J Autom Reasoning (2010) 45:415–435
DOI 10.1007/s10817-009-9132-y

Reusing a JML Specification Dedicated to Verification for Testing, and Vice-Versa: Case Studies

Lydie du Bousquet · Yves Ledru · Olivier Maury · Catherine Oriat · Jean-Louis Lanet

Received: 5 October 2008 / Accepted: 6 May 2009 / Published online: 2 June 2009

Abstract Testing and verification are two activities which have the same objective: to ensure software dependability. In the Java context, the Java Modelling Language (JML) has been proposed as specification language. It can be used both for verification and test. Usually, the JML specification is designed with a specific purpose: test or verification. This article addresses the question of reusability of a JML specification provided for one activity (resp. verification or test) in the other context (resp. test or verification). Two different case studies are considered.

Keywords Software dependability · Java · JML · Verification · Testing

1 Introduction

Testing and verification are two classical approaches used in order to achieve software dependability. As it has been noticed in the *Test And Proof* conference series (TAP 2007, 2008, and 2009), verification and testing have been pursued by distinct communities using rather different techniques and tools. During the last decade, an increasing number of efforts were carried out to combine both approaches.

In this article, we focus on the validation and verification of Java programs. The Java Modelling Language (JML) has been proposed as specification language for those applications. It can be used both for verification and test activities, and a large amount of tools have been proposed to support both of them.

L. du Bousquet (✉) · Y. Ledru · O. Maury · C. Oriat
Laboratoire d'Informatique de Grenoble (LIG Labs),
Universités de Grenoble, BP 72, 38402 Saint Martin d'Hères cedex, France
e-mail: lydie.du-bousquet@imag.fr
URL: http://www.liglab.fr/

J.-L. Lanet
Laboratoire XLIM, Université de Limoges, 123,
avenue Albert Thomas, 87060 LIMOGES CEDEX, France

One would think that using the same language for test and verification would ease the combination of both activities. In reality, it still seems not to be so common. Indeed testing and verification have two different objectives, and a (JML) specification designed with a specific purpose (test or verification), may not be appropriate for the other activity. Verification aims at assessing the correctness of some properties for all inputs in all "situations". On the other hand, testing is an incomplete validation approach. It aims at detecting faults but it cannot guarantee that all faults are discovered. Since verification is difficult, it requires a good knowledge of the application code, and is limited by the power of the tools (for instance, it is very difficult to verify some properties on floats). Hence, properties written for verification may specify a subset of the application, and this is not appropriate to detect faults on the unspecified parts of the application. Since testing is easier to carry out (it requires less knowledge of the code and is less restricted by the power of tools), testers may pay less attention to the way the specification is written, making it less usable for verification.

This article addresses the question of reusability of a JML specification provided for one activity (resp. verification or test) in the other context (resp. test or verification). Two different case studies are considered. The first one is a Banking application, which has first been verified then tested. The second one is a Home Network Services application, which has been tested and then verified. In both cases, evolution of the JML specification had to be undertaken in order to ease the second step of the validation (resp. test or verification).

In the following, Section 2 presents JML. Section 3 details the work done to test the Banking application after the verification. Section 4 describes the work done to verify the Home Network Services application after it was tested. Section 5 concludes with the lessons learnt during both experiments.

2 JML: Language and Tools

This section briefly presents JML and some associated tools.

2.1 The JML Language

Java Modelling Language (JML) is designed to specify Java programs by expressing formal properties and requirements on classes and their methods. The Java syntax of JML makes it easier for Java programmers to read and write specifications. The language is based on Java, with some additional keywords and logical constructions. Examples of JML assertions are given in Figs. 2 and 6. For more details, see [14, 16].

The JML specification appears as special purpose 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 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 [16].

- *Preconditions* in the *requires* clause give the assertions that must hold before this method can be called. If that is not true, then the method is under no obligation to fulfil the rest of the specified behaviour.
- *Postconditions* are expressed in the *ensures* clauses. They express the results and the properties expected to hold just after the method execution. The *exsures* clause is a special kind of postcondition (signal clause) for exception specification.

JML extends the Java syntax with several keywords.

- `\result` is the value returned by the method. It can only be used in *ensures* clauses of a non-void method.
- `\old`. 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` are universal and existential quantifiers.

JML can also be used to define *annotation statements* that may be interspersed with Java statements in the body of a method. For instance, a loop statement can be annotated with *loop invariants* or *variant functions*, that are written above the loop itself. Both are used to help verification of the partial correctness and the termination of a loop statement. Moreover, various assertions can be used to specify abstract data types. For example, the *initially* clause allows one to define properties that must be established by constructor methods of a class.

JML is more expressive than the Java assertion mechanism. The assertion mechanism was introduced in version 1.4 of the Java language. An assertion is a boolean condition that can be evaluated at run-time. In Java, options to the compiler allow turning 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 checked 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 with several verification and testing tools.

2.2 JML for Testing: Principles and Tools

The JML release consists of several tools to check the syntax and typing of specifications [5]. It also includes the `jmlrac` tool (JML runtime assertion checker), which uses the JML annotations to add runtime assertions to the compiled Java code [8].

The assertions are executed before, during and after the execution of a given method or constructor call. When a method (or constructor) is executed, three cases may happen.

All checks succeed: the behaviour of the method conforms to 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 behaviour of the method 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 test will not bring useful information because it is performed outside of the specified behaviour. This test delivers an *Inconclusive* verdict. For example, $\sqrt{x}$ has a precondition that requires x being positive. Therefore, a test of a square root method with a negative value leads to an *Inconclusive* verdict. But, if the square root method is called with a negative value inside a method under test, then a *Fail* verdict is delivered.

