On the Value of Fault Injection on the Modeling Level

Bernhard K. Aichernig

International Institute for Software Technology
United Nations University, Macao SAR China
P.O. Box 3058, Macao
bka@iist.unu.edu

Abstract. In this paper we advocate the integration of fault injection into a formal methods tool set. We start by giving some motivations from our past industrial experiences with VDM. Then, we report about our recent work on specification-based test case generation via model mutation. In this testing approach the idea is not to cover a specification structurally, e.g. all expressions/statements, but to cover a predefined set of possible faults, like semantic misunderstandings. A prototype tool has been developed that generates such test cases that could be easily adapted to VDM specifications. Finally, the validation role of fault injection is discussed.

1 Introduction

In this paper we advocate fault injection as a further tool in model-based development. Traditionally, fault injection has been used to test the fault-tolerance of systems or to assess the quality of test cases. Here, we concentrate on the second aspect that has become known as *mutation testing*.

Mutation testing is a fault-based testing technique introduced by Hamlet [7] and DeMillo et al. [5]. It is a means of assessing test suites by injecting faults into a program text. If the test suite cannot detect some of the faults it needs to be improved. The idea is to introduce small changes, so called program *mutations*, that represent programmer's errors commonly found in software. Usually, a set of deviating programs, the *mutants*, are generated, each containing one single fault.

What kind of mutants to be generated is determined by the *mutation operators* mapping language elements to defined alterations. Typical operators exchange variable names, increase or decrease numeric constants by one, alter comparison operators etc. Part of the mutation operators are language dependent. In the C language family, for example, replacing an equivalence == for an assignment = would be such a language dependent mutation operator modeling a common fault in these programs.

A major obstacle in this approach are *equivalent mutants* containing mutations that do not represent errors. Hence, no test case exists that is able to distinguish these mutants from the original. This is a major problem in assessing test cases, since if a mutant passes all tests it is not clear if the test cases are insufficient or if the mutant simply does not represent an error. Therefore, in case all mutants passes the tests, the tester needs to inspect the mutant if it constitutes an equivalent mutant.

This testing technique is called *fault-based*, because the faults injected by the set of mutation operators determine the test coverage. Here, the aim is that the test suite covers all anticipated faults in a program. This is in sharp contrast to structural testing strategies, where testers try to cover certain parts or paths in a program. Typical examples of this kind of coverage include statement coverage (each program statement should be executed once), branch coverage (each branch in the control flow should be executed once) as well as the data-flow criteria (e.g. for every variable each path from the variable's value definition to its use should be executed).

Structural test coverage has become the most dominant test coverage paradigm in model-based testing, too. On this more abstract level, test cases are designed to cover, e.g.,

- all the expressions in a formal specification written in a model-based specification language like B, VDM-SL, or Z;
- each disjunct in the disjunctive normal form of an interface pre-postcondition specification;
- the states and transitions of (Extended) Finite State Machines;
- the exhaustive set of traces explored by model checkers.

An underlying assumption taking this approach to model-based software testing is that the implementation under test reflects the structure of the model. However, this depends highly on the level of abstraction, and seems more promising for low-level design models like state-transition diagrams, or abstractions generated in a bottom-up fashion as typically used in model checking. For highly abstract models found in top-down development methods we advocate a fault-based testing strategy.

Fault-injection on the modeling level can serve two purposes:

1. to generate test cases for testing an implementation
2. to validate the model

Before these two roles of fault-injection are discussed, we give some motivation by looking back on our past experiences in Section 2. Then, we discuss test case generation by fault-injection in Section 3 and model validation in Section 4. Section 5 presents our concluding remarks including pointers to related work and a recommendation for putting the presented technique on Overture's agenda.

2 Experience from Two VDM Projects

In this section we summarize the lessons learned from two industrial projects using VDM with respect to testing [2, 8].

