
20 }

   /*@ requires true;
     @ modifies b1, b2, b3;
     @ ensures b1==0 && b2==0 && b3==0;
25 */
   public void Init(){
       b1 = 0;
       b2 = 0;
       b3 = 0;
```

```
30 }

   /*@ requires x>=0 && x<=5 && b1+b2+b3+x<=40;
     @ modifies b1, b2, b3;
     @ ensures b1+b2+b3==\old(b1+b2+b3)+x;
35 */
   public void Add(int x){
       if ((b1+x) <= b2)
         {b1=b1+x;}
       else
40       if ((b2+x) <= b3)
           {b2=b2+x;}
         else
           {b3=b3+x;}
   }
45
   /*@ requires x>=0 && x<=5 && x<=b1+b2+b3;
     @ modifies b1, b2, b3;
     @ ensures b1+b2+b3==\old(b1+b2+b3)-x;
   */
50 public void Remove(int x){
       if ((b3-x) >= b2)
         {b3=b3-x;}
       else
         if ((b2-x) >= b1)
55         {b2=b2-x;}
         else
           {b1=b1-x;}
   }
```

Fig. 1. Buffer: JML specification and a possible Java implementation

2.2 The Java Modeling Language

The Java Modeling Language (JML) [10] is an annotation language used to specify Java programs by expressing formal properties and requirements on the classes and their methods. The Java syntax of JML makes it easier for Java programmers to read and write specifications. The core expression of the language is based on Java, with new keywords and logical constructions.

We illustrate the JML syntax on the buffer example given in Fig. 1. The JML specification appears within special Java comments, between /*@ and @*/ or starting with //@. The specification of each method precedes its interface declaration. This follows the usual convention of Java tools, such as JavaDoc, which put such descriptive information in front of the method.

JML annotations rely on three kinds of assertions: class invariants, preconditions and postconditions. Invariants have to hold in all visible states. A visible state roughly corresponds to the initial and final states of any method invocation [9]. The invariant stated in Fig. 1, line 6, indicates that the maximum size of the system is 40 elements. JML relies on the principles of Design by Contract [12] which states that to invoke a method, the system must satisfy the method precondition, and as a counterpart, the method has to establish its postconditions.

A method's precondition is given by the *requires* clause. In our example, `Init` precondition is set to true (see Fig. 1, line 22). The preconditions of `Add` and `Remove` (lines 32 and 46) are such that the number of element to be added into (resp. removed of) the system is positive, less or equal to 5 and does not cause buffer overflow (resp. underflow).

Postconditions are expressed in the *ensures* clauses. For instance, the postconditions of `Add` and `Remove` (Fig. 1, lines 34 and 48) express that the add and remove operations are correctly performed.

JML extends the Java syntax with several keywords. `\result` denotes the return value of the method. It can only be used in *ensures* clauses of a non-void method. `\old(Expr)` (Fig. 1, lines 34 and 48) refers to the value that the expression `Expr` had in the initial state of a method. `\forall` and `\exists` designate universal and existential quantifiers.

3 JACK Proof Obligation Generator

The Java Applet Correctness Kit (or JACK) [2] provides an environment for the verification of Java and JavaCard programs annotated with JML. JACK aims at proving properties of a given Java class, considered in isolation; these properties are expressed as JML assertions. It implements a fully automated weakest precondition calculus that generates proof obligations (POs) from annotated Java sources. Each proof obligation is related to a path in the source code of the program.

Those proof obligations can be discharged using different theorem provers. The Jack proof manager sends the proof obligations to the different provers, and keeps track of proved and unproved proof obligations. Currently proof obligations can be generated for the B method's prover (developed by Clearsy), the Simplify theorem prover (notably used by ESC/Java), the Coq proof assistant, and PVS. For the case studies we present in this paper, Simplify was used.

JACK consists of two parts: (1) a *converter*, which is a lemma generator from Java source annotated with JML into adequate formalism (such as B lemmas, for B prover), and (2), a *viewer*, which allows developers to understand the generated lemmas. The mathematical complexity of the underlying concepts is hidden. JACK provides a dedicated proof obligation viewer that presents the proof obligations connected to execution paths within the program. Goals and hypotheses can be displayed in a Java/JML like notation.

Buffer example

Figure 1 provides an incorrect implementation of the Buffer specification. Indeed, the `Remove` method is incorrect because the statement in line 57 may set buffer `b1` to a negative value while keeping the total number of elements positive, which is forbidden by the class invariant.