The code generated by `jmlc` can be used in combination with JUnit [15] in a testing process. The JML-JUnittool [9] is a combinatorial testing tool which generates simple test cases consisting of a call to one of the constructors of the given class, followed by a single call to one of the methods of the object under test. The tool generates combinations of selected values of the constructor and 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.

For the testing experiment in Section 3 of this article, we have mainly used two tools: Jartege and Tobias. Both use JML as test oracle.

Jartege allows random generation of unit tests for Java classes specified in JML [24]. As in the JML-JUnit tool, JML assertions are used as a test oracle. Jartege randomly generates test cases, which consist of a sequence of constructor and method calls for the classes under test. The random aspect of the tool can be parameterized by associating weights to classes and operations, and by controlling the number of instances which are created.

Tobias is a tool for combinatorial testing. Unlike JML-JUnit that generates test cases which consist of a *single* call to a class constructor, followed by a *single* call to one of the methods (see [8]), Tobias supports combination of calls to the methods [18, 21]. Tobias is available as an Eclipse plug-in [17].

2.3 JML and Verification

Several tools are available for formal verification of Java programs specified in JML [4, 5]. The ESC/Java tool aims at identifying (and correcting) errors early in the development (static validation) [7, 13]. It does not aim to provide a formal verification of the code. JACK [2, 6], Why/Krakatoa [12, 20] and KeY [1, 3] are three available environments for verification of Java programs specified in JML.

2.3.1 ESC/Java

ESC/Java[1] is an *Extended Static Checking* tool. The verification is *static* since the code is verified without being executed in a Java Virtual Machine. It is *extended* since the tool detects more errors than can be detected with traditional static analysis.

ESC/Java uses the Simplify prover to reason about the program semantics. It raises warnings in case of classical runtime errors, such as null dereferences, array bound errors, type cast errors, etc. It also warns about synchronization errors

[1] ESC/Java can be downloaded at http://kind.ucd.ie/products/opensource/ESCJava2/download.html

in concurrent programs (race conditions and deadlocks). Finally, ESC/Java issues warnings if the source code violates the JML assertions.

2.3.2 JACK

JACK[2] (Java Applet Correctness Kit) is a tool for the validation of Java applications annotated in JML [2]. JACK implements a weakest precondition calculus to automatically generate proof obligations for each path of the control flow. They can both be discharged to automatic and interactive theorem provers such as Coq or Simplify. Proof obligations are first expressed in an intermediate representation, and are then translated into the adequate language for the chosen prover. The tool is integrated into Eclipse.

2.3.3 Why/Krakatoa

"Why"[3] is a generic platform for deductive program verification [12]. The core of the platform ("Why" tool) produces verification conditions and sends them to existing provers. Several provers can be used: proof assistants such as Coq, PVS, Isabelle/HOL, HOL 4, HOL Light, Mizar and decision procedures such as Simplify, Alt-Ergo, Yices, Z3, CVC3, etc. Krakatoa is dedicated to the translation of Java programs annotated in JML, into the input language of "Why" (similar to ML) [20].

2.3.4 KeY

The KeY[4] System is a formal software development tool that aims to integrate design, implementation, formal specification, and formal verification of object-oriented software as seamlessly as possible. At the core of the system is a theorem prover for a first-order Dynamic Logic for Java. The tool has an easy-to-use graphical interface and seamlessly integrates automated and interactive verification [1, 3]. KeY also uses Simplify.

3 From Verification to Testing

This section focuses on the work done to reuse a specification dedicated to verification for testing activities. Section 3.1 deals with the presentation of the case study, a banking application. Section 3.2 briefly describes the JML specification produced for verification. Section 3.3 details the work done for test, and especially focuses on the errors found. Section 3.4 proposes a partial conclusion.

3.1 The Banking Application Case Study

The banking application case study, proposed by Gemplus, deals with money transfers [10]. The application administrator (the bank officer) can create accounts. The

[2] The current version can be downloaded at http://www-sop.inria.fr/everest/soft/Jack/jack.html under Cecill C licence.

