Test Case Generation by OCL Mutation and Constraint Solving

Bernhard K. Aichernig Percy Antonio Pari Salas
International Institute for Software Technology
United Nations University (UNU-IIST),
P.O.Box 3058, Macau S.A.R. China
{bka,percy}@iist.unu.edu

Abstract

Fault-based testing is a technique where testers anticipate errors in a system under test in order to assess or generate test cases. The idea is to have enough test cases capable of detecting these anticipated errors. This paper presents a method of fault-based test case generation for pre- and postcondition specifications. Here, errors are anticipated on the specification level by mutating the pre- and postconditions. We present the underlying theory by giving test cases a formal semantics and translate this general testing theory to a constraint satisfaction problem. A prototype test case generator serves to demonstrate the automatization of the method. The current tool works with OCL specifications, but the theory and method are general and apply to many state-based specification languages.

1 Introduction

Testing is a highly controversial subject with respect to verification. The obvious reason is that, *in general*, it cannot demonstrate the absence of faults. This can only be achieved by (formal) correctness proofs. However, in practice testing is still the most popular verification technique. Taking into account the obvious need of practitioners, Formal Methods have included testing into their realm over the last decade. This research placed testing on a solid foundation leading to a series of testing methods and tools that work on the specification level.

Basically, what has happened is to give up the *in general* and clarify under which circumstances (hypothesis) testing can serve as a verification vehicle in the scientific sense. For example, assuming an equivalence relation (possibly derived from a formal specification), testing of representative input values can be defended. Another example, is model-checking, where the concentration on finite domains and abstraction in general, has led to successful exhaustive testing techniques.

In the current paper yet another test hypothesis will be exploited: We assume that we can anticipate the possible errors made by an implementer. This technique is generally known as *fault-based testing* and its most prominent form is *mutation testing*. Originally in mutation testing, a program text has been altered (mutated) in order to check if a given set of test cases is able to detect this change. In our work, we advance this technique in two directions: (1) we mutate specifications, not programs, in order to model errors that may happen during the whole development process; (2) we do not only analyze the coverage of given test cases, but generate test cases that will cover the introduced faults. The present paper will show that our technique challenges Dijkstra's famous sentence about testing, since we claim: *Testing can show the absence of faults, if we have a knowledge of what can go wrong.* Ideas of what can go wrong, can be systematically derived from formal models of an implementation. By introducing possible faults, test cases can be generated that will ensure that this kind of fault does not survive.

Consider the well-known Triangle example, specified in Fig. 1. 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. 2. 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).

```
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
```

Figure 1. Specification of a triangle in OCL.

Proceedings of the Fifth International Conference on Quality Software (QSIC'05)
1550-6002/05 $20.00 © 2005 IEEE

```
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
```

Figure 2. Two mutated specifications for the triangle example.

The first author has developed a general theory of fault-based testing, in which it can be formally proven that test cases will detect certain faults. This theory is based on the general concept of refinement (implementation relation) and can be instantiated to different semantic frameworks where this concept exists. The present paper shows an instantiation to pre-postcondition specifications in first-order logic. A tool has been developed that works on OCL specifications, giving them a design semantics as used in the Unifying Theories of Programming [15]. The tool builds on three pillars: (1) the already mentioned fault-based testing theory of the first author [4] to state the problem; (2) Dick and Faivre's classical work on specification-based testing [12] to divide the problem; and (3) on Constraint Logic Programming [20] to search for solutions. It should be emphasized that none of these three is special to OCL and that, therefore, the presented technique could be equally applied to any model-oriented specification language, including B, JML, RAISE, VDM-SL, and Z.

In the following, we describe our approach in detail. Section 2 briefly introduces the reader to the theory of designs of [15] and to constraint solving. Section 3 links concepts on testing with the theory of designs and presents an algorithm that generates test cases that will find anticipated errors in a design. Some results of the use of a prototype are presented in Section 4. Section 5 discusses related work. Finally, Section 6 presents our conclusions and some general directions for future work. Further details can be found in [17].

2 Preliminaries

Before presenting the testing theory, it is necessary to clarify the terms used throughout the paper and give the necessary definitions. In addition, a brief introduction to constraint solving is given.