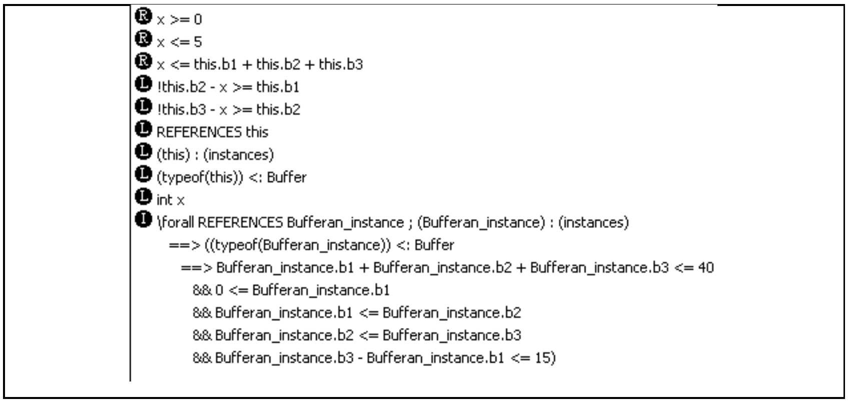
For this example, Simplify (version 1.2.0) was not able to prove 8 proof obligations. Some of them correspond to correct code and require the user to add further elements in the specification to help the prover. Others correspond to the erroneous implementation of `Remove`, and the prover will never be able to establish them. Table 1 illustrates each proof obligation and the corresponding specification and code lines. Figure 2 gives all proof obligations related to the invariant preservation by the `Remove` method.

4 The Testing Process

JML specifications can be used as oracle for a test process. In this section, we first cover some principles of conformance testing with JML, before introducing the

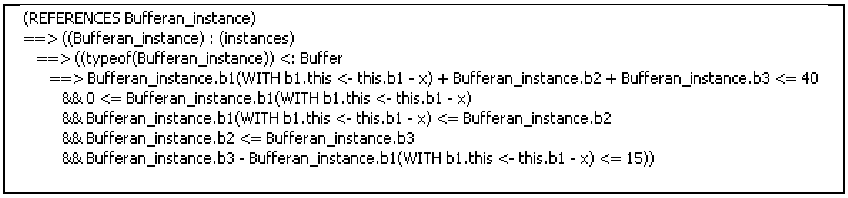
Method	PO	specification line	code lines
Buffer	1	6	16-20
	2	7	16-20
Add	3	34	37, 40, 43-44
	4	6	37, 40, 43-44
	5	9	37, 40, 43-44
Remove	6	48	51, 54, 57-58
	7	6	51, 54, 57-58
	8	7	51, 54, 57-58

Table 1
Buffer specification/code lines related to unproved PO



```
Ⓡ x >= 0
Ⓡ x <= 5
Ⓡ x <= this.b1 + this.b2 + this.b3
Ⓛ !this.b2 - x >= this.b1
Ⓛ !this.b3 - x >= this.b2
Ⓛ REFERENCES this
Ⓛ (this) : (instances)
Ⓛ (typeof(this)) <: Buffer
Ⓛ int x
Ⓘ \forall REFERENCES Bufferan_instance ; (Bufferan_instance) : (instances)
     ==> ((typeof(Bufferan_instance)) <: Buffer
       ==> Bufferan_instance.b1 + Bufferan_instance.b2 + Bufferan_instance.b3 <= 40
         && 0 <= Bufferan_instance.b1
         && Bufferan_instance.b1 <= Bufferan_instance.b2
         && Bufferan_instance.b2 <= Bufferan_instance.b3
         && Bufferan_instance.b3 - Bufferan_instance.b1 <= 15)
```

(a)



```
(REFERENCES Bufferan_instance)
==> ((Bufferan_instance) : (instances)
  ==> ((typeof(Bufferan_instance)) <: Buffer
    ==> Bufferan_instance.b1(WITH b1.this <- this.b1 - x) + Bufferan_instance.b2 + Bufferan_instance.b3 <= 40
      && 0 <= Bufferan_instance.b1(WITH b1.this <- this.b1 - x)
      && Bufferan_instance.b1(WITH b1.this <- this.b1 - x) <= Bufferan_instance.b2
      && Bufferan_instance.b2 <= Bufferan_instance.b3
      && Bufferan_instance.b3 - Bufferan_instance.b1(WITH b1.this <- this.b1 - x) <= 15))
```

(b)

Fig. 2. One proof obligation related to the implementation given Fig. 1

TOBIAS tool.