[3] "Why" can be downloaded at http://why.lri.fr/

[4] KeY can be downloaded at http://www.key-project.org/download/

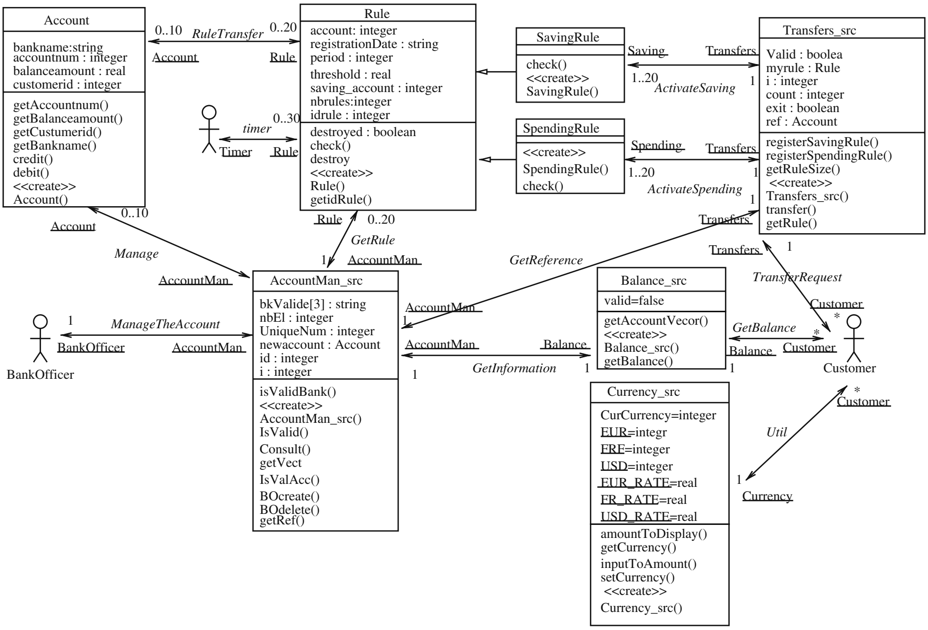


Fig. 1 Class diagram of the banking application

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 an application already used in the real world. 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: the `Account` class, the `AccountMan_src` class to create and delete accounts, the `Transfers_src` class to define spending and saving rules to transfer money from an account to another according to different thresholds, the `Balance_src` class that allows the customer to have access to his accounts, and the `Currency_src` class to convert currencies. The three remaining classes are dedicated to the definition of the transfer rule principles.

3.2 The Banking Application JML Specification for Verification

For this application, the JML annotations were originally designed to support a verification process. Both the Java application and the JML code were written by engineers at Gemplus. The application was first informally specified and modelled in UML. The Java code was then produced. Finally, the JML assertions were added to

Fig. 2 JML annotation for the method register spending rule

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

the code. The JML specification was originally designed to evaluate Jack, Gemplus's prover for JML [2, 6].

The application was not totally verified at this stage: some proof obligations were not satisfied and some parts of the code were not annotated. The JML assertions, the informal requirements and the code were then directly used for the testing phase. An example of those annotations is given Fig. 2.

This was the first use of JML for verification purposes by the Gemplus Research team. 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 verification process: annotation statements were inserted to guide the prover.

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 `Balance_src` and `AccountMan_src` where they

Table 1 Some metrics of the banking application

Classes	Java lines	JML lines
Transfer_src	116	150
AccountMan_src	105	236
Currency_src	93	28
Balance_src	64	58
Spending_rule	40	42
Saving_rule	40	42
Rule	40	23
Account	30	36
Total	518	615

Table 2 JML assertion distribution in the banking example

Classes	Number of	Number of lines of			
	methods	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

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.

3.3 Testing the Banking Application

Two testing campaigns were performed by two different teams in the Laboratoire d'Informatique de Grenoble (LIG). The code and annotations were re-used without modification by the LIG testing team. During this work, we tried to answer the following questions:

1. Can the banking application JML specification dedicated to verification be used for testing?
2. Is the JML specification detailed enough to allow accurate validation by test?

Both teams worked separately during a bounded time period (3 days). For both teams, the testing work consisted in producing some test data sequences and executing them.

The first team made a critical code review and then used random testing (with Jartege). The code review phase took one person-day. It allowed the detection of four errors (see Fig. 3). Those were corrected before the random testing phase. Information from the code review was used to target random tests to suspicious parts of the code. This testing phase revealed five new errors or suspicious situations in one day.

The second team applied a combinatorial testing approach based on the informal requirements: the requirements document was used as a basis for the design of test inputs. First, seven general properties from the requirements were identified. Then, some "abstract scenarios" were expressed to define sets of similar test cases. The Tobias tool was used to instantiate the abstract scenarios into executable JUnit test cases. Seventeen abstract scenarios were produced, which were unfolded into 1,241 test cases. Those represented 40,000 Java code lines (for JUnit).

In parallel with the test execution, the second team performed a critical analysis of the execution results. This helped us to find some cases where the code and the JML specification were consistent, but were different from the requirements or contrary to common sense. It took 6 person-days to analyze the specification, produce

	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

Fig. 3 Errors detected

the abstract scenarios, execute the tests and analyze the traces. Sixteen errors or suspicious situations were discovered by the execution of these tests.

The testing efforts of both teams aimed at discovering inconsistencies between informal requirements, JML specification and code. Three cases were identified: JML specification and code are inconsistent (1); JML specification and code are consistent and both are inconsistent with informal requirements (2); JML specification, code and informal requirements are consistent but overlook common sense requirements (3).

At the end of both processes, 18 different errors or suspicious situations were identified (Fig. 3). We say that there is an error when the JML assertion checker raises an exception. Java exceptions also often reveal errors in the code[5] (case 1).

[5] Or reveal a missing *exsures* clause.

We call suspicious situations the cases where the formal specification and the code have the same behaviour, but do not correspond to the informal requirements or to common sense (cases 2 and 3).

Each error was carefully analysed in order to classify them. Figure 3 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.

- *Floating-point approximations*
  There are five cases related to floating-point approximations (errors 3, 4, 5, 6, 18). The `float` type is used to represent the account balance. The errors are revealed when the postcondition and the code compute the same "value" in different ways. For example, $(x + y) - z$ is not always equal to $x + (y - z)$ when x, y, z are float numbers. With float numbers, + and - operations are not commutative due to their limited precision[6] *(case 1: JML specification and code inconsistent).*
- *Erroneous JML specification*
  Three 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 should have been ($a == \backslash old(a)$), expressing that the value of the attribute has not been changed.[7] 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. It was actually detected by code review *(JML specification and code both inconsistent with informal requirements).*
- *Limit*
  There are four cases that are dealing with "limits" (err. 1, 2, 11 and 12), i.e. boundary values. Let us detail 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 code and in the JML specification. *(JML specification and code both inconsistent with informal requirements).*
  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. This is a typical example where the success of a test actually reveals a problem. *(JML specification, code and informal requirements inconsistent with common sense).*
- *Design mistake*
  Errors 9, 10, and 13 have been classified as design mistakes. One critical attribute is `public` instead of being `private` (err. 10). It is possible to assign the same identifier to two different accounts if two account managers are created (err. 9)*(JML specification and code inconsistent)*. The banking application deals with threads, but there is no protection (i.e. critical section) to prevent a concurrent

[6] During the verification process, the approximation problem was not addressed.

