A case study in JML-based software validation

L. du Bousquet, Y. Ledru, O. Maury, C. Oriat
LSR-IMAG,
BP 72,
38402 St-Martin-d'Hères, France
{ldubousq, ledru, maury, oriat}@imag.fr

J.-L. Lanet
Gemplus Research Labs
La Vigie, av. du Jujubier,
13705, La Ciotat cedex, France
Jean-Louis.Lanet@gemplus.com

Abstract

This paper reports on a testing case study applied to a small Java application, partially specified in JML. It illustrates that JML can easily be integrated with classical testing tools based on combinatorial techniques and random generation. It also reveals difficulties to reuse, in a testing context, JML annotations written for a proof process.

An experiment in JML-based software validation

L. du Bousquet Y. Ledru O. Maury C. Oriat
LSR-IMAG,
BP 72,
38402 St-Martin-d'Hères, France
{ldubousq, ledru, maury, oriat}@imag.fr

J.-L. Lanet
Gemplus Research Labs
La Vigie, av. du Jujubier,
13705, La Ciotat cedex, France
Jean-Louis.Lanet@gemplus.com

Abstract

This paper reports on testing experiments applied to a small Java application, partially specified in JML. These experiments took advantage of the executable subset of JML to act as an oracle for the testing process. JML played a central role in the validation of this application. First the JML specification was developed during a proof process. Then the resulting specification acted as oracle in a subsequent testing process. This paper reports on this second step. It illustrates that JML can easily be integrated with classical testing tools based on combinatorial techniques and random generation. It also reveals difficulties to reuse, in a testing context, formal JML annotations originally intended to support a proof process.

1 Introduction

The automation of validation activities (test and proof) often requires a model or specification of the system under validation. In a classical waterfall process, such models are developed in early activities, during requirements and specification phases. But the model can also be developed later, when the validation activity actually takes place. From an economical point of view, the development of a model may be an expensive task and it makes sense to try to reuse the same model in several development activities. This also ensures some consistency between these phases. This paper reports on an experiment to reuse a JML specification, developed for a proof activity, during a testing phase.

The formal methods community has designed a wide variety of executable specification languages. Recently, the advent of the Java Modeling Language (JML) has provided an interesting tool in the context of Java programs testing. JML is a behavioral interface specification language that can be used to specify Java modules [10, 12, 13]. From a practitioner's point of view, the syntax of JML, based on Java, makes it easier to read and to write specifications.

This paper reports on a testing experiment carried out during a cooperation between the LSR laboratory and Gemplus. In the past years, Gemplus has led several research projects related to the formal development of smart card applications [4], mainly using the B method [1]. JML was adopted in recent projects dedicated to the proof of Java Card applications using the Jack tool [3].

Proving a piece of code guarantees its correctness with respect to its formal specification, which may appear very appealing in the context of critical applications. Unfortunately, the undecidable character of their underlying logics makes it impossible for theorem provers to automatically discharge all proof obligations for a given program. In many cases, 10 to 20% of the proofs must be carried out interactively which may require significant efforts from skillful engineers. Moreover, the proof only guarantees the conformance of the code to the properties expressed in the specification. In many cases, the specification does not cover the full requirements document. Requirements may be left out of the specification, either because they are forgotten by the specifiers, or because they are known to be difficult to express or to prove. In this context, additional validation activities, such as testing, must be undertaken.

Objectives of the Experiment The experiment reported in this paper addressed the validation of a Java application specified in JML. It is a simplified banking application, considered by the Gemplus researchers as representative of critical applications in the context of smart cards.

Fig. 2 summarizes the development process followed by this application. Starting from a requirements document, a java program was developed, using classical development techniques. The program entered then a first validation step based on proof techniques. This step led to the development of a JML specification of the banking application, and to the discovery and correction of several errors. This activity was carried out by Gemplus researchers, and is partially reported in [3]. Actually, [3] reports that 5 of the 8 classes of Fig. 1 were concerned by the proof process and that 944 of the 1055 proof obligations were proved automatically.