2.1 Testing from Specifications

The vocabulary of computer scientists is rich with terms for naming the unwanted: bug, error, defect, fault, failure, etc. are commonly used. Here, we adopt the standard terminology as recommended by the IEEE Computer Society:

Definition 2.1 *An* **error** *is made by somebody. A good synonym is mistake. When people make mistakes during coding, we call these mistakes bugs. A* **fault** *is a representation of an error. As such it is the result of an error. A* **failure** *is a wrong behavior caused by a fault. A failure can occur when a fault executes.*

In this work we aim to generate test-cases on the basis of possible errors during the design of software. Examples of such errors might be a missing or misunderstood requirement, a wrongly implemented requirement, or simple coding errors. In order to represent these errors we will introduce faults into formal specifications. The faults will be introduced by deliberately changing a design, resulting in wrong behavior possibly causing a failure.

In this work we restrict ourselves to model-based (model-oriented) specifications. More precisely, we use the design calculus of the Unifying Theory of Programming to assign specifications a precise semantics. Designs are a special form of predicates with a pre- and postcondition part, together with an alphabet. The alphabet is a set of variables that declares the observation space. The free variables of a design predicate are a subset of the alphabet and represent state variables before (undecorated variable names) and after execution (decorated variable names) of a program. In addition, special Boolean variables ok and ok' denote the successful start and termination of a program. Formally, we define

Definition 2.2 (Design) *Let P and Q be predicates not containing ok or ok′.*

$$P \vdash Q \;=_{df}\; (ok \wedge P) \;\Rightarrow\; (ok' \wedge Q)$$

A design is a relation whose predicate is (or could be) expressed in this form.

Implication establishes a refinement order (actually a lattice) over designs. Thus, more concrete implementations imply more abstract specifications.

Definition 2.3 (Refinement)

$$D_1 \sqsubseteq D_2 \;\; =_{df} \;\; \forall v, w, \cdots \in A \bullet D_2 \;\Rightarrow\; D_1 \,,$$

for all D_1, D_2 with alphabet A.

Alternatively, using square brackets to denote universal quantification over all variables in the alphabet, we write $[D_2 \;\Rightarrow\; D_1]$. Obviously, this gives the well-known properties that preconditions are weakened under refinement and postconditions are strengthened (become more deterministic):

Theorem 2.1 (Refinement of Designs)

$$[(P_1 \vdash Q_1) \;\Rightarrow\; (P_2 \vdash Q_2)] \quad \textbf{iff}$$

$$[P_2 \;\Rightarrow\; P_1] \;\; \textit{and} \;\; [(P_2 \wedge Q_1) \;\Rightarrow\; Q_2]$$

For a definition of common operators on designs, like sequential composition, conditional, choice, etc., we refer to [15]. According to Definition 2.1, *faults* represent errors. These errors can be introduced during the whole development process in all artifacts created. Consequently, faults appear on different levels of abstraction in the refinement hierarchy ranging from requirements to implementations. Obviously, early introduced faults are the most dangerous (and most expensive) ones, since they may go undetected into an implementation; or formally, a faulty design may be correctly refined into an implementation. Again refinement is the central notion in order to discuss the roles and consequences of certain faults and design predicates that are most suitable for representing faults.

Definition 2.4 (Design Fault) *Given an intended design D, and an unintended design D' in which during creation an* error *was made. Then, we define a* fault *in design D' as the syntactical deviation* from *D in D', if and only if refinement does not hold:*

$$D \not\sqsubseteq D'$$

We call D' a faulty design.

Consequently, not all errors lead to faults. Here, for being a fault, a possible (external) observation of this fault must exist. For example, adding by mistake redundant constraints to a design does not result in a faulty design (since refinement holds). However, changing the alphabet leads to a faulty design. (detected during type checking). Note also, the use of the term *intended* (*unintended*) in the definition, instead of *correct* (*incorrect*). This is necessary, since the latter is only defined with respect to a given specification, but faults can already be present in such specifications from the very beginning.