4.1 JML as a Test Oracle

JML has an executable character. It is possible to use invariant assertions, as well as pre- and postconditions as an oracle for conformance testing. JML specifications are translated into Java by the `jmlc` tool, added to the code of the specified program, and checked against it, during its execution.

The executable assertions are thus executed before, during and after the execution of a given operation. Invariants are properties that have to hold in all *visible states*. A visible state roughly corresponds to the initial and final states of any method invocation [9]. When an operation is executed, three cases may happen.

All checks succeed: the behavior of the operation conforms with the specification for these input values and initial state. The test delivers a PASS verdict. An **intermediate or final check fails**: this reveals an inconsistency between the behavior of the operation and its specification. The implementation does not conform to the specification and the test delivers a FAIL verdict. An **initial check fails**: in this case, performing the whole test will not bring useful information because it is performed outside of the specified behavior. This test delivers an INCONCLUSIVE verdict. For example, $\sqrt{x}$ has a precondition that requires x to be positive. Therefore, a test of a square root method with a negative value leads to an INCONCLUSIVE verdict.

4.2 Test Cases Definition

We define a test case as a sequence of operation calls. For example, in the following, test case `TC1` initializes the buffer system, adds two elements and removes one of them.

```
TC1 : Init() ; Add(2) ; Remove(1)
TC2 : Init() ; Add(-1)
TC3 : Init() ; Add(2) ; Remove(3)
TC4 : Init() ; Add(3) ; Remove(2) ; Remove(1)
```

Each operation call may lead to a PASS, FAIL or INCONCLUSIVE verdict. As soon as a FAIL or INCONCLUSIVE verdict happens, we choose to stop the test case execution and mark it with this verdict. A test case that is carried out completely receives a PASS verdict.

In the context of the buffer specification, the test cases `TC2` and `TC3` should produce an INCONCLUSIVE verdict: in `TC2`, Add is called with an incorrect negative parameter, in `TC3`, one tries to remove more than what has been added so far. If tests `TC1` and `TC4` are executed against a "correct" implementation, they should produce a PASS.

4.3 Test Case Generation

Combinatorial testing performs combinations of selected input parameters values for given operations and given states. For example, a tool like JML-JUnit [3] generates test cases which consist of a single call to a class constructor, followed by a single call to one of the methods. Each test case corresponds to a combination of parameters of the constructor and parameters of the method.

TOBIAS is a test generator based on combinatorial testing [4]. It adapts combinatorial testing to the generation of sequences of operation calls. The input of TOBIAS is composed of a test pattern (also called test schema) which defines a set of test cases. A schema is a bounded regular expression involving the Java methods and their associated JML specification. TOBIAS unfolds the schema into a set of sequences, then computes all combinations of the input parameters for all operations of the schema.

The schemas may be expressed in terms of *groups*, which are structuring facilities that associate a method, or a set of methods, to typical values. Groups may also involve several operations. Let S2 be a schema:

$$\begin{cases} \texttt{S2} = \texttt{BufGr\^{}\{1..3\}} \text{ with} \\ \texttt{BufGr} = \{\texttt{Init}()\} \cup \{\texttt{Add}(x) | x \in \{1,2,3,4,5\}\} \cup \{\texttt{Remove}(y) | y \in \{2,3,5\}\} \end{cases}$$

BufGr is a set of (1+5+3)=9 instantiations. The suffix ^{1..3} means that the group is repeated 1 to 3 times. S2 is unfolded into 9+(9*9)+(9*9*9)=819 test sequences.

Several case studies [1,6] have shown that TOBIAS increases the productivity of the test engineer by allowing him to generate thousands of test cases from a few lines of schema description.

TOBIAS includes tools to turn such a test suite into a JUnit file. The execution of contructor Buffer is automatically added at the beginning of all test sequences. Executing this test suite against the erroneous buffer implementation (given in Fig. 1) reveals failures: 17 tests fail, 378 succeed and 424 are inconclusive. All failing tests report that the error occurs when checking line 7 of the invariant after the execution of Remove. This corresponds to PO #8. JML does not allow to point out a particular statement where the error was introduced because assertions are only checked at the exit of the operation.

5 Test Coverage Measurement and Proof Obligations

At this point of the study, we know that there is an error in operation Remove, and that PO #8 is false. Can we get more confidence from the tests that the remaining POs are correct?