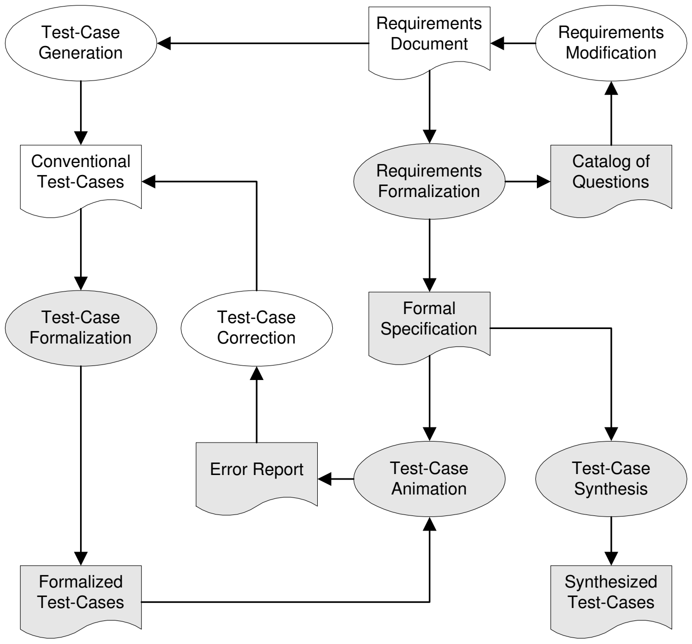

Fig. 1. Validation of test cases and requirements using executable VDM models.

2.1 The Projects

Both projects were in cooperation with Frequentis (www.frequentis.com), a company producing voice communication systems for air traffic control. The first project used VDM++ for modeling their voice communication system VCS 3020. The aim was to validate the functional requirements and to assess the quality of the existing system test cases. The second project targeted a network of such VCSs designed to transfer air traffic control tasks between different airports. Again, we focused on the requirements as well as on the acceptance tests produced in parallel to our modeling efforts.

The project followed the method of formalizing the requirements in an executable VDM model and then running the test cases on it. Figure 1 depicts the method in detail:

1. The starting point is the informal requirements documentation containing the user and software requirements of the system that should be tested.
2. Usually, in an industrial environment a traditional test design process is already established. This traditional test design method takes the informal requirement documents and produces one test-case for each requirement.
3. In parallel to the conventional test design, a formal methods expert creates an abstract VDM prototype out of the informal requirements descriptions. This formalization process usually highlights a series of open issues in the

requirements. These are documented in a *catalogue of questions* concerning the requirements.

4. The requirement engineer reacts to the questions in the catalogue. Some questions are due to a lack of domain knowledge of the specifier, others need a clarification and then a modification of the requirements. In some cases the questions have to be forwarded to the customer (user) of the system under development — validation has been initiated.
5. When test-cases are available, these are formalized, as well. The formalized test-cases are sequences of calls to the operations of the abstract prototype. Furthermore, observer routines are used for checking the test results as specified in the conventional test documents. This phase uncovers errors in the test-cases, like missing steps, or wrong input/output specifications.
6. The abstract prototype is executed (animated) with the formalized test-cases. The coverage information that is collected during the test runs highlights uncovered parts in the requirement specification.
7. Additional test-cases are designed from the test-specification such that 100 % of the specification are covered.

2.2 Issues in the Requirements and Test Documentation

System-level testing is based on the user and software requirements of the system under test. Consequently, the quality of the test-cases depends on the quality of the requirement documentation. Common sources of errors in the test case design are

- missing requirements,
- ambiguous requirements, or even
- conflicting requirement descriptions.

This is especially valid for large projects, where system-level tests are designed and carried out by independent test engineers. During the formal specification of the VCS system in the first project a total number of 64 issues have been found in the various documents of the system. In the second project the formalization of 140 requirements uncovered 108 open issues leading to 33 changes in the requirements document.