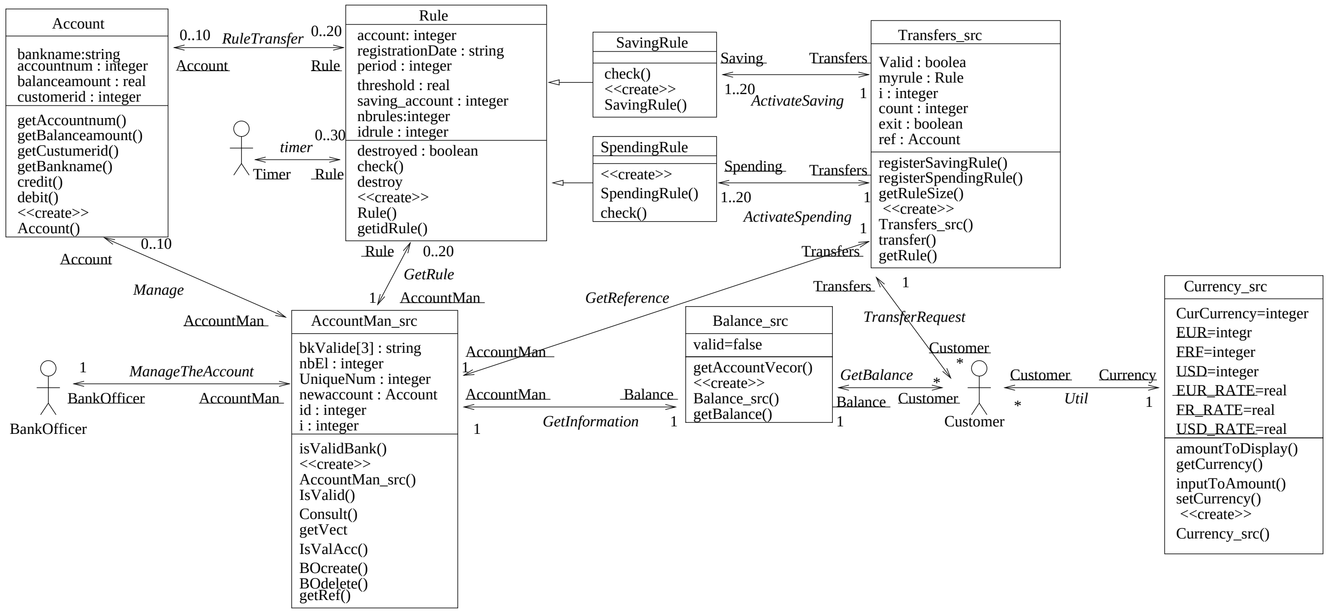

Figure 1. Class diagram of the banking application

This paper reports on a subsequent validation step which used the JML specification as an executable oracle, but also took into account the requirements document. This second validation step was carried out by the LSR team, with little visibility on the previous proof activity. In particular, the LSR team did not know which parts of the requirements were not formally specified, and which parts of the code and the specification were not proved automatically. Our experiments tried to answer the following questions:

1. Is JML, written for proof, easily reusable for test?
2. Is JML well-suited for validation by test?

Moreover, this allowed us to experiment several testing tools developed in our laboratory on an independently developed case study.

In this experiment, test data were produced by two different teams, using different techniques. The first one used a code review and random testing. The second team used a combinatorial testing approach. Each team worked during a limited time period. We have then compared the number of errors found and analysed how the errors were found (JML oracle or human critical analysis). These results were reported to the Gemplus researchers.

The whole experiment shows that JML can be used as a test oracle with several kinds of testing tools (here combinatorial and random testing tools). It reveals that JML annotations written for proof are not necessarily adapted to a testing activity, and must be written with testability in mind as a major concern. Although this experiment did use classical testing and validation techniques (random and combinatorial testing, code reviews), they turned out to be quite efficient and were easily adapted to JML.

The paper is structured in 5 sections. Sect. 2 surveys the application under test. Sect. 3 is a short overview of JML, illustrated by the case study. Sect. 4 presents the testing experiments. Sect. 5 tries to answer the questions of this introduction and draws the conclusions of this work.

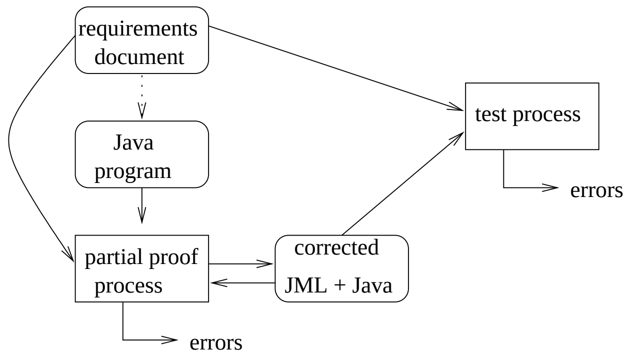

Figure 2. Validation process

2 Case study

The case study is a small banking application which deals with money transfers. The application administrator (the bank officer) can create accounts. 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 periodical transfers. These transfer rules can be either saving or spending rules. Moreover, the application includes some features to convert money from one currency to another.

The case study is a simplified version of a real application. This application is running on a central server, which is linked to several smart card terminals. For the simplified case study, the smart card terminals have been withdrawn.

The banking application code is composed of eight classes (Fig. 1), among which:

- an account class,
- an account manager that creates and deletes accounts,
- a transfer class that defines spending and saving rules to transfer money from an account to another according to different thresholds,
- a balance class that allows the customer to have access to his accounts,
- and the currency converter.

The three remaining classes are dedicated to the definition of the transfer rule principles.

Some metrics of this application are given in Table 1. The fact that the number of JML annotation lines is larger than the Java code length is mainly due to the proof process. Moreover, although the Gemplus researchers are quite used to formal specification and proof, it was their first use of JML for proof purposes.

3 JML: language and tools

3.1 The JML language

JML is a language designed 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 some new keywords and logical constructions. For a detailed JML description, see [10, 12].