[7] These properties could have been expressed with the JML keyword *not_modify*.

access an account (err 13). These errors were revealed either by the tests or by code review.

- *Counter-intuitive behaviour*
  Errors 16 and 17 denote counter-intuitive behaviours. In fact, it is possible to delete an account on which there are some active saving or spending transfer rules. This case is neither specified informally nor formally. So, it is not possible to conclude whether the application behaviour 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. *(JML specification, code and informal requirements inconsistent with common sense)*.
- *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.

```
/*@ requires true;
  @ ensures input == null ==> \result == 0;
  @ exsures (Exception e) false; @*/
public float inputToAmount(String input) {
...
if (input == null) {... return 0; }
            else { amount = new Float(input); ... } }
```

  If this method is called with an incorrect string (for instance `inputToAmount("aaa")`), it will raise an exception when calling `new Float(input)`. This can therefore be considered as an error in the specification: the `exsures` clause should be modified to allow this exception or the precondition should be stronger in order to exclude illegal input values. *(JML specification and code inconsistent)*
  The fact that the input should be a string is not indicated in the informal requirements. This error can thus be considered as a design mistake (the parameter should have been typed as `float`).

Expressiveness of JML Using JML, seven out of 18 errors were detected. 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 specification. Table 3 shows which kind of properties were actually detected using JML and which ones could have been detected if the specification was more complete.

Table 3 Potential to detect more errors with JML assertions

Error type	Detected by JML assertions	Detectable by JML assertions
Limit	No	Yes
Floating point	Yes	
Postcondition	Yes	
Design	No	One of the three errors
Counter-intuitive	No	Yes

3.4 Partial Conclusion

At the beginning of this section, two questions were asked. This section will now try to answer them, and add some lessons about the choice of a testing strategy.

- *Can the banking application JML specification dedicated to verification be used for testing?*
  It is very attractive to reuse the same specification with several tools. Here the specification was first created for verification purposes then reused for testing purposes. Before the verification process, one writes the specification focusing on the main parts, i.e. what has to be verified (invariants, pre and post-conditions). Then, during the verification process, some new assertions are added (mainly *annotation statements*), to help the verification tool.
  For the testing process, tools take advantage of the fact that a large subset of JML assertions are executable. Non-executable assertions are expressions that can not be translated to Java due to various factors. For example, `\forall` or `\exists` have to be iterated over a finite range of integers or a JML set to be executable. Since verification tools do not need JML assertions to be executable, only a subset of the specification can be reused.
  About the executable part of the specification, one should notice that the annotation statements are often too close to the Java code to help find errors. But although they do not contribute to the test oracle, they do not harm the testing process. These elements of the specification can even be useful for regression testing, provided they are sufficiently abstract to express the functionalities and not how they are implemented. The only negative influence of these specification statements is that they increase the size of the specification and tend to give some misleading confidence that the specification is complete.
  In summary, only the executable part of the JML specification can be reused for testing. This may include annotation statements which are too close to the code to reveal errors.
- *Is the JML specification detailed enough to allow accurate validation by testing?*
  JML has a good expressiveness to cover most of the requirements of the banking application. Unfortunately, like most formal languages it faces the risk of incomplete specifications: while writing specification for verification purposes, the type of properties and the way they are written are implicitly influenced by the ability of the verification tools. For example, for the Banking application, engineers deliberately chose not to describe properties about the float values, since they knew that JACK could not handle them. Thus, only seven of the 18 errors were found because they violated JML assertions. But 80% to 90% of the errors could have been detected if adequate JML assertions had been available (see Table 3). This reveals the incomplete character of the provided specifications which reduces the testability of the application.
  We divided the errors into three categories. Category 1 corresponds to the seven errors we discovered where code and specification were inconsistent. Five of them are related to floating point errors and could not be detected by the verification process because JACK does not support float variables. The two other errors (nine and 13) could have been detected by a verification process.
  The remaining errors (categories 2 and 3) could not be revealed by the specification. Category 2 corresponds to the incompleteness of the specification with

respect to the requirements. Here, we believe that the systematic use of traceability techniques can help reduce this incompleteness by clearly marking the requirements not covered by the specification. Finally, category 3 corresponds to the incompleteness of the requirements documents. This problem should be addressed with adequate requirements engineering techniques but definitely remains a difficult issue. It must be noted that the use of more sophisticated specification-based testing techniques or the use of verification techniques would not have improved our capability to detect these errors since such techniques only find errors covered by the specification.

- *About the choice of a testing strategy*
  The two testing approaches have not revealed exactly the same errors. The first approach, combining code review (human validation) and random testing (automated data selection and oracle), allows one to detect two errors unfound by the second approach. On the other hand, the second approach based on the study of the informal requirements and on combinatorial testing (manual data selection, human and automated oracle) detects nine errors unfound by the first approach. It is important to notice that nine of these 11 errors were detected thanks to human analysis and correspond to categories 2 and 3. This makes us think that the ability to find faults automatically (revealed by runtime errors or JML assertion violation) does not depend on the testing approach but on the accuracy of the JML specification.

4 From Testing to Verification

This section focuses on the work done to reuse a JML specification dedicated to testing, for verification activities. Section 4.1 deals with the presentation of the case study, a Home Network Services application. Section 4.2 briefly describes the testing work. Section 4.3 details the work done for verification, and especially focuses on the refactoring of the code and the specification. Section 4.4 proposes a partial conclusion.