A further source of errors is the complexity of the system-level test-cases itself. Usually, several interactions with the system are necessary for establishing an internal state prior to the testing of a given requirement. Designing such a test from an informal prose document by hand leads to several mistakes in the test case descriptions.

Common errors that have been found in the test documentation during our two experiments are

- missing test cases,
- missing interactions (test steps) in a test case,
- wrong interactions (test steps),
- wrong input-data of a test,

– wrongly predicted test results.

We used the IFAD VDM Tools to run the existing system test cases in both projects. The tool visualizes which expressions have been covered by the test executions highlighting missing parts in the specification.

In the first project the internal system test cases were executed. Since they had been in use for a while the test case documentation was quite mature, no important faults in the test cases were reported. However, their coverage was far too low. The execution on the VDM model showed that only approximately 80% of the formal specification were covered by the test cases, leaving some 20% of the specification's functionality untested!

In the second project, the acceptance test cases to be analysed were rather new. This time the existing test-cases covered 100 % of the abstract prototype, but the 65 test-cases with 200 test-steps contained 16 faults. Since almost all detected faults occurred in different test-cases, this means that approximately 25 % of the test-cases were faulty.

2.3 Lessons learned

Test cases enhance communication. The engineers liked the availability of test-cases to validate our formal specifications. They did not like to read the VDM models, but with the test cases run on the model they were assured that the VDM model reflects their requirements. We actually view test cases as a special form of specification, well suited to communicate with partners not familiar with a model.

Test cases need to be validated as well. The high number of faults in the test cases is a problem. If testers cannot rely on their test documentation it is likely that they will misinterpret test results. Therefore, the test cases need to be validated with respect to their consistency with the requirements. Existing test cases can be executed on an executable model as was done in our projects. Another way of ensuring consistency is to generate the test cases from the model.

Expression coverage not appropriate for assessing or generating test cases. We showed that the system test cases only covered about 80% of our VDM specification. However, our claim that this is not sufficient from a software engineering point of view is hard to defend. Are our additional 20% necessary? We assume so, because expression coverage is the most basic coverage known. However, the size and thus the number of expressions of a model depends on the skills of the modeller. Hence, a biased model would enforce test cases not motivated by the requirements. Thus, expression coverage is appropriate for testing a model, but questionable for assessing or generating test case for testing an implementation.

Control-flow based coverage criteria do not help. One might envision more advanced coverage criteria based on control-flow criteria. However, the control flow of a specification might be considerably different from the implementation under

```
context Ttype(a:int,b:int,c:int):String
pre:  a>=1 and b>=1 and c>=1 and a<(b+c) and b<(a+c) and c<(a+b)
post: if ((a=b) and (b=c)) then result="equilateral" else
      if ((a=b) or (a=c) or (b=c)) then result="isosceles" else
      result="scalene" endif endif
```

Fig. 2. Original specification of a triangle in UML's OCL.

test. In general, abstraction reduces the states and transitions in finite state machine descriptions. This raises the question, about the right level of abstraction with respect to system level testing and more important, how this level can be scientifically defended. Such a defence must be based on the efficiency of detecting errors. However, so far, we lack sufficient empirical evidence that structural coverage metrics applied to specifications are efficient.

Alternative: fault-based testing. A promising alternative is fault-based testing, since it explicitly links the test cases to be considered to the faults that can be detected. In addition, the question about the right level of abstraction can be answered: the level of abstraction is accurate if all the desired faults to be tested can be modeled in the specification. In the following, we discuss this approach in more detail.

3 Test Case Generation by Fault Injection

Every test selection strategy relies upon a test hypothesis that justifies the reduction of the generally infinite number of test cases needed for showing correctness. When we use fault injection for generating test cases we assume that we can anticipate the possible errors made by an implementer. In a sense, we are replying to Dijkstra's famous statement that testing can never show the absence of test cases but only their presence:

> *Testing can show the absence of faults, if we have a knowledge of what can go wrong.*

In fact, it is a common strategy of testers who are domain experts to test for known weaknesses of a system or for common errors made by implementers. We claim that having a model available, the tester has a more systematic way of reasoning what might go wrong by mutating the model. In fact, fault injection can be done

1. automatically by a set of mutation operators,
2. manually by interactively altering the specification.