Classes	Java lines	JavaDoc lines	JML lines
Transfer_src	116	34	150
AccountMan_src	105	51	236
Currency_src	93	20	28
Balance_src	64	38	58
Spending_rule	40	33	42
Saving_rule	40	33	42
Rule	40	22	23
Account	30	20	36
Total	518	251	615

Table 1. Metrics of the banking application

```
/** Copyright (c) 2002 GEMPLUS group.
 *  All Rights Reserved.
 *-----------------------------------------
 *  Project name:  COTE  - Case Study -
 *  Version 1.0 1/9/2002
 *-------------------------------------  */
package banking;

public class Currency_src {
  private static final int EUR = 1;
  //@ private invariant EUR == 1;
  private static final float EUR_RATE=6.55957f;
  //@ private invariant EUR_RATE == 6.55957f;
  private static final int FRF = 2;
  /*@ private invariant FRF == 2; @*/
  private static final float FRF_RATE =1.0f;
  /*@ private invariant FRF_RATE ==1.0f; @*/
  private /*@ spec_public */ int CurCurrency;
  /*@ private invariant (CurCurrency >=0
                      && CurCurrency <3); @*/

/*@ requires true;
  @ exsures (Exception e) false;  @*/
  public String amountToDisplay(float amount){
    float rate;
    String toDisplay;
    switch (CurCurrency) {
      case (EUR) : rate = EUR_RATE; break;
      case (FRF) : rate = FRF_RATE; break;
      default : rate = 1;
    }
    toDisplay = (Double.toString
        (Math.round((amount/rate)*100)/100));
    System.out.println("aToD = " + toDisplay);
    return Double.toString(toDisplay);
  }

/*@ requires true;
  @ modifies CurCurrency ;
  @ ensures  CurCurrency >= 0 ;
  @ ensures  CurCurrency == 0
          || CurCurrency == 1
          || CurCurrency == 2 ;
  @ exsures (Exception e) false;  @*/
  public void setCurrency(String s) {
    if (s == null) {CurCurrency = 0; return;}
    if (s.compareToIgnoreCase("FRF")==0)
           CurCurrency = FRF;
    else if (s.compareToIgnoreCase("EUR")==0)
           CurCurrency = EUR;
    else {CurCurrency = 0;}
} }
```

Figure 3. Currency_src class excerpt

Let us illustrate the JML syntax on two parts of the banking application. Fig. 3 describes a part of the `Currency_src` class. Fig. 4 is a part of the JML specification of the `registerSpendingRule` method of the `SpendingRule` class.

In Fig. 3, for sake of brevity, we reduced the original code to one dealing with only two currencies: French Franc (FRF) and Euro (EUR). They are coded by a numeric code (1 or 2). The 0 value is reserved to code undefined currency. The changing rate between Franc and Euro is fixed (1 EUR = 6.55957 FRF). The `amountToDisplay` method converts a French Franc amount into a currency selected by the user with the `setCurrency` method.

The JML specification appears as specialized Java comments: between /*@ and @*/ or starting with //@. The specification of a method precedes its interface declaration, following the usual convention of Java tools such as JavaDoc.

JML annotations adopt a "design by contract" style of specifications, which relies on three types of assertions: class invariants, preconditions and postconditions.

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 [10].

The invariant stated in Fig. 3 at lines 19-20, indicates that the CurCurrency value should always be between 0 and 2. The two very simple invariants at lines 13 and 17 express that EUR_RATE and FRF_RATE will always stand for 6.55957 and 1 respectively[1].

Preconditions in the *requires* clause say which assertions must hold before this method can be called. If that is not true, then the method is under no obligation to fulfill the rest of the specified behavior.

In our example, most preconditions are set to true (see Fig. 3, lines 22 and 38). 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. It is a typical example of *defensive programming* style.

Postconditions are expressed in the *ensures* clauses. They express the results and the properties expected to hold just after the method execution. For instance, the postcondition of `setCurrency` (Fig. 3, lines 41 to 43) expresses that the `CurCurrency` attribute value should be equal to 0, 1 or 2. This postcondition is another expression of the invariant given on lines 19-20. The case study includes a large variety of postconditions, ranging from `true` expressions (cf. the postcondition of the `amountToDisplay` method Fig. 3, line 23), to more complex ones, that contain some state variables, may refer to the relation between the final and the initial states (denoted by `\old`), and involve quantified expressions over sets or vectors (see Fig. 4).

[1]Since these variables were defined as "final", these invariants were typically designed for the proof phase.

```
1   /*@ requires true ;
      @ ensures (threshold > 0 && period >= 0
       && account != spending_account
       && account >= 0 && spending_account >= 0
5      && (\exists int i; i >= 0 &&
          i< accman.LocalVector.size() &&
          ((Account)(accman.LocalVector.
                     elementAt(i))).accountnum
                            == account)
10    && (\exists int i; i >= 0 &&
          i< accman.LocalVector.size() &&
          ((Account)(accman.LocalVector.
                     elementAt(i))).accountnum
                         == spending_account))
15    ==>(\result == 0 &&
         (rules.size()==(\old(rules.size())+1)));
      @ ensures ...
      @ exsures (Exception e) false;  @*/
      public int registerSpendingRule(String date,
20         int account, float threshold,
           int spending_account, int period)
       { ... }
```

Figure 4. A complex JML annotation

The exsures clauses are a special kind of postcondition for exception specification. Fig. 3, line 23, the *exsures* clause specifies that the `amountToDisplay` method should never raise an exception.