Line coverage reports whether each executable statement is encountered. It is also known as statement coverage, segment coverage [14], C1 and basic block coverage. Basic block coverage is the same as statement coverage except that the unit of code measured is each sequence of non-branching statements.

For the 819 test cases generated from S2 JCoverage [8] reports that 100% of the Java statements have been executed. So, at least all the operations have been covered, and all JML assertions have been evaluated while exiting these operations. But, at this point, nothing guarantees that the path of each proof obligation has been covered by a test.

To evaluate if a test case $\texttt{TC}_i$ is relevant w.r.t. an unproved PO $\texttt{PO}_k$, a first simple idea is to evaluate the line coverage associated with $\texttt{TC}_i$ and to compare it with the lines of the $\texttt{PO}_k$ path. As a first approximation, if $\texttt{TC}_i$ does not execute all code lines associated to $\texttt{PO}_k$, then it does not follow the same path, and $\texttt{TC}_i$ is then not relevant to have an idea on $\texttt{PO}_k$'s correctness. If $\texttt{TC}_i$ executes at least all code lines associated to $\texttt{PO}_k$, then it may follow the same path. So $\texttt{TC}_i$ may be relevant to increase the confidence in $\texttt{PO}_k$'s correctness.

For the Buffer example, we have executed the 819 test cases and we have analysed their line coverage with JCoverage. We gathered test cases with respect to their line code coverage into 25 "packets", as described in Table 2. All test cases of a given packet cover the same lines (presumably with different values).

All 17 failed tests belong to packet #15. A closer look at the coverage of packets #14 to #17 shows that they share the same coverage of the `Buffer` constructor and operation `Add`. We can also notice that, among these 4 packets, line 57 is only executed in packet #15. We know that the error was related to operation `Remove`; this closer look at coverage information suggests that the error is located at line 57.

Still, our main concern is to increase our confidence in unproved POs. Table 2 tells us that all test cases were related to POs #1 and #2. More than 500 test cases are related to POs #3, #4 and #5, and more than 400 test cases are related to POs #6, #7 and #8. Since failed tests were only related to PO #8, we can increase our confidence in the unproved POs, since each of them has been tested several hundred times[5] and did not reveal any error.

[5] To be more accurate, we should have excluded inconclusive tests from Table 2. We intend to include this improvement in a future version of our reporting tool.

Packet	lines of code	# of TC	PO
1	16-20, 27-30	3	1,2
2	16-20, 27-30, 37, 40, 43-44	70	1,2,3,4,5
3	16-20, 27-30, 37, 40, 43-44, 51, 54, 57-58	54	1,2,3,4,5,6,7,8
4	16-20, 27-30, 37, 40, 43-44, 51-52, 58	16	1,2,3,4,5
5	16-20, 27-30, 37, 40-41, 43-44	30	1,2,3,4,5
6	16-20, 27-30, 37-38, 44, 51, 54, 57-58	20	1,2,6,7,8
7	16-20, 27-30, 51, 54, 57-58	30	1,2,6,7,8
8	16-20, 27-30, 51, 54-55, 57-58	12	1,2,6,7,8
9	16-20, 37, 40, 43-44	8	1,2
10	16-20, 37, 40, 43-44, 51, 54, 57-58	32	1,2,3,4,5,6,7,8
11	16-20, 37, 40, 43-44, 51, 54-55, 57-58	23	1,2,3,4,5,6,7,8
12	16-20, 37, 40, 43-44, 51-52, 54, 57-58	38	1,2,3,4,5,6,7,8
13	16-20, 37, 40, 43-44, 51-52, 58	71	1,2,3,4,5
14	16-20, 37, 40-41, 43-44	102	1,2,3,4,5
15	16-20, 37, 40-41, 43-44, 51, 54, 57-58	31	1,2,3,4,5,6,7,8
16	16-20, 37, 40-41, 43-44, 51, 54-55, 58	14	1,2,3,4,5,6,7,8
17	16-20, 37, 40-41, 43-44, 51-52, 58	18	1,2,3,4,5
18	16-20, 37-38, 40, 43-44, 51, 54, 57-58	76	1,2,3,4,5,6,7,8
19	16-20, 37, 40-41, 44, 51, 54-55, 57-58	12	1,2,6,7,8
20	16-20, 37-38, 40-41, 43-44	35	1,2,3,4,5
21	16-20, 37-38, 44, 51, 54, 57-58	63	1,2,6,7,8
22	16-20, 37-38, 44, 51, 54-55, 57-58	12	1,2,6,7,8
23	16-20, 51, 54, 57-58	6	1,2,6,7,8
24	16-20, 51, 54-55, 57-58	23	1,2,6,7,8
25	16-20, 51-52, 54-55, 57-58	10	1,2,6,7,8
	total of TC	819	

Table 2
25 packets for the 819 Buffer example tests

6 Test Relevance Analysis using Mutation

Another way to measure the quality of the test set is to evaluate its fault detection capabilities. Mutation analysis can be used for this purpose.