4.1 The HNS Case Study

The second case study deals with Home Network Services (HNS). HNS consist of one or more networked appliances connected to a LAN at home. One of the major HNS applications is the integrated services of networked home appliances (called *integrated service* in the following). An integrated service orchestrates several home appliances via a network in order to provide more comfortable and convenient living for the users. For instance, the *DVD Theater Service* turns on a DVD player, switches off the lights, selects 5.1ch speakers and adjusts the volume automatically (Fig. 4).

Nakamura et al. have proposed a framework that adapts the legacy appliances with conventional infrared remote controllers [22, 23]. The key ideas are (1) to use a programmable infrared remote controller to control the different appliances, and (2) to rely on a service-oriented architecture (SOA) (see [19, 25]).

For each appliance, a self-contained component is implemented in Java and deployed as web service (using Apache AXIS) (Fig. 5). Methods like `On()` and `Off()` are open interfaces for accessing basic features of the appliance. They use a

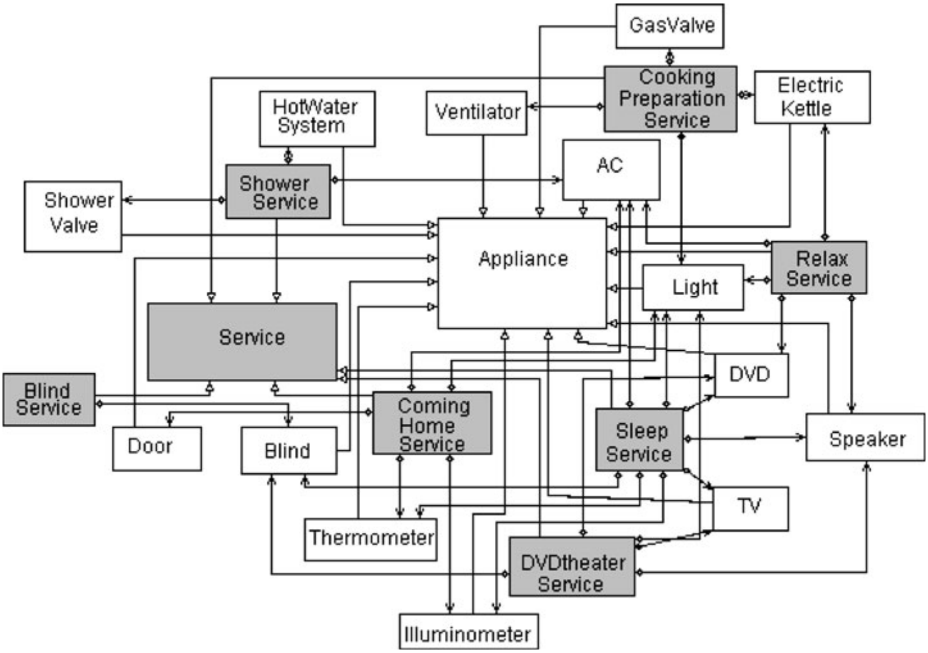
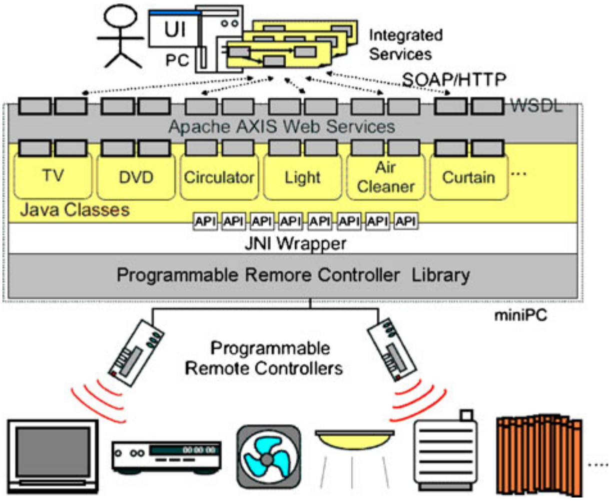


Fig. 4 Class diagram of the HNS application, restricted to appliances and integrated services (*in grey*)

set of APIs by which the PC can send infrared signals to the appliances (Ir-APIs). Ir-APIs have been implemented by wrapping the programmable infrared remote controller with a Java Native Interface (JNI Wrapper).

Integrated services can be implemented in this framework as client applications. An integrated service invokes the methods of the appliance components. The application was mainly developed in Nara and Kobe universities. The core of the

Fig. 5 HNS



application, consisting of appliance and integrated services, has been developed under several versions. The one which is studied here was developed by a Master student at the LIG labs. The Java code and the JML assertions were written in parallel by this student.

The initial version was composed of 25 classes among which 14 were appliance components and seven, integrated services (Fig. 4). One level of inheritance was introduced for the appliances and the integrated services. A specific class (`HomeEnvironment.java`) describes the sensor values (temperature, light level,...). The version has 2,000 lines of code. In this code, 209 JML annotations were inserted (17 preconditions, 150 postconditions, and 42 invariants).

4.2 Testing the HNS Services

To test the appliance and integrated services, we adopted a combinatorial approach based on the informal requirements as for the Banking application. More than 30 test schemas were described and unfolded by Tobias [11]. Each schema has between 500 and 5,000 test cases. Tests cases were then translated in the JUnit format and executed within the Eclipse environment. 10 errors were found (at the appliance and service levels). These errors reveal mainly inconsistencies between code and JML specifications.

4.3 Verifying the HNS Services

Two types of works were done to verify the HNS classes: one with ESC/Java and the other with deductive verification tools (JACK, KeY, Why/Krakatoa). The code and annotations were modified during the verification process, in order to correct the errors or to detail incomplete parts, and to continue the verification process. During the work, we tried to answer the following questions:

1. Can the HNS specification dedicated to test be used for verification?
2. If no, how should the testing specification be modified to support verification?