JML extends the Java syntax with several keywords.

- `\result` (Fig. 4, line 15): Its value is the value returned by the method. It can only be used in *ensures* clauses of a non-void method.
- `\old` (Fig. 4, line 16): An expression of the form \old(Expr) refers to the value that the expression Expr had in the initial state of a method.
- `\forall` and `\exists` (Fig. 4, lines 5 and 10): They are universal and existential quantifiers .

In the banking example, 362 of the 615 lines of JML assertions are distributed as shown in Table 2. Postconditions (the *ensures* clause) represent most of the JML assertions, especially in classes `AccountMan_src` and `Balance_src` where they are dedicated to the specification of error codes. The remaining 253 lines of JML correspond to loop invariants, to additional keywords such as the *modifies* clauses, or to comments.

3.2 JML associated tools

The JML release consists of several tools to check the syntax and typing of specifications [2]. It also includes the

Classes	number of methods	nb. of lines of invariant	requires	ensures	exsures
Transfer_src	7	5	6	108	6
AccountMan-src	8	17	8	9	7
Currency_src	7	7	7	6	7
Balance_src	3	1	2	37	2
Spending_rule	2	20	13	6	1
Saving_rule	2	20	13	4	1
Rule	5	3	6	6	2
Account	7	5	8	9	7
Total	41	81	63	185	33

Table 2. JML assertion distribution in the Banking example

`jmlc` tool, that uses the JML annotations to add runtime assertions to the compiled java code [5]. JML assertions are thus evaluated at execution time against the behaviour of the program execution. If the assertion is not verified, an exception is generated which reports the kind of assertion which failed (invariant, pre- or postcondition). In most cases[2], such an assertion violation reveals a difference between the behaviour specified in JML assertions and the code execution. The fault can be either in the specification or in the program.

The code generated by `jmlc` can be used in combination with JUnit [11] in a testing process. The JML-JUnit tool [6] is a combinatorial testing tool which generates simple test cases consisting of a single call to the methods of the object under test. The tool generates combinations of selected values of the method parameters to result in a large set of test cases. The tool then exploits JUnit to run the tests and `jmlc` to provide an executable oracle.

Several tools are available for formal proofs of Java programs specified in JML. The LOOP tool [18] and the JIVE environment [15] convert JML into PVS models. The Krakatoa [15] tool translates JML into an internal language from which proof obligation are expressed into Coq. Jack is built on B [3].

Finally, the ESC/Java tool [8] is a lightweight tool that aims at identifying (and correcting) errors early in the development (static validation). It does not aim to provide a formal proof of the code. For example, ESC/Java is efficient to warn about potential null pointer usage.

[2]When a precondition evaluates to false due to a wrong choice of parameters by the tester, it does not reveal an error and leads to an inconclusive test execution.

3.3 JML vs Java assertions

The assertion mechanism is a new feature of version 1.4 of the Java Programming Language. An assertion is a boolean condition that can be evaluated at run-time. Options to the java compiler allow to turn the evaluation of assertions on and off. Java assertions are a simpler mechanism than JML:

- Java assertions are pure Java expressions and do not benefit from the additional constructs of JML (e.g. `\old`, `\result`, `\forall` and `\exists`).
- While JML features various kinds of assertions (invariants, pre- and postconditions), Java assertions are of a single kind. With JML, an invariant is written once and executed after each method invocation. To obtain a similar result with Java assertions, the invariant must be copied at all places where it must be checked.
- The only tool supporting Java assertions is the Java compiler, while JML is associated to several proof and testing tools.

4 The validation experiment

The validation work aimed at evaluating how the existing JML annotations, produced during a proof process, could be exploited in a testing process. Gemplus provided the informal requirements (in French), the JML annotations and the source code. A test plan was also produced by Gemplus, but not used in the experiments reported in this paper.

The main purpose of the LSR work was to find some errors in the code. To be more precise, we tried to find some inconsistencies between the code, the JML assertions and the informal requirements. Three cases can be identified.

- The JML assertions and the code are not consistent. It is detected during the test, when a JML assertion is violated or when an unexpected Java exception is raised.
- The JML assertions are inconsistent with the informal requirements but consistent with the Java code. Such an error can only be detected thanks to a human analysis of the JML assertions or of the test executions.
- The JML assertions, the code and the informal requirements are consistent with one another, but the observed behavior reveals that some common sense requirements have been overlooked.

To do this work, the LSR researchers were divided into two teams. Both teams worked separately in a bounded time period (3 days). The first team made a critical code review and then used random testing. The second team applied a combinatorial testing approach based on requirements. A human critical analysis of the execution results was performed in parallel with the JML automatic decision. This helped us to find cases where the code and the JML specification were consistent, but different from the requirements.

For both teams, the testing work consisted in producing some test data sequences and executing them.