2.2 Constraint Solving

The problem of generating test cases from a formal specification can be represented as a Constraint Satisfaction Problem (CSP). A constraint satisfaction problem consists of a finite set of variables and a set of constraints. Each variable is associated with a set of possible values, known as its domain. A constraint is a relation defined on some subset of these variables and denotes valid combinations of their values. A solution to a constraint satisfaction problem is an assignment of a value to each variable from its domain, such that all the constraints are satisfied. Formally, the conjunction of these constraints forms a predicate for which a solution should be found. Since, we are interested in special solutions for input-output relations, we define a solution as follows:

Definition 2.5 (Solution) *Let $P(v, v')$ be a predicate, with v and v' being lists of unprimed and primed free variables. Given a pair (x, X'), with x being a list of values of same length as v and X' being a finite set of value lists with the same length as v'. Then,*

$$(x, X') \textbf{ \textit{is solution of} } P \quad \textbf{iff} \quad \forall\, x' \in X' \cdot P(x, x')$$

We call (x, X') a maximal solution, if X' is maximal.

A constraint satisfaction problem is always embedded into a Constraint System. A Constraint System formally specifies the syntax and semantics of the constraints of interest. It is defined [20] as a *tuple* $(\Sigma, \mathcal{D}, \mathcal{CT}, \mathcal{C})$ where

- Σ is a signature that contains all possible constraint and function symbols,
- $\mathcal{D}$ is a domain together with an interpretation of the symbols in Σ,
- $\mathcal{CT}$ is a non-empty and consistent theory over Σ,
- $\mathcal{C}$ are the allowed constraints.

A constraint solver implements an algorithm for solving well-formed constraints within a CSP in accordance with a constraint theory. The constraint theory $\mathcal{CT}$ defines the semantics of a constraint system and is composed, in our case, of a set of transformation rules over one or more specific domains. The syntax of well-formed constraints is defined by the set of allowed constraints $\mathcal{C}$.

Our approach is then to embed the test generation problem modelled as a CSP into a specially designed and implemented Constraint System. But this is not a novelty because this approach has been widely explored and implemented. The novelty in our approach is that we derive the constraints to be solved from the more general theory of refinement, or more precisely non-refinement. The constraints presented

in the next section are the conditions when refinement of a mutant does not hold. These conditions form an equivalence class of fault-adequate test cases in which every test case is able to find the associated fault.

3 Fault-based Test Case Generation

3.1 Test Cases

We take the point of view that test cases are specifications that for a given input define the expected output. Consequently, we define test cases as a sub-theory of designs.

Definition 3.1 (Test Case, deterministic) *Let i be the input vector and o be the expected output vector, both being lists of values, having the same length as the variable lists v and v' respectively. Furthermore, equality over value lists should be defined.*

$$t_d(i,o) \;=_{df}\; v = i \vdash v' = o$$

Although sufficient for deterministic programs, test cases derived from a specification have to take non-determinism into account. Therefore, we generalize the notion of a test case as follows:

Definition 3.2 (Test Case, general) *Let i be the* input vector *and O a possibly infinite set containing the expected* output vector(s). *Both, i and o, $o \in O$, are lists of values, having the same length as the value lists v and v' respectively.*

$$t(i,O) \;=_{df}\; v = i \vdash v' \in O$$

Previous work of the first author [2] has shown that refinement is the key to understand the relation between test cases, specifications and implementations. Refinement is an observational order relation, usually used for step-wise development from specifications to implementations, as well as to support substitution of software components. Since we view test cases as (a special form of) specification, it is obvious that a correct implementation should refine its test cases. Thus, test cases are abstractions of an implementation, if and only if the implementation passes the test cases. This view can be lifted to the specification level. When test cases are properly derived from a specification, then these test cases should be abstractions of the specification. Formally, we define:

Definition 3.3 *Let T be a set of test cases, S a specification and I an implementation, all being designs, and*

$$T \sqsubseteq S \sqsubseteq I$$

we define

- *T as a correct test set with respect to S,*
- *implementation I passes the test cases in T,*
- *implementation I conforms to specification S.*

Finding a test case t that detects a given fault is the central strategy in fault-based testing. For example, in classical mutation testing, D is a program and D' a mutant of D. Then, if the mutation in D' represents a fault, a test case t should be included to detect the fault. Consequently, we can define a fault-based test case as follows:

Definition 3.4 (Fault-adequate Test Case) *Let t be an input-output test case (possibly non-deterministic). Furthermore, D is a design and D' its faulty version. Then, t is a **fault-adequate test case** when*

$$t \sqsubseteq D \quad \wedge \quad (t \not\sqsubseteq D')$$

We say that a fault-adequate test case detects *the fault in D'. Alternatively we can say that the test case* distinguishes *D and D'. In the context of mutation testing, one says that t* kills *the mutant D'. All the test cases that detect a certain fault form a* fault-adequate equivalence class.