4.3.1 Using ESC/Java

ESC/Java was the first tool to be used during the verification process. The verification was carried out on the code that was tested and corrected. The verification of the Java classes was long and uneasy. The JML assertions were not sufficient to allow an automatic verification, even if they allowed to perform testing.

The first ESC warnings were obtained for `Appliance.java`. They were related to the use of set methods of `HomeEnvironment.java`. The use of these methods in `Appliance.java` could lead to a violation of `HomeEnvironment` assertions. In order to solve the problem, `Appliance.java` class had to be refactored. This operation has impacted all subclasses of `Appliance.java`.

Then, to remove several warnings, several JML assertions had to be inserted. Indeed, several postconditions were added to specify the returned result. For instance, for the method `public String getPower()`, the following postcondition was added `//@ ensures \result.equals(powerState);`.

Several warnings were related to the problem of null values. Some assertions were added in order to specify that the variables or the attributes (of type String) were not

```
public  class Appliance {
  protected /*@  spec_public @*/ String Name; // used for printing messages
  protected /*@  spec_public non_null @*/ HomeEnvironment currentEnv ;
  protected /*@  spec_public non_null @*/ String  powerState = "OFF";
  protected /*@  spec_public non_null @*/ String  internalState = "OFF";
  //@ public invariant (!powerState.equals("OFF") ==> powerState.equals("ON"));
  //@ public invariant (!powerState.equals("ON") ==>  powerState.equals("OFF"));
  //@ public invariant (!internalState.equals("OFF") ==> internalState.equals("ON"));
  //@ public invariant (!internalState.equals("ON") ==> internalState.equals("OFF"));

  //@ public invariant minConsumption<=maxConsumption;
  //@ public invariant minConsumption >= 0 ;
  //@ public invariant applianceCurrentConsumption>=0 ;
  //@ public invariant applianceCurrentConsumption<=maxConsumption;
  //@ public invariant powerState.equals("ON")==>(applianceCurrentConsumption>=minConsumption);
  //@ public invariant powerState.equals("OFF")==>(applianceCurrentConsumption==0);
  protected /*@  spec_public  @*/ int maxConsumption =0;
  protected /*@  spec_public  @*/ int minConsumption=0;
  protected /*@  spec_public  @*/ int applianceCurrentConsumption=0;
 [..]
  //@ modifies \everything;
  //@ requires powerState.equals("ON");
  //@ ensures powerState.equals("OFF");
  //@ ensures internalState.equals("OFF");
  //@ ensures applianceCurrentConsumption == 0;
  public /*@  spec_public  @*/ void  powerOff(){
        internalState = "OFF";
        powerState="OFF";
        applianceCurrentConsumption = 0;
        System.out.println(Name + " is powered off");
  }
  public /*@ pure @*/ int getConsumption(){return applianceCurrentConsumption;}

  //@ ensures \result != null;
  //@ ensures \result.equals(powerState) ;
  public /*@ pure @*/ String getPower(){ return powerState; }

  //@ ensures \result != null;
  //@ ensures \result.equals(internalState) ;
  public /*@ pure @*/ String getInternalState(){ return internalState;}

  //@ requires powerState.equals("ON");
  //@ requires internalState.equals("ON");
  //@ ensures internalState.equals("OFF");
  protected /*@  spec_public  @*/ void  switchOff(){
        setApplianceMinConsumption();
        internalState="OFF";
        System.out.println(Name + " is switched off");
  }

  //@ requires powerState.equals("ON");
  //@ requires internalState.equals("OFF");
  //@ ensures internalState.equals("ON");
  protected /*@  spec_public  @*/ void  switchOn(){
        internalState="ON";
        setApplianceMaxConsumption();
        System.out.println(Name + "is switched on");
  } [..] }
```

Fig. 6 JML annotations for the appliance java class

null. Moreover, all constructors of appliances were modified in order to initialize all attributes explicitly.

After code refactoring, the code size represents 2,400 lines of code. The new JML assertions represent more than 600 lines of code (see the code of Appliance Fig. 6). At the end of the process, all appliance and service properties seem to be validated:

there was no remaining warning. However, one has to be careful. ESC/Java is neither complete nor sound. Some errors may not be reported and false alarms may be issued.

From a general point of view, the verification of the code with ESC/Java required more work than expected. Indeed, the effort spent to complete the specification to help the tool was underestimated for two reasons. First, the code of the appliance API and integrated services is quite simple (no loop for instance). Second, the testing phase did not reveal inconsistencies between the assertions and the code. So it was expected that verification would be easy. Actually the verification process did not detect additional errors.

4.3.2 Using JACK, Why/Krakatoa and KeY

JACK, Why/Krakatoa and KeY were successively used in order to perform the verification. The process was difficult, and the result was not as good as expected (for the three tools).

A difficulty was that the JML version used for the project was no longer compatible with JACK. Moreover, Krakatoa (in the version used) did not accept the whole syntax of JML. For instance, assertions such that `non_null` or `pure` were not accepted. Several files had to be modified.

Regarding the use of KeY, the verification could be carried out only for file `HomeEnvironment.java`. The main reason was that the JML assertions deal with strings, which are currently not supported by KeY.

After the use of ESC/Java, the class `HomeEnvironment.java` has 27 methods (1 constructor, 11 get methods, 15 set methods) and three invariants. Each get method has been declared “`pure`”. Half of the set methods were associated with a precondition. None of them has a postcondition. KeY produced between 3 and 5 proof obligations for each method. All of them were verified. Most of them were verified automatically with Simplify or Yices provers. For three methods, the “elementary arithmetic strategy” had to be used. For one method, we had to increase the number of computing steps (1,100 instead of 1,000 by default).