.

Consider the well-known Triangle example, specified in Fig. 2. The function **Ttype** returns the triangle type that three lengths represent. This specification can be mutated introducing possible faults like changing a variable's name for any other valid name, or changing the order in a nested *if-statement* as shown in Fig. 3. The idea is to generate two test cases $a = 1, b = 2, c = 2, result = “isosceles''$ and $a = 1, b = 1, c = 1, result = “equilateral''$ that will detect the two errors (mutations).

We have developed a general theory of fault-based testing, in which it can be formally proven that test cases will detect certain faults [3]. This theory is based on the general concept of refinement (implementation relation) and can be instantiated to different semantic frameworks where this concept exists. We have instantiated the general theory to pre-postcondition specifications and developed an algorithm and tool. In the following, we describe the tool in detail. In its current form it accepts OCL input, but the front end could be easily adapted to accept, e.g. VDM, specifications.

3.1 Test Case Generation Algorithm

The test case generation algorithm relies on constraint solving of three conditions. It searches for a test case $t(i, O)$ with input i and a set of possible outputs O. For the proofs that the algorithm generates test cases that are adequate with respect to an error, we refer to [9].

Algorithm 1 *Given a specification with pre and postcondition $D(Pre \vdash Post)$ and its faulty design $D'(Pre' \vdash Post')$ as inputs. All variables in their alphabet range over finite sets. Then, an input-output test case T is generated as follows:*

Step 1: A test case T is searched by:

1. *finding a pair (i_c, o_c) being a solution of*
$$Pre \wedge Post' \wedge \neg Post$$
2. *If it exists, then the test case $T = t(i, O)$ is generated by finding a maximal solution (i, O) of*
$$Pre \wedge Post \wedge (v = i_c)$$

Step 2: If the former does not succeed, then we look for a test case $T = t(i, O)$ with (i, O) being a maximal solution of

$$\neg Pre' \wedge Pre \wedge Post$$

```
context Ttype(a:int,b:int,c:int):String
pre:  a>=1 and b>=1 and c>=1 and
      a<(b+c) and b<(a+c) and c<(a+b)
post: if((a=a) and (b=c))
      then result="equilateral"
      else if((a=b) or (a=c) or (b=c))
        then result="isosceles"
        else result="scalene"
        endif
      endif
```

```
context Ttype(a:int,b:int,c:int):String
pre:  a>=1 and b>=1 and c>=1 and
      a<(b+c) and b<(a+c) and c<(a+b)
post: if((a=b) or (a=c) or (b=c))
      then result="isosceles"
      else if ((a=b) and (b=c))
        then result="equilateral"
        else result="scalene"
        endif
      endif
```

Fig. 3. Two mutated specifications for the triangle example.

```
a=2, b=2, c=1, result=isosceles
a=2, b=3, c=4, result=scalene
a=1, b=1, c=1, result=equilateral
a=2, b=1, c=2, result=isosceles
a=1, b=2, c=2, result=isosceles
```

Fig. 4. DNF-based test cases

Note that the algorithm does not produce a counter example, but a positive test case $t(i, O)$. Therefore, Step 1 consists of two steps. First, the counter example for behavioural refinement (i_c, o_c) is generated. Then, the test input i_c of the counter example is used to find the predicted output (set) satisfying the original specification.

Note that this algorithm is partial, since the search space over the variables is restricted to a finite domain. We can say that if no test case is identified after those three steps, then the original and the mutant specifications are equivalent in the context of the finite domains of the constraint satisfaction problem. We will see in Section 4 that detecting behavioural equivalence plays an important role in validation of the models.