4.1 Code review and random testing

Approach The first team used a random testing approach [7]. For that purpose, we have built a random test generator for Java programs specified in JML.

Before the test generator was used, a code review was carried out. A code review consists in reading the source code (here the application code and the JML specification), in order to find errors. When some errors were discovered, test cases were manually written to illustrate them. Here, the code review also helped to identify suspicious portions of code where it made sense to target a more extensive testing effort.

Random testing tool principles Jartege (Java Random Test Generator) is a tool, developed in the LSR, which enables random dynamic generation of unit tests for Java classes specified in JML.

Jartege interacts with the program under test and chooses randomly one of the methods whose precondition is true. The method is then executed by the system under test, and Jartege proceeds iteratively to build a large sequence of calls. The test sequences generated by Jartege can be saved and replayed later. The random aspect of the tool can be parameterized in several ways, in order to control and target the testing effort:

1. Some *weights* are associated with classes and operations by the user. Classes and operations are chosen by Jartege according to these weights. In particular, it is possible to forbid to call some operation by associating a null weight with it.
2. It is possible to control the number of instances of a class with creation probability functions. This feature permits either to create a few instances of a class and make numerous method calls to these instances, or to create many instances of this class.
3. Jartege provides the possibility to define generators for some primitive parameter of a given method. This is particularly useful for operations which have a strong precondition.
4. Jartege allows one to write test fixtures, in a similar way to JUnit, with setUp and tearDown methods.

Jartege is written in Java. It uses Java introspection to discover the available operations of each class under test. Each generated call is executed in order to eliminate calls which violate an entry precondition. Some of Jartege classes have been specified in JML, and tested with itself.

Results of the team The code review phase took one person-day. It allowed to detect four errors. Those were corrected before the random test phase. In its turn, the test phase allowed to reveal five new errors or suspicious situations in one day.

4.2 Requirements based combinatorial testing

Approach First, some general properties from the requirements were identified. For example, the transfers must be done if the money amount is correctly set, and if the accounts exist and are different. Moreover, a transfer operation between two accounts must modify them as expressed and must not modify other accounts (no side-effect).

Then, the test cases were derived from the properties. To do that, some "abstract scenarios" were first expressed to define sets of similar test cases. Thus, to test the previous informal property, one should try to transfer some money between (non)existing accounts, with (in)valid values. Every time, the set of existing accounts should be consulted to detect possible side-effects.

Combinatorial tool principles To express abstract scenarios, the Tobias tool was used. Tobias is a LSR tool designed for combinatorial testing. It is used to instantiate the abstract scenarios into executable test cases for JUnit. For example, S1 (Fig. 5) performs a money transfer from *b1* to *b2* with amount *c*. The transfer is followed by a balance check of account *a*. The tool will expand this abstract scenario, producing all possible combinations of parameters *b1, b2, c* and *a*. S1 will produce 320 test cases.

Fig. 5 gives another scenario S2. It was designed to test the currency converter. The S2 expression defines three tests. They first set the currency to respectively FRF, EUR or CHF and then they display the value of 1 FRF into the chosen currency.

$$\begin{gathered}\text{S1 = transfers.M1 ; balance.M2}\\ \text{S2 = currency.M3 ; currency.M4}\\ \textit{with}\end{gathered}$$

$$\begin{cases} M1 = \{\mathit{transfer}(b1, b2, c) \mid b1, b2 \in \{1, 2, 3, 100\}, \\ \qquad c \in \{100.0, 99.9, 1000, 0, -2.1\} \\ M2 = \{\mathit{getAccountsVector}(a); \mathit{getBalances}(a) \mid \\ \qquad a \in \{0, 1, 2, 3\}\}\ \} \\ M3 = \{\mathit{setCurrency}(f) \mid f \in \{'\mathrm{FRF}', '\mathrm{EUR}', '\mathrm{CHF}'\}\} \\ M4 = \{\mathit{amountToDisplay}(1.0)\} \end{cases}$$

Figure 5. Two abstract scenarios for Tobias

Results of the team The team produced 17 abstract scenarios (which are organized into 7 properties), which were instantiated into 1241 test cases. Those represented 40 000 Java code lines (for JUnit). It took 6 person-days to analyze the specification, produce the abstract scenarios, execute the tests and analyze the traces. 16 errors or suspicious situations were discovered.

4.3 Found errors or suspicious situations

At the end of both processes, 18 different errors or suspicious situations were uncovered. We say that there is an error when the JML assertion checker raises an exception. Java exceptions also often reveal errors in the code[3]. We call suspicious situations the cases where formal specification and code have the same behavior, but do not correspond to the informal requirements or to common sense.

