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.

L. du Bousquet, Y. Ledru, O. Maury, C. Oriat, J.-L. Lanet A case study in JML-based software validation. In *(short paper) Proceedings of 19th Int. IEEE Conf. on Automated Sofware Engineering (ASE'04)*, pp. 294-297, IEEE CS Press, Linz, Sep. 2004

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.

1. Context and motivations

The automation of validation activities (test and proof) often requires a model or specification of the system under validation. Since the development of a model may be an expensive task, 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 attempt to *reuse* a JML specification, developed for a proof activity, during a testing phase.

JML (Java Modeling Language) is a behavioral interface specification language that can be used to specify Java modules [3, 5]. JML annotations adopt a "design by contract" style of specifications, which relies on three types of assertions: class invariants, preconditions and postconditions. Several kinds of validation tools [1] take advantage of JML specifications: testing, static analyser and proof tools. From a practitioner's point of view, the syntax of JML, based on Java, makes it easier to read and to write specifications.

In the past years, Gemplus has led several research projects related to the formal development of smart card applications. Recently, Gemplus has adopted JML in research projects dedicated to the proof of Java Card applications [2]. Our case study started from a simplified banking application, developed in Java. The application was partially validated by Gemplus during a proof process which added JML annotations. Several errors were found and corrected. A second validation step, based on test, was carried out by the LSR, 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[1].This paper reports on this second validation step, organised to experiment two of our testing tools, and to answer the following questions:

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

2. Case study

The case study is a simplified version of a real banking application which deals with money transfers. It was initially developed as a case study to validate the Jack prover [2]. The application allows to create accounts, to consult them and to make money transfers from one account to another. "Transfer rules" (either saving or spending rules) can be defined to schedule periodical transfers. The application also allows to convert money from one currency to another.

The banking application code is composed of eight classes: 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, a *currency converter*, and three classes dedicated to of the transfer rule implementation.

This application has a total of 518 lines of Java, annotated with 615 lines of JML (table 1). 362 of the 615 lines of JML correspond to invariants, pre- and postconditions. 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

1 At the time of this experiment, the Jack proof tool was still under development, and some aspects of the specification, such as float numbers, were not yet covered.

Classes	Java lines	Java Doc lines	number of methods	nb. of lines of JML				
				invariant	requires	ensures	exsures	total
Transfer_src	116	34	7	5	6	108	6	150
AccountMan−src	105	51	8	17	8	9	7	236
Currency_src	93	20	7	7	7	6	7	28
Balance_src	64	38	3	1	2	37	2	58
Spending_rule	40	33	2	20	13	6	1	42
Saving_rule	40	33	2	20	13	4	1	42
Rule	22	23	5	3	6	6	2	23
Account	30	20	7	5	8	9	7	36
Total	518	251	41	81	63	185	33	615

Table 1. Metrics in the Banking example

as the *modifies* clauses, or to comments. The fact that the number of JML annotation lines is larger than the Java code length is mainly due to the proof process.

3. Validation step

The main purpose of the 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 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.

Two LSR teams worked separately during 3 days, to produce some test data sequences and execute them.

3.1. Code review and random testing

One team first carried out a code review and then used a random testing approach, using the Jartege tool. Jartege (Java Random Test Generator) [7] was developed by the LSR. It enables random dynamic generation of unit tests for Java classes specified in JML.

Random testing tool principles 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 allows to define generators for some primitive parameter of a given method. This is particularly useful for operations with a strong precondition.
4. Jartege allows one to write test fixtures, in a similar way to JUnit [4], 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 Jartege classes have been specified in JML, and tested with itself.

Results of the team The code review phase took one person-day. and detected 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.

3.2. Requirements based combinatorial testing

Approach The second team used a 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 [6]. 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 (see below) 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.

Scenario S2 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.

S1 = transfers.M1 ; balance.M2
S2 = currency.M3 ; currency.M4

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

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.

Test schemas capture the knowledge of the test engineer in a very compact and abstract form. From Gemplus viewpoint, 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. It is cheaper to write and to maintain; and the systematic unfolding of schemas may produce some test sequences that were not originally imagined.

Results of the team The team produced 17 abstract scenarios (organized into 7 properties), which were instantiated into 1241 test cases (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.

3.3. Found errors or suspicious situations

At the end of both processes, 18 different errors or suspicious situations were uncovered[2]. We say that there is an error when the JML assertion checker raises an exception. 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 18 errors can be classified as follows.

Floating-point approximations. 5 cases are related to the floating-point approximations (err. 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: $(x+y)-z$ or $x+(y-z)$. The result is different because with float, + and - operations are not commutative due to their limited precision[3].

2 Errors 3, 10, 13 and 14 were fi xed after the code review phase, 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

Table 2. Errors detected

Limit. 4 cases are dealing with “limits” (err. 1, 2, 11 and 12). For example, 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. Also, 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 judged as a suspicious situation.

Wrong postcondition. 3 cases are in the postconditions, typically several \old arguments were forgotten. For instance, err. 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. This 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. Three errors 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).

3 During the proof process, the approximation problem was not tackled.

Counter-intuitive behavior. Two suspicious situations 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. 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. A method needs a parameter to be a string representing a float. This is neither expressed in the informal requirements nor in JML assertions. This error can thus be considered as a precondition inadequacy (the existing JML precondition does not indicate the parameter form), under-specification (the informal specification does not indicate the parameter form), or design mistake (the parameter could have been typed as float).

4. Conclusion

This paper has reported on a testing case study where a JML specification, expressed during a partial proof process was reused as a test oracle. 18 errors or suspicious situations were detected, 7 of them being revealed by the JML mechanisms. 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.

Is JML well-suited for validation by test? This case study exploited the executable character of JML specifications to use these as a test oracle. 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 check that these elementary oracles are mutually consistent. Moreover, the same JML oracle was used by both teams with different testing tools (Jartege and Tobias/JUnit). This single oracle approach may also bring benefits in the maintenance of tests. If system 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.

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 case study has thus revealed the incompleteness of the available formal specification. 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.

Is JML, written for proof, reusable for test? A specificity of this study 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 no prejudice to the testing process. They 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 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, 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 application under test.

Acknowledgments This work was partially supported by the COTE project (French RNTL). We would also like to thank Farid Ouabdesselam and Jean-Luc Richier for comments on an early version of this work.

References

[1] 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 *FMICS'03*, volume 80 of *ENTCS*, pages 73–89. Elsevier, 2003.

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

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

[5] G. Leavens, A. 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.

[6] 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, 2004. Springer.

[7] C. Oriat. Jartege, a tool for random generation of unit tests for Java classes. Technical Report RR1069, 2004.