A new refactoring of the code was carried out. The attributes of type String were in fact used to implement an enumerated type. The code and the JML assertions were modified so that integers were used to implement those enumerated types.

KeY was then used again on a small part of the application. It was possible to verify automatically more than one half of the proof obligations related to appliances and integrated services, with the help of Simplify and Yices provers. However, some proof obligations are still pending. No additional error was found.

4.4 Partial Conclusion

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

- *Can the HNS specification dedicated to testing be used for verification?*
  Properties which were stated for the test process were properties we wanted to be verified. In that sense, the JML specification can be reused. However, some elements must be taken into account.

First, the verification process is limited by the power of the tools: some parts of the JML language that is supported by the testing tools are not currently supported by the verification tools. In fact, the same properties written during the verification process would probably have been designed differently, in order to help the verification tools. For instance, some properties about the initial states of the appliances or services could have been specified using the *initially* clause (which is not executable). So, some parts of the testing specification are not usable.

Moreover, the testing specification is possibly incomplete. Properties were written with the objective to be used for testing. So, the tester implicitly chose to write properties with respect to an appropriate (executable) subset of JML. With a larger subset, other properties could have been specified. In the HNS specification for instance, quantifiers and the `initially` clause are never used.

- *If no, how should the testing specification be modified to support validation by verification?*

  It must be noticed that in this experiment, a large effort was required to adapt the code and the JML specification to the verification process. Improvements only restructured the code and increased the redundancy of the specification. We were not aware that it corrected any bug in the code or in the specification, since no failure was exhibited.

  We can distinguish two types of works to support verification process. First, one has to provide code and specification compatible with the tool abilities. During our experiment, we had to refactor both the code and the JML specification in order to carry out the verification. A first refactoring was carried out for ESC/Java. It mainly consisted of (1) a simplification of the coupling of the methods and classes, and (2) a multiplication by four of the size of the JML specification. A second refactoring was needed by the deductive provers, in order to translate enumerated type (from String to Integer).

  Second, one may have to add specific assertions to help the tools. For this application, the assertions to be inserted were mainly related to null values. It also concerned some indications of the value of the returned results. In this application, there were no loops, so no loop assertions were required. However, it is quite usual to add those types of assertions to help the verification process.

5 Conclusion and Lessons Learnt

This article reports on two case studies implemented in Java and specified in JML (Java Modelling Language). We specially address the problem of reusing a JML specification produced for one activity (resp. verification or test) for the other (resp. test or verification). These two case studies bring about interesting lessons on the use of JML in a Java validation process.

Writing a Specification (in JML) In the HNS case study, the JML assertions were written during coding, for testing purposes. A part of the assertions were devoted to express the internal consistency of the classes. For instance, we expressed the expected value of one attribute with respect to the values of the other attributes. Those assertions were really useful during the whole development process, as a way to maintain consistency among the classes during the different evolutions.

In the banking application case study, 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. Therefore, care should be taken to express the specification in a different, and often more abstract, way. This should increase implementation freedom, and result in specifications which are more robust to evolution.

The copy-paste effect is a problem especially for testing. From the two case studies, we noticed that the copy-paste effect had less impact on invariants than on postconditions. This is due to the fact that writing invariants requires to step back, since it will concern the class in a global way and not only a method, as it is the case for postconditions. For this reason, one should favour the identification of invariants when writing a JML specification, especially for existing code.

An important point to notice is that JML is not completely supported by the different tools. For instance, verification tools such as ESC/Java, JACK, Why/Krakatoa and Key support only a part of the JML constructions. Similarly, the JML runtime assertion checker (`jmlrac`), used for test, supports only executable features of JML (for instance, a `\forall` is not executable if the following expression does not concern a JML set or an integer interval). So, assertions are written with an adequate subset of JML, with respect to the approach plan to be used.

From a methodological point of view, it seems more appropriate to write the specification independently of the code and to write first the invariants to reduce the copy-paste effect, then pre and postconditions. If testing is planned, one should check if all the assertions are executable. If it is not the case, an executable expression should be added. If verification is planned, annotation statements should be added.

Using a Specification for Testing or Verification For a JML specification to be usable for testing or verification, one should pay attention to two elements. First, one should keep in mind that the specification is never complete and can possibly be inconsistent with informal requirements. In the first case study, the analysis shows that the two testing approaches have comparable outcomes: they detect quite the same inconsistencies between the formal specification and the code. However, the human analysis of respectively the code and the test results raise additional errors.

A second point that needs attention is the fact that the code and the specification should be designed so that the verification and both testing tools could be used. It is especially critical for the verification process, which is possible only if the construction (code and assertions) are supported. It is also the case for the testing process, for instance, results should be observable. So in both case, during the whole development, one should "design for test" and/or "design for verification", that we can summarize as "design for validation and verification".

Reusing a Formal Specification Formal specifications are aimed to be used several times during software development, and it is often the case that the intended use of the specification influences its style and contents. But, the way the specification will be used (resp. the tools that are going to be used) has an influence of how the specification has to be written. Reusing a specification should then be done carefully.

A JML specification designed for verification contains several assertions, such as invariants associated with constants or loop invariants, which were added specifically for verification in order to help the tools. Unfortunately this over-specification can become an obstacle to evolutions of the system.

Over-specification is not spread uniformly in the specification. For the banking case study, several methods were clearly under-specified (the postcondition is not stated) and it was not possible to make a judgement on the correctness of their execution.