The following table lists all errors, with their types and the way they were discovered. Errors 3, 10, 13 and 14 were fixed between code review and random testing, in order to facilitate the random testing process.

	team 1		*team 2*		
Err.	Code review	Random testing	With Tobias	Type of error	Method of detection
1			X	limit	human oracle
2			X	limit	human oracle
3	X	▨	X	floating-point	code rev + JML or.
4		X	X	floating-point	JML oracle
5		X	X	floating-point	JML oracle
6		X	X	floating-point	JML oracle
7			X	postcondition	JML oracle
8			X	postcondition	JML oracle
9		X	X	design	JML oracle
10	X	▨		design	code review
11			X	limit	human oracle
12			X	limit	human oracle
13	X	▨	X	design	code rev + Java ex.
14	X	▨		postcondition	code review
15		X	X	several*	Java exception
16			X	counter-intuitive	human oracle
17			X	counter-intuitive	human oracle
18			X	floating-point	human oracle

*precondition mistake, under specification, or design mistake

The 18 errors can be classified as follows.

- **Floating-point approximations**. There are 5 cases related to the floating-point approximations (errors 3, 4, 5, 6, 18). The floating point type is used to represent the account balance. The errors occur when the postcondition and the code compute the same "value" in different ways. For instance, in `SavingRule_src` class, the postcondition of the `check()` method stands for
    savingRef.balanceamount ==
        \old(savingRef.balanceamount) +
        \old(accountRef.balanceamount) - threshold
  and the code computation is
    savingRef.balanceamount =
        savingRef.balanceamount +
        (accountRef.balanceamount - threshold)
  The result is different because $(x + y) - z$ is not equal to $x + (y - z)$ with x, y, z being float numbers. With float, + and - operations are not commutative due to their limited precision[4].
- **Limit**. There are 4 cases that are dealing with "limits" (err. 1, 2, 11 and 12). Let us see two examples.
  A transfer rule can be registered with a time period of 0, which is forbidden in the informal requirements, but not in the JML specification.
  One informal requirement says that there is no limit amount for a credit. So testers tried to credit one account with the Java pre-defined constant POSITIVE_INFINITY. The fact that this operation is accepted was considered as a suspicious situation.
- **Wrong postcondition**. 3 cases are in the postconditions, typically several \old arguments were forgotten (err. 7, 8, and 14). For instance, error 14 is due to an assertion indicating that the new value of an attribute is equal to itself ($a == a$). The correct assertion is ($a == \backslash old(a)$), saying that the value of the attribute has not been changed[5]. This specification error is a typical example of error that can not be discovered with a black-box testing approach, since the assertion is always true.
- **Design mistake**. Errors 9, 10, and 13 have been classified as design mistakes. One critical attribute is public instead of private (err. 10). It is possible to assign the same identifier to two different accounts if two account managers are created (err. 9). The banking application deals with threads, but there is no critical section to access an account (err 13).
- **Counter-intuitive behavior**. Errors 16 and 17 denote counter-intuitive behaviors. It is possible to delete an account on which there are some active saving or spending transfer rules. This is neither specified informally nor formally. So, it is not possible to conclude whether the application behavior is correct or not. Intuitively, one can imagine that the removal of an account, which is a transfer destination may create some access conflict if the rule is not deactivated before.
- **Several classifications**. Error 15 falls into several categories. The method `inputToAmount` of the `Currency_src` class needs a parameter to be a string representing a float. Some test cases called this method with a wrong parameter and resulted in Java run time exception. This is not indicated in the informal requirements and not expressed in the JML assertions. This error can therefore be considered as a precondition inadequacy (the existing JML precondition does not indicate the parameter form), under-specification

[3]Then reveal a missing *exsures* clause.

[4]During the proof process, the approximation problem was not tackled.

[5]This properties could also have been expressed with the JML keyword *not_modify*.

(the existing informal specification does not indicate the parameter form), or design mistake (the parameter could have been typed as float).

4.4 Code coverage

We still did not know whether all the errors of the banking application were found. Testing is always a process which is difficult to end. To have some kind of feedback about the quality of the testing suites, we have evaluated the code coverage of the application.

One should note that during the test phase, the LSR had no coverage tool for Java program testing. It is several months after the end of the testing experience, that we have installed JCoverage tool [9], and used it with the existing random and combinatorial test suites. Here are the results.

Code statement coverage		
Class	random test suite	combinatorial test suite
Account	87%	87%
AccountMan_src	86%	91%
Balances_src	77%	71%
Currency_src	98%	88%
Rule	96%	96%
SavingRule	100%	100%
SpendingRule	100%	100%
Transfers_src	86%	90%

This coverage analysis reveals that the test suites do not cover 100% of the instructions. With the combinatorial approach, some cases were clearly forgotten by the testers. For instance, the testers forgot to check transfer rules with negative values.

But one could also notice that there is

- some dead code: several methods are defined as "private" (and hence do not appear in the public interface of their class) but are never called inside the class (e.g. method `getCurCurrency()` in `Currency_src`),
- one public method of class `Transfers_src` which is not declared in the informal specification (same name than the declared one, but with different parameters). Since Tobias scenarios were built from the informal specification, our test suite did not include calls to this unspecified function.

4.5 Testability of the application

Testability is an evaluation of how effectively the software can be tested. Intuitively, the higher the testability is, the easier (or the cheaper) will be the testing phase. All aspects of a program may make it more difficult to test. This can range from incomplete or ambiguous specification to complex code. During this experiment, we have noticed testability problems at four levels.