3.2 Example

We return to the Triangle example to illustrate the technique and the tool's functionality. The OCL specification for the Triangle example was already shown in Fig. 2.

Our tool is capable of generating test cases either in the classic way via DNF partitioning of the (original) OCL specification or by applying the fault-based algorithm (Algorithm 1). Choosing the DNF partitioning strategy the tool returns the test cases shown in Fig. 4. Here, for every equivalence class (domain partition) one test case is chosen. The strategy is to cover each partition.

In contrast to the DNF strategy, the fault-based algorithm generated test cases that cover faults. Generating the fault-based test cases for the two mutant OCL specifications in Fig. 3 results exactly in the two test cases $a = 1, b = 2, c = 2, result = \text{``}isosceles''$ and $a = 1, b = 1, c = 1, result = \text{``}equilateral''$ already presented in Section 3.

Analyzing these results we observe that the tool is generating valid test cases. Moreover, they are able to detect these kind of faults. However, also the DNF test cases of Fig. 4 would discover the two faults represented by the two mutants. Therefore, one could argue that the fault-based test cases do not add further value.

However, in general the fault-based strategy has a higher fault-detecting capability. Consider the two different mutated specifications shown in Fig. 5 below. One can easily see that the DNF test cases in Fig. 4 are not able to reveal these faults.

However, the fault-based algorithm generates precisely those test cases needed to unreveal the faults: $a = 2, b = 2, c = 2, result = \text{“}equilateral''$ for the first one, and $a = 3, b = 2, c = 4, result = \text{“}scalene''$ for the second one. Optionally, we may ask the tool to generate all fault-adequate test cases for every domain partition. Then, the additional test case $a = 1, b = 3, c = 3, result = \text{“}isosceles''$ for the second mutant is returned as well.

This example, although trivial, demonstrates the automation of an alternative approach to software testing: Instead of focusing on covering the structure of a specification, which might be rather different to the structure of the implementation, one focuses on possible faults. Of course, the kind of faults, one is able to model depend on the level of abstraction of the specification — obviously one can only test for faults that can be anticipated. For a larger example on testing the security policy of a database management system see [9].

Next, we discuss the validation aspect of our tool and fault-based testing in general.

4 Model Validation by Fault Injection

4.1 Testing Executable Models

Similar to the mutation testing of programs, fault injection can be used to assess and improve the test suite for validating an executable model. A possible strategy is first to design test cases that cover every expression as in the projects described in Section 2 and then, if the model passes all tests, a set of mutants is generated in order to check the fault-detecting power of these test cases. The test hypothesis is that if these test cases are able to find these small changes in the specification, then they are also able to find other more complex faults.

Especially useful for validation purposes is to inject the faults interactively. Therefore, UNU-IIST has recently extended the RAISE tools to support interactive mutation testing [6]. The current emacs interface of the tools allow to select a part of the specification and replace it with another term. Each mutant is stored separately following a standard naming scheme. When, sufficient mutants have been produced, the tests included in the RAISE specification are executed. With the help of the emacs diff tool the test results of the original and the mutants can be easily compared.

```
context Ttype(a:int,b:int,c:int):String
pre:  a>=1 and b>=1 and c>=1 and
      a<(b+c) and b<(a+c) and c<(a+b)
post: if((a=b) and (b=1))
      then result="equilateral"
      else if((a=b) or (a=c) or (b=c))
        then result="isosceles"
        else result="scalene"
        endif
      endif
```

```
context Ttype(a:int,b:int,c:int):String
pre:  a>=1 and b>=1 and c>=1 and
      a<(b+c) and b<(a+c) and c<(a+b)
post: if((a=b) and (b=c))
      then result="equilateral"
      else if((a=b) or (a=c) or (b=2))
        then result="isosceles"
        else result="scalene"
        endif
      endif
```

Fig. 5. Mutant Specifications for the Triangle Example

This feature is especially useful in controlling the testing efforts of a tester. UNU-IIST's fellows are often producing executable RAISE specifications of several kilo-lines. The fellows are typically asked to validate the specifications by means of test cases added to the specification. In one of our recent projects, 319 test cases found 28 errors in the RSL specification [1].