A specification designed for test tends to express a judgement about the results. It generally specifies the expected behaviours. The main requirement is that the assertions have to be executable. In order to verify such a specification, refactoring may have to be carried out for both code and assertions. In the case of the HNS application, it is still not clear for us if the refactoring had to be carried out in order to help the tools or in order to correct some remaining errors, since no failure was demonstrated before the refactoring.

As a conclusion, since specification description is motivated by different concerns, it should be interesting to use structuring and documentation mechanisms that identify parts of the specification according to their rationale and intended use. In particular, each assertion of the specification should either be linked to the requirement it expresses or marked as a proof annotation.

Acknowledgements The work on the first case study was partially supported by the COTE RNTL project (http://www.irisa.fr/cote/). The work on the second case study was partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Young Scientists (B) (No. 18700062), Scientific Research (B) (No. 17300007), and Comprehensive Development of e-Society Foundation Software program. It is also supported by JSPS and MAE under the Japan-France Integrated Action Program (PHC-SAKURA). A special thanks to Natasha King who corrects the English phrasing.

References

1. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY tool. Softw. Syst. Model. **4**, 32–54 (2005)
2. Barthe, G., Burdy, L., Charles, J., Grégoire, B., Huisman, M., Lanet, J.-L., Pavlova, M., Requet, A.: JACK—a tool for validation of security and behaviour of Java applications. In: 5th International Symposium Formal Methods for Components and Objects (FMCO). Lecture Notes in Computer Science, vol. 4709, pp. 152–174. Amsterdam, The Netherlands (2006)
3. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: the KeY Approach. Springer, New York (2007)
4. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. In: Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS'03). Electronic Notes in Theoretical Computer Science, vol. 80, pp. 73–89 (2003)
5. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. STTT **7**, 212–232 (2005)
6. Burdy, L., Requet, A., Lanet, J.-L.: Java applet correctness: a developer-oriented approach. In: The 12th International FME Symposium, Pisa, Italy (2003)
7. Chalin, P.: Early detection of JML specification errors using ESC/Java2. In: Proceedings of the 2006 Conference on Specification and Verification of Component-Based Systems (SAVCBS), pp. 25–32. Portland, Oregon (2006)
8. Cheon, Y., Leavens, G.T.: A runtime assertion checker for the java modeling language (JML). In: Arabnia, H.R., Mun, Y. (eds.) International Conference on Software Engineering Research and Practice (SERP '02), pp. 322–328. Las Vegas, Nevada (2002)

9. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: the JML and JUnit way. In: 16th European Conference on Object-Oriented Programming (ECOOP'02), pp. 231–255 (2002)
10. du Bousquet, L., Ledru, Y., Maury, O., Oriat, C., Lanet, J.-L.: A case study in JML-based software validation (short paper). In: Automated Software Engineering (ASE). Linz, Austria (2004)
11. du Bousquet, L., Nakamura, M., Yan, B., Igaki, H.: Using formal methods to increase confidence in one home network system implementation. Case study. In: Workshop on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2007). Revue des Nouvelles Technologies de l'Information, vol. RNTI-SM-1. Poitiers, France (2007)
12. Filliâtre, J.-C., Marché, C.: The why/krakatoa/caduceus platform for deductive program verification. In: 19th International Conference Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 4590, pp. 173–177. Berlin, Germany (2007)
13. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: PLDI, pp. 234–245 (2002)
14. JML: The java modeling language (JML) home page (2008). http://www.cs.iastate.edu/~leavens/JML.html
15. JUnit: JUnit (2008). http://www.junit.org
16. Leavens, G., Baker, A., Ruby, C.: JML: a notation for detailed design. In: Kilov, H., Rumpe, B., Simmonds, I. (eds.) Behavioral Specifications of Businesses and Systems, pp. 175–188. Kluwer, Dordrecht (1999)
17. Ledru, Y., Dadeau, F., du Bousquet, L., Ville, S., Rose, E.: Mastering combinatorial explosion with the tobias-2 test generator. In: 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), pp. 535–536. USA (2007)
18. Ledru, Y., du Bousquet, L., Maury, O., Bontron, P.: Filtering TOBIAS combinatorial test suites. In: Fundamental Approaches to Software Engineering (FASE'04). LNCS, vol. 2984. Barcelona, Spain (2004)
19. Loke, S.W.: Service-oriented device ecology workflows. In: First International Conference on Service-Oriented Computing (ICSOC 2003). Lecture Notes in Computer Science, vol. 2910, pp. 559–574. Trento, Italy (2003)
20. Marché, C., Paulin-Mohring, C., Urbain, X.: The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. J. Log. Algebr. Program. **58**(1–2), 89–106 (2004)
21. Maury, O., Ledru, Y., du bousquet, L.: Using TOBIAS for the automatic generation of VDM test cases. In: Third VDM Workshop (in conjunction with FME'02) (2002)
22. Nakamura, M., Tanaka, A., Igaki, H., Tamada, H., Matsumoto, K.: Adapting legacy home appliances and web services. In: Int. Conf. on Web Services (ICWS 2006), pp. 849–858 (2006)
23. Nakamura, M., Tanaka, A., Igaki, H., Tamada, H., Matsumoto, K.: Constructing home network systems and integrated service using legacy home appliances and web services. Int. J. Web Serv. Res. **5**(1) (2009)
24. Oriat, C.: Jartege: a tool for random generation of unit test for java classes. In: First International Conference on the Quality of Software Architechtures and Second International Workshop of Software Quality (QoSa/SOQUA). Lecture Notes in Computer Science, vol. 3712, pp. 242–256 (2005)
25. Papazoglou, M.P., Georgakopoulos, D.: Special issue: service-oriented computing. Introduction. Commun. ACM **46**(10), 24–28 (2003)