Informal requirements Some informal requirements were not possible to validate. For example, the transfer rules are applied with a periodicity defined by the user. It was possible to check that the transfer rules were *periodically* applied, but it was not possible to check that the effective transfer operations were done at the time period indicated. This real-time property could not be expressed in JML.

Incomplete JML specification The JML specification stands for the oracle. If the JML postconditions or invariants are under-specified, it is difficult for tests to detect automatically some errors.

For instance, when a saving rule with wrong parameters is registered, an error code should be returned. These are specified in the informal requirements, but they are not described in JML (although it could be expressed). Thus, one can not test whether an error with a wrong error code is produced. The error codes are specified with JML in all classes but `Transfers_src`.

Similarly, the `amountToDisplay` method of `Currency_src` (Fig. 3) has no postcondition. Thus, during this experiment, we could not possible to check *automatically* that the displayed amount is correct (a human oracle or an additional JML assertion is needed).

Influence of the code on JML In a classical waterfall model, the specification is expected to be written before coding. In this experiment, since the code was taken from an existing application, JML assertions have been added after the coding phase. As a result, some postconditions may have been influenced by the code. Actually, it is tempting to simply copy-paste the code of the method in the JML assertion and then to replace "=" with "==" and add some "\old" keywords. Unfortunately, this often results in copying coding errors into the specification and prevents the detection of errors by a proof or testing process. Therefore, care should be taken to express postconditions in a different, and often more abstract, way. This should also result in specifications that are more robust to evolutions.

Code Some functionalities could have eased the banking application testing. For instance, it is possible to register transfer rules, but it is not easy to suspend or to delete them. The notion of account exists, but there is no public method to get the account objects.

4.6 Reporting errors to Gemplus

The results of these testing experiments were reported to Gemplus. It turns out that the errors we discovered were either already known by Gemplus, or at least expected by them. For example, errors related to float numbers had been consciously left out of their proof process. Actually, we only missed one error, which was difficult to detect because it was not covered by the JML specification.

5 Conclusion

This paper reports on a testing experiment to validate a small Java application (500 LoC). The starting point of our work involved three documents: natural language requirements, JML specification and Java code. The Java code had already undergone some level of proof with respect to the JML specification.

Our validation activity was mainly based on testing, complemented with a preliminary code review process. Random testing and combinatorial testing were used to produce more than thousand test cases. These test cases took advantage of the executable character of JML and used the JML specification as an oracle. As a result, 18 errors or suspicious cases were detected, 7 of them being revealed by the JML mechanisms.

At the beginning of this paper, several questions were asked. We may now try to answer them.

5.1 Is JML well-suited for validation by test?

The use of JML in a testing process corresponds to a light-weight approach to formal methods. The choice of JML by Gemplus is motivated, amongst others, by its Java syntax. Experiments such as this one have shown that software engineers quickly learn to read and eventually to write JML specifications.

The experiment reported in this paper only addressed the use of JML for testing. Testing is a classical software development activity that is well integrated into development processes. Since JML is easily integrated with popular tools such as JUnit, we believe that a testing approach based on JML is compatible with a wide variety of development processes, ranging from classical waterfall life-cycles to extreme programming practices.

JML as a test oracle This experiment exploited the executable character of JML specifications to use it as an oracle for the tests. 1241 test cases were generated by the Tobias tool. These definitely took advantage of using the JML specification as a single centralized oracle. Using this single oracle prevented us from scattering it in the JUnit test cases, which requires to write a new piece of oracle for each new test and to make sure that these elementary oracles are mutually consistent. Moreover, the same JML oracle was used by both teams which used different testing tools (Jartege and Tobias/JUnit).

This single oracle approach may also bring benefits in the maintenance of tests. If the systems evolution includes some regression, it is often needed to rewrite large portions of the test suite. Using JML, specification changes are immediately available in the oracle.

Expressiveness of JML 7 out of 18 errors were detected using JML, but many other errors correspond to properties which could have been expressed formally using JML. The experiment has thus revealed the incompleteness of the available formal specification. The following table shows which kinds of properties were actually detected using JML and which ones could have been detected if the specification was more complete.

Error type	detected by JML assertions	detectable by JML assertions
limit	No	Yes
floating point	Yes	
postcondition	Yes	
design	No	1 of the 3 errors
counter-intuitive	No	Yes

We believe that JML has a good expressiveness to cover most of the requirements of this application: 80 to 90% of the errors could have been detected if adequate JML assertions had been available.

To be honest, the detection of floating point errors is due to the fact that JML specifications and code are slightly different. This is unsufficient to detect all floating points errors. A more adequate treatment of floating point operations requires to specify the expected precision of these operations.

5.2 Is JML, written for proof, reusable for test?

A specificity of this experiment was that the JML specification came out of a proof process led by our industrial partner. Several parts of the specification were only expressed to help the proof process. They are often too close to the Java code to help find errors. But although these elements of the specification do not contribute to the test oracle, they do not arm the testing process. These elements of specification can even be useful for regression testing, provided they are sufficiently abstract to express the functionalities and not how they are implemented.