Note that our definitions solely rely on the lattice properties of designs. Therefore, our fault-based testing strategy scales up to other lattice-based test models as long as an appropriate refinement definition is used.

3.2 Test Case Generation Algorithm

From the definition of *fault-adequate equivalence class* we have developed an algorithm to generate a test case which is inside this equivalence class.

Algorithm 1 *Given a design $D(Pre \vdash Post)$ 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

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

The algorithm is designed to produce test cases such that the original D refines it (T is a positive test case). Therefore, Step 1 involves two phases: finding a solution i_c, o_c and finding the test case $t(i,O)$. In these step, the solution (i_c, o_c) represents a counterexample of $D \sqsubseteq D'$, with o_c being one output (from a possible non-empty set of outputs) that differ from the original. A test case $t(i_c, o_c)$ would be a negative test case reporting when a test fails. However, since we are interested in positive test cases that predict all the possible outputs according to the original specification, T includes all outputs O satisfying the postcondition. This also prevents T from reporting false negatives.

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 two steps, then the original and the mutant specifications are equivalent (in the context of the finite domains of the CSP).

Next, we show that the test case generated by the algorithm is fault-adequate. Thus, we have to show that the test case is correct with respect to D as well as it covers the fault in D'. We split this analysis into two theorems.

Theorem 3.1 (Test Case Correctness) *Given a design $D(Pre \vdash Post)$ and its faulty design $D'(Pre' \vdash Post')$, and a test case $t(i,O)$ generated by Algorithm 1*

$$t(i,O) \sqsubseteq D$$

Proof. *The proof can be split into two cases reflecting Steps 1 & 2 of the algorithm:*

1. *From (i,O) **is solution of** $Pre \wedge Post \wedge (v = i_c)$ it follows that (i,O) **is solution of** $Pre \wedge Post$. Thus,*

$$\begin{array}{ll} & t(i,O) \sqsubseteq D \\ = & \qquad \{\textit{by Theorem 2.1}\} \\ & [(v = i) \Rightarrow Pre] \wedge \\ & [(v = i) \wedge Post \Rightarrow v' \in O] \\ = & \quad \{\textit{since } (i,O) \textbf{ \textit{is solution of}} \ (Pre \wedge Post) \\ & \qquad \textit{and } O \textit{ is maximal}\} \\ & true \wedge true \\ = & true \end{array}$$

2. *follows the same style of reasoning as the first case.*

Theorem 3.2 (Fault coverage) *Given a design $D(Pre \vdash Post)$ and its faulty design $D'(Pre' \vdash Post')$, and a test case $t(i,O)$ generated by Algorithm 1*

$$t(i,O) \not\sqsubseteq D'$$

Proof. *Again, this proof is split into the two cases of the algorithm:*

1. *When (i,O) is solution of $Pre \wedge Post \wedge (v = i_c)$. This test case is only generated, if (i_c, o_c) **is solution of** $(Pre \wedge \neg Post \wedge Post')$ exists. We see immediately that $i = i_c$ and $\{o_c\} \cap O = \{\}$ and deduce*

$$\begin{array}{ll} & t(i,O) \not\sqsubseteq D' \\ = & \qquad \{\textit{by Theorem 2.1}\} \\ & \neg[(v = i) \Rightarrow Pre'] \quad \vee \\ & \neg[((v = i) \wedge Post') \Rightarrow (v' \in O)] \\ = & \exists\, v \bullet ((v = i) \wedge \neg Pre') \quad \vee \\ & \exists\, v, v' \bullet ((v = i) \wedge Post' \wedge (v' \notin O)) \\ = & \quad \{\textit{since solution } (i_c, o_c) \textit{ is a witness for 2nd. } \exists\} \\ & \exists\, v \bullet ((v = i) \wedge \neg Pre') \quad \vee \quad true \\ = & true \end{array}$$