Mutation analysis is based on seeding the implementation with a fault by applying a mutation operator, and checking whether test set identifies this fault or not [5,11]. A mutated program is called a mutant. A mutant is said to be killed if the test suite reveals its error.

An appropriate set of mutation operators should be representative of classical programming errors. The idea behind mutation testing is quite simple: if a test suite kills all mutants generated by these operators then, since it is able to find these small differences, it is likely to be good at finding real faults.

When using mutation programs such as MuJava [11,13], two kinds of problems may arise. First, applying a large set of mutation operators to a real-size program usually results in a huge number of mutants. Secondly, some of the mutants are actually equivalent to the original program and can not be killed.

In order to limit the number of mutants, we applied mutations only to statements that are involved in the path related to unproved POs. For instance, we generated 20 mutants corresponding to the unproved PO #7 and our test suite killed 100% of them. An interesting point is that different tests of a same packet may kill different mutants. This means that these packets feature some kind of diversity.

At this point of the case study, we have reached sufficient confidence in the correctness of the remaining proof obligations to get back to an interactive proof activity. Actually, only POs #1 to #5 deserve to be proved at this stage, because corrections of `Remove` will not affect their correctness. Of course, nothing guarantees that our test suite was able to detect all kinds of subtle errors. This is why a final proof activity is definitely needed to assess program correctness. Still, the benefit of our testing activity is that the validation engineer will not waste time trying to prove false proof obligations, or even correct ones such as #6 or #7 which may be affected by the correction of `Remove`.

7 Banking Application Case Study

Industrial Case Study.

The combination of the proof/test processes was experimented on an industrial case study provided by Gemplus (a smart card manufacturer). The case study is a banking application which deals with money transfers [1]. It has been produced by Gemplus Research Labs, and used for the first experimentation of JACK. This case study is somehow representative of Java applications connected to smart cards. The application user (i.e. the customer) can consult his accounts and make some money transfers from one account to another. The user can also record some “transfer rules”, in order to schedule regular transfers. These rules can be either saving or spending rules.

The case study is actually a simplified version of an application already used in the real world, written with 500 LOC, distributed into 8 classes. The specification is given in JML. Most preconditions are set to true. Since the application deals with money, and since some users may have malicious behaviors, the application is expected to have defensive mechanisms. Thus, it is supposed to accept any entry, but it should return error messages or raise exceptions if the inputs are not those expected for a nominal behavior.

In order to evaluate our approach, we worked on a correct version of the program, and introduced an error in the `Currency_src` class.

Proof Process.

We used JACK with Simplify for this example. For some unknown reason, we were only able to compile 7 of the 8 classes with the JACK tool. Table 3 reports on the total number of POs generated by JACK and the number of POs which remained unproved after the automatic proof process.

Test Process.

For each class, we produced only one TOBIAS schema. They were rather straightforward, as `S2` in the previous case study. Their design and unfolding with the TOBIAS tool only took us a few minutes. Each schema produced between 48 and 1024 test cases. We then executed them, and, as expected, failed tests were only related to `Currency_src`.

Test Coverage and Proof Obligations.

We grouped our tests into packets on the basis of statement coverage. As Table 3 reports, most classes correspond to a small number of packets. This motivates further research using some path coverage tool instead of statement coverage, in order to have a more accurate distribution of test cases.

Killing Mutants.

The mutation analysis was not possible for `Account` and `Rule` due to unsolved technical problems. For the other classes, we could notice that all mutants were killed for `Balances_src` and `Transfers_src`. However, no mutant have been killed for `SavingRule` and `SpendingRule`. Clearly, testing schemas for those two classes were not relevant enough. More insightful test schemas must be defined to generate appropriate test suites for these classes and increase the confidence in their correctness.

8 Conclusion and Perspectives

This paper has proposed an approach to combine a proof process with testing activities. The goal is to help the validation engineer decide on the correctness of unproved proof obligations. Our testing process starts from a combinatorial testing phase where hundreds of test cases are generated with small design efforts (few schemas can generate several thousand of test cases). Their execution may reveal errors in the code under validation and hence point out false proof obligations. The huge number of succeeded tests, and an evaluation of the quality of the test suite, should increase the confidence in the remaining proof obligations.