The main negative influence of these specification statements is that they increase the size of the specification and tend to give some confidence that the application is sufficiently specified. In this perspective, automatic annotation tools [17] which simply propagate annotations or translate code into annotations, will also lower the ratio of annotations useful for the test in the overall specification.

Well-structured documentation of the JML assertions may contribute to solve this problem. It is important to trace where the annotations come from: are they the translation of requirements or were they added to document the code?

As a conclusion, since JML annotations are motivated by different concerns, it should be interesting to use structuring mechanisms that identify them according to their rationale and intended use.

5.3 Jartege and Tobias

The testing phase made use of classical or simple test techniques. From a methodological point of view, Tobias tests were designed using the category and partition approach, which is a well known and efficient method [16]. Jartege is based on random testing, which turned out to be a good way to find errors quickly and at low cost. Both methods appeared complementary. On the one hand, random testing is not very good at finding errors "at the limits", because the probability to reach such a boundary can be very low. Such errors can easily be targeted by Tobias test schemas. For example, in this experiment, several tests were targeted to the use of pre-defined Java values such as POSITIVE_INFINITY or NEGATIVE_INFINITY. On the other hand, random testing may help to find errors corresponding to unclassical scenarios, which are less likely to be anticipated in a test plan.

Tobias test schemas are one of the original points of the tool. Unlike JML-JUnit which only issues a single method call per test case, Tobias starts from a schema which corresponds to a sequence of method calls and generates the combinations of all parameters and all methods of the schema. This allows to use a combinatorial approach on the basis of an abstract test case (the schema) which corresponds to a behaviour targeted by the test engineer. The tool was able to find 16 out of the 18 errors. It was perceived as a productivity amplifier which effectively helped writing large sets of tests on the basis of test schemas adequately designed by the test engineers.

Test schemas capture the knowledge of the test engineer in a very compact and abstract form. For example, in another experiment, the Gemplus researchers succeeded in expressing a test plan (of 50 test sequences) in 5 test schemas, which were unfold into more than 2000 executable test cases. From an industrial point of view, the ability to express very abstract test cases (schemas) from which a large set of executable tests can be automatically generated, was considered to be *really* interesting. First, it is cheaper to write and to maintain a few abstract schemas than many executable tests. Second, the systematic unfolding of schemas by Tobias may produce some test sequences that were not originally imagined by the human tester. These advantages have to be confirmed with other experiments.

Nevertheless, a classical drawback of the combinatorial approach is combinatorial explosion. The Tobias tool and a special purpose test driver have been build which exploit the results of previously failed tests to avoid playing tests which will eventually fail [14].

As a conclusion, we believe that JML associated to simple automated tools provides an interesting framework for the validation of Java applications at reasonable cost. But it requires some discipline from the software engineers to clearly distinguish between the portions of JML that were added to support a proof process and those that express the actual abstract specification of the system under test.

References

[1] J.R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[2] L. Burdy, Y. Cheon, D. R. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An Overview of JML Tools and Applications. In *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS'03)*, volume 80 of *ENTCS*, pages 73–89. Elsevier, 2003.

[3] 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.

[4] L. Casset. Development of an Embedded Verifi er for Java Card Byte Code Using Formal Methods. In *International Symposium of Formal Methods Europe (FME'02)*, volume 2391 of *LNCS*. Springer, 2002.

[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, *International Conference on Software Engineering Research and Practice (SERP '02)*, Las Vegas, Nevada, June 2002. CSREA Press.

[6] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *16th European Conference on Object-Oriented Programming (ECOOP'02)*, number 2374 in LNCS. Springer, June 2002.

[7] J. Duran and S. Ntafos. An Evaluation of Random Testing. *IEEE Trans. on Software Engineering*, 10(4):438–444, 1984.

[8] C. Flanagan, K. R. M. Leino, Lillibridge M., G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. of the ACM SIGPLAN Conference on Programming language design and implementation*. ACM Press, 2002.

[9] JCoverage. http://jcoverage.com/.

[10] The Java Modeling Language (JML) Home Page. http://www.cs.iastate.edu/ leavens/JML.html.

[11] JUnit. http://www.junit.org.

[12] 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 Specifi cations of Businesses and Systems*. Kluwer, 1999.

[13] G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specifi cation language for Java. Technical Report 98-06q, Iowa State University, Dept of Computer Science, 2002.

[14] Y. Ledru, L. du Bousquet, O. Maury, and P. Bontron. Filtering TOBIAS combinatorial test suites. In *Fundamental Approaches to Software Engineering (FASE'04)*, volume 2984 of *LNCS*, Barcelona, Spain, 2004. Springer.

[15] J. Meyer and A. Poetzsch-Heffter. An Architecture for Interactive Program Provers. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1785 of *LNCS*. Springer, 2000.

[16] S. Ntafos. On random and partition testing. In *Proceedings of ACM SIGSOFT international symposium on Software testing and analysis*, pages 42–48. ACM Press, 1998.

[17] M. Pavlova. Automatic code annotation using JML. Thèse, Université de Jussieu, France, Septembre 2003.

[18] J. van den Berg and B. Jacobs. The LOOP Compiler for Java and JML. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *LNCS*. Springer, 2001.