2. *When (i,O) **is solution of** $\neg Pre' \wedge Pre \wedge Post$.*

$$\begin{array}{ll} & t(i,O) \not\sqsubseteq D' \\ = & \qquad \{\textit{by Theorem 2.1}\} \\ & \neg[(v = i) \Rightarrow Pre'] \quad \vee \\ & \neg[((v = i) \wedge Post') \Rightarrow (v' \in O) \\ = & \exists\, v \bullet ((v = i) \wedge \neg Pre') \quad \vee \\ & \exists\, v, v' \bullet ((v = i) \wedge Post' \wedge (v' \notin O)) \\ = & \qquad \{\textit{since i is a witness to 1st. } \exists\} \\ & true \quad \vee \\ & \exists\, v, v' \bullet ((v = i) \wedge Post' \wedge (v' \notin O)) \\ = & true \end{array}$$

It is important to point out that fault-adequacy does not necessarily mean that a program implementing the faulty design will be rejected. The reason is non-determinism. Consider, e.g. $D = (x' > 1)$, $D' = (x' > 0)$ and an implementation $P_1 = (x' = 3)$. Here, $D \not\sqsubseteq D'$ and P_1 refines both D and D'. Obviously, the fault-adequate test case $T = t(1, \{2,3,4,\ldots\})$ cannot detect any fault (because there is not one). We can think of P_1 as an implementation that started from a wrong specification D', but during implementation the error got corrected (by chance). The idea of having T is to prevent cases where the implementer was not so lucky, like with $P_2 = (x' = 1)$.

4 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. 1.

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. 3. Here, for every equivalence class (domain partition) one test case is chosen. The strategy is to cover each partition.

In contrast to the DNF stategy, the fault-based algorithm generated test cases that cover faults. Generating the fault-based test cases for the two mutant OCL specifications in Fig. 2 results exactly in the two test cases

```
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
```

Figure 3. DNF-based test cases.

```
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
```

Figure 4. Two further mutations.

$a=1,b=2,c=2,result=$"$isosceles''$ and $a=1,b=1,c=1,result=$"$equilateral''$ already presented in Section 1.

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. 3 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. 4 below. One can easily see that the DNF test cases in Fig. 3 are not able to reveal these faults.

However, the fault-based algorithm generates precisely those test cases needed to unreveal the faults: $a=2,b=2,c=2,result=$"$equilateral''$ for the first one, and $a=3,b=2,c=4,result=$"$scalene''$ 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 $a=1,b=3,c=3,result=$"$isosceles''$ 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. Finally, it should be added that the test case generator also helps in understanding the specification. Experimenting with different mutations and generating fault-adequate test cases for them is a valuable vehicle for validation.

For a larger example on testing the security policy of a database management system see [17].

5 Related Work

Nowadays, test case generation from software specifications has become a popular area of research and it is out of the scope of this paper to provide a complete overview on the progress made. Perhaps the most prominent contribution leading to a gradual reconciliation between the areas of software testing and formal methods came from Gaudel [14]. The most relevant papers to our work include [12, 7, 16, 5, 11]. In those papers, several techniques have been applied with a strong predominance of state machine based techniques. Although different, most of these techniques have one point in common, the use of partition analysis and Disjunctive Normal Form.

Another related work [1] presents a prototype tool called UML-CASTING. The approach implemented by this tool also uses constraint solving for instantiating values for a particular test case. Furthermore, it uses a similar input model to our tool. It supports test case sequencing using information extracted from UML state diagrams. However, its input model is more restricted than ours in terms of the constraint language itself and does not include the mutation approach.

Software testing in general, and mutation testing in particular, have been introduced in the theory of Refinement Calculus [2] and in the Unified Theory of Programming [4]. The theoretical part of the presented work is a direct result of these testing theories.

Others have worked on mutation testing on the specification level. To our present knowledge Budd and Gopal were the first [8]. They applied a set of mutation operators to specifications given in predicate calculus form. The method relies on having a working implementation generating output.