However, not always is testing done so thoroughly. A typical problem is that testing happens very late and the quality of the test cases suffer due to time restrictions. The mutation testing facility easily allows the responsible supervisor to control the quality of the test cases and, in addition, gives the fellow an immediate feedback how well he did his testing job. Controlling the quality of test cases is a general problem in practice.

4.2 Investigating Equivalent Mutants

Despite this obvious application of fault injection to models, there is a more subtle aspect of specification mutation with respect to validation. Here, one is interested in the equivalent mutants. More precisely, one tries to understand why some mutations do not represent an error. By doing so one gains a deeper understanding of a model, often redundancies in a model can be detected.

Imagine a postcondition specification being a conjunction of several expressions. By deleting a conjunct one would expect a different semantics, unless the deleted conjunct was actually redundant.

A prerequisite for such an analysis is a tool that reports the equivalence of two specifications. A fault-based test case generate might be used. If our tool, for example, cannot find a test case, we have an equivalence, at least in the context of the finite domain of its constraint solver. In a security example for validating the tool we actually had to update the rather small specification three times, since we found redundancies by this kind of fault injection analysis.

For behavioural specification languages, like CSP or CCS, refinement (CSP) or equivalence checkers may be used to detect equivalent mutants. Recently, Stepney's group in York reported such a validation case study using the FDR model checker to analyse CSP mutants that refine the original [10].

5 Conclusions

We reported on the role of fault injection in specification-based testing and validation. A tool has been developed to generate test cases for covering a set of injected faults. We showed that fault injection can also play an important role in the validation of executable or non-executable models.

5.1 Related Work

Others have worked on mutation testing on the specification level.

Tai and Su [13] propose algorithms of generating test cases that guarantee the detection of operator errors, but they restrict themselves to the testing of singular Boolean expressions, in which each operand is a simple Boolean variable that cannot occur more than once. Tai [12] extends this work to include the detection of Boolean operator faults, relational operator faults and a type of fault involving arithmetic expressions. However, the functions represented in the form of singular Boolean expressions constitute only a small proportion of all Boolean functions.

Perhaps the first one, who applied mutation testing to Z specifications was Stocks [11]. He presented the criteria to generate test cases to discriminate mutants, but did not automate his approach. Furthermore, our refinement-based theory is a generalization of his Z-based framework. Our theory can be applied to quite different specification models, e.g. to CSP, Label Transition Systems etc., provided a notion of refinement (or a similar implementation relation) is available.

More recently, Simon Burton presented a similar technique as part of his test case generator for Z specifications [4]. He uses a combination of a theorem prover and a collection of constraint solvers. The theorem prover generates the DNF, simplifies the formulas (and helps formulate different testing strategies). This is in contrast to our implementation, where Constraint Handling Rules [14] are doing the simplification prior to the search — only a constraint satisfaction framework is needed. Here, it is worth pointing out that it is the use of Constraint Handling Rules that saves us from having several constraint solvers, like Burton does. As with Stocks' work, Burton's conditions for fault-based testing are instantiations of our general theory.

Wimmel and Jürjens [15] use mutation testing on specifications to extract those interaction sequences that are most likely to find security issues. This work is closest to ours, since they generate test cases for finding faults.

5.2 Future Work

We see that fault injection on the specification level gets more and more attention. An open research question is the choice of useful mutation operators. An idea we are currently investigating is the selection of mutation operators that represent common semantic misunderstandings of a modeling language. A famous example are the various semantics of state transition diagrams. By means of such fault-based test cases one could prevent the semantic misinterpretations to be implemented.