Two techniques are proposed to evaluate the quality of the test suite: a comparison of statement coverage with the paths related to unproved proof obligations, and an assessment of the fault capabilities based on mutation testing. The first evaluation technique results in a distribution of the test suite into "packets" of tests which cover the same set of statements. The second evaluation restricts mutations to those which hit statements appearing in the path of unproved PO. The assessment of the quality of the test suite can be further refined by crossing information gathered from these two techniques. Different tests grouped in the same packet exhibit more "diversity"if they kill different mutants.

The approach was experimented on two case studies in the context of Java/JML. We used the JACK proof environment, the TOBIAS test generator, and JCoverage and MuJava for quality assessment.

We divide the future work into four points as below:

Statement vs Path coverage. Since JACK is based on the notion of path, it makes sense to use path coverage instead of statement coverage. Besides the fact that we do not have such a tool available in our environment, we suspect that this more detailed analysis will slow down the testing process, and may in some cases result into over-detailed test reports. Therefore, we believe that it should be provided as an option.

Automatic process. Our approach only makes sense if the whole testing process is cheaper than interactive proof activities. Here each step is automated. JACK associates automatically PO to paths. Tests can be generated automatically thanks to combinatorial tools, such as TOBIAS. JCoverage analyses automatically lines covered during test execution. Grouping test cases is done by sorting JCoverage results. Killing mutants is done automatically.

Reporting. In the previous section, we mentioned the potential interest of crossing packet information with mutation scores. The tool which will compute this information has still to be developed. Further effort should be dedicated to provide the

Class	# PO	# unproved PO	# TC	# TC packets	#Mutant	# killed
`Account`	8	2	84	3	*MuJava exception*	
`Balances_src`	114	96	72	4	10	10
`Currency_src`	92	16	244	32	17	12
`Rule`	17	9	72	4	*MuJava exception*	
`SavingRule`	62	31	48	2	5	0
`SpendingRule`	37	20	48	2	5	0
`Transfers_src`	1170	560	1024	3	81	81
`AccountMan_src`	*JACK (version 1.6.7) was not able to compile this class*					

Table 3
Banking example

user with synthetic reports on the quality of his tests (describing how relevant tests are with respect to coverage, mutation analysis, ...).

Feeding assertions into the proof process. The tests generated with TOBIAS are designed independently of the structure of the code or the specification. We expect that they could provide interesting input to the Daikon invariant generator [7]. This would allow to feedback of the proof process with assertions generated from the tests, resulting in a secondary benefit of the testing activity.

References

[1] L. du Bousquet, Y. Ledru, O. Maury, C. Oriat, and J.-L. Lanet. A case study in JML-based software validation (short paper). In *Proceedings of 19th Int. IEEE Conf. on Automated Sofware Engineering (ASE'04)*, pages 294–297, September 2004.

[2] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: a developer-oriented approach. In *the 12th International FME Symposium*, Pisa, Italy, September 2003.

[3] Y. Cheon and G.T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *ECOOP 2002*, volume 2474 of *LNCS*, pages 231–255. Springer, 2002.

[4] D.M. Cohen, S.R. Dalal, J. Parelius, and G.C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 13(5):83–88, 1996.

[5] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[6] S. Dupuy-Chessa, L. du Bousquet, J. Bouchet, and Y. Ledru. Test of the ICARE platform fusion mechanism. In *12th International Workshop on Design, Specification and Verification of Interactive Systems*, volume 3941 of *LNCS*, pages 102–113. Springer, 2005.

[7] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Eng.*, 27(2):99–123, 2001.

[8] The JCoverage Home Page, 2005. http://www.jcoverage.com/.

[9] The JML Home Page, 2005. http://www.jmlspecs.org.

[10] G.T. Leavens, A.L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.

[11] Y.-S. Ma, J. Offutt, and Kwon Y. R. Mujava : An automated class mutation system software testing. *Verification and Reliability*, 15(2):97–133, June 2005.

[12] B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, October 1992.

[13] The MuJava Home Page, 2005. http://www.isse.gmu.edu/faculty/ofut/mujava/.

[14] Simeon Ntafos. A Comparison of Some Structural Testing Strategies. *IEEE Trans. Software Eng.*, 14(6):868–874, June 1988.