Overture. We advocate to put fault-based test case generation on the Overture project's agenda. VDM++ is a suitable candidate for the presented testing technique. The fault-based testing approach fits very well into the model-driven design approach with a strong focus on validation. As we explained in the beginning, test cases play a crucial role in validating the model. Fault injection adds a further dimension to this validation process, since it introduces a fault analysis into the design process, very similar to what is done in safety analysis. The generated test cases form the fault prevention mechanism. Furthermore, the equivalent mutant analysis helps to reduce bias in the models.

In interesting aspect of the VDM++ language is its ability to model computational as well as behavioural aspects of a system. Hence, faults injection, too, can be applied to both dimensions of a model. In this paper we have only discussed fault injection into computational (pre-postcondition) models. However, mutating the behavioural protocol of a VDM++ component would lead to an integration testing strategy. Fault-based integration testing is a research direction we are currently working on.

The Eclipse tool plug-ins we are envisioning would include a set of specialised VDM++ constraint or SAT solvers, a graphical interface where a tester is able to select and mutate parts of the specification, a mutation operator definition interface, and the actual test execution environment.

References

1. Satyajit Acharya and Chris George. Specifying a mobile computing application environment using rsl. Technical Report 300, United Nations University, International Institute for Software Technology (UNU-IIST), 2004.
2. Bernhard K. Aichernig, Andreas Gerstinger, and Robert Aster. Formal specification techniques as a catalyst in validation. In *Proceedings of the 5th IEEE High Assurance Systems Engineering Symposium (HASE 2000), November 15–17, Albuquerque, New Mexico*, pages 203–207. IEEE, 2000.
3. Bernhard K. Aichernig and Jifeng He. Testing for design faults. *Formal Aspects of Computing Journal*, 2005. Under consideration for publication. Available at `http://www.iist.unu.edu/~bka/publications/`.
4. Simon Burton. Automated Testing from Z Specifications. Technical Report YCS 329, Department of Computer Science, University of York, 2000.
5. R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.
6. Chris George. RAISE tool user guide. Technical report, United Nations University, International Institute for Software Technology (UNU-IIST), 2004.
7. Richard G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.
8. Johann Hörl and Bernhard K. Aichernig. Validating voice communication requirements using lightweight formal methods. *IEEE Software*, pages 21–27, May/June 2000.
9. Percy Antonio Pari Salas and Bernhard K. Aichernig. Automatic Test Case Generation for OCL: a Mutation Approach. Technical Report 321, United Nations University, International Institute for Software Technology (UNU-IIST), 2005.
10. Thitima Srivatanakul, John Clark, Susan Stepney, and Fiona Polack. Challenging formal specifications by mutation: a CSP security example. In *Proceedings of APSEC-2003: 10th Asia-Pacific Software Engineering Conference, Chiang Mai, Thailand, December, 2003*, pages 340–351. IEEE, 2003.
11. Philip Alan Stocks. *Applying formal methods to software testing.* PhD thesis, Department of computer science, University of Queensland, 1993.
12. K.-C. Tai. Theory of fault-based predicate testing for computer programs. *IEEE Transactions on Software Engineering*, 22(8):552–562, 1996.
13. K.-C. Tai and H.-K. Su. Test generation for Boolean expressions. In *Proceedings of the Eleventh Annual International Computer Software and Applications Conference (COMPSAC)*, pages 278–284, 1987.

14. Thom Frühwirth. Theory and Practice of Constraint Handling Rules. *The Journal of Logic Programming*, 1994.
15. Guido Wimmel and Jan Jürjens. Specification-based test generation for security-critical systems using mutations. In Chris George and Miao Huaikou, editors, *Proceedings of ICFEM'02, the International Conference of Formal Engineering Methods, October 21–25, 2002, Shanghai, China*, Lecture Notes in Computer Science. Springer-Verlag, 2002.