Tai and Su [22] 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 [21] 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.

Stocks applied mutation testing to Z specifications [19]. 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.

Woodward investigated mutation operators for algebraic specifications [24].

More recently, Simon Burton presented a similar technique as part of his test case generator for Z specifications [9]. 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 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.

An alternative tool to a constraint solver is a model checker. Black et al. studied mutation operators using the SMV model checker [6]. Recently, our group applied the CADP model checker to generate tests for testing webservers [3].

A group in York has recently started to use fault-based techniques for validating their CSP models [18]. Their aim is not to generate test cases, but to study the equivalent mutants. Similar research is going on in Brazil with an emphasis on protocol specifications written in the Estelle language [10].

Wimmel and Jürjens [23] 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.

6 Conclusions

In this paper a mutation testing approach for automatic test case generation from model-based specifications has been presented. Its general idea is to model faults on the specification level by altering a given specification. Then, test cases are generated that will discover such faults. Starting from a general theory that is based on specification refinement, we developed an algorithm for finding such test cases for pre- and postcondition specifications. The general algorithm has been instantiated in a test case generator for the OCL specification language. This tool combines partition analysis and constraint solving.

It is worth pointing out that both, the theory and the algorithm are general and not linked to a certain specification language. Even the core of the constraint solver is largely independent of OCL and could be fairly easily adapted to languages such as RAISE, VDM-SL, Z, B or JML.

The next steps of research involve case studies in order to gain more experience with our testing approach. We are in particular interested if it is possible to identify classes of useful mutation operators. These operators will rather define patterns of alternation than strict syntactic mappings (as in program mutation). Many are expected to be domain or application specific. Furthermore, it will be interesting to explore the links to safety analysis, since it might provide a systematic way for identifying relevant mutations. Obviously, security is another application area.

The future will show if the fault-based testing strategy will play a major role in testing. We believe so, because it supports the tester in concentrating on possible issues in a more direct way than the traditional structural testing methods do.

Acknowledgement. Our constraint solver was implemented using JCK [13] from Ludwig Maximilians University of Munich. The OCL parser was implemented with CUP[1] from Scott E. Hudson. We acknowledge Chris George for his contribution on reviewing the draft of this work.

References

[1] L. Van Aertryck and T. Jensen. UML-CASTING: Test Synthesis from UML Models using Constraint Resolution. In *Proc.AFADL'2003 (Approches Formelles dans l'Assistance au Dévelopment de Logiciel)*, January 2003.

[2] B.K. Aichernig. Mutation Testing in the Refinement Calculus. *Formal Aspects of Computing*, 15(2-3):280–295, November 2003.

[3] B.K. Aichernig and C. Corrales Delgado. Test purpose generation by specification mutation in distributed systems. Technical Report 313, United Nations University, International Institute for Software Technology (UNU-IIST), 2004.

[4] B.K. Aichernig and J. He. Testing for Design Faults, 2005. Under consideration for publication in Formal Aspects of Computing.

[5] M. Benattou, J.M. Bruel, and N. Hameurlain. Generating Test Data from OCL Specification. In

[1] http://www.cs.princeton.edu/ appel/modern/java/CUP/

ECOOP'2002 Workshop on Integration and Transformation of UML Models (WITUML02), June 2002.

[6] P.E. Black, V. Okun, and Y. Yesha. Mutation of model checker specifications for test generation and evaluation. pages 14–20, 2001.

[7] F. Bouquet, B. Legeard, and F. Peureux. CLPS-B - A Constraint Solver for B. In *Proceedings of the conference of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, pages 188–204, Grenoble, France, April 2002.

[8] T.A. Budd and A.S. Gopal. Program testing by specification mutation. *Comput. Lang.*, 10(1):63–73, 1985.

[9] S. Burton. Automated Testing from Z Specifications. Technical Report YCS 329, Department of Computer Science, University of York, 2000.

[10] S.dR.S. de Souza, J.C. Maldonadoand S.C.P.F. Fabbri, and W.L. de Souza. Mutation testing applied to Estelle specifications. *Software Quality Journal*, 8(4):285–301, 1999.

[11] J. Derrick and E.A. Boiten. Testing refinements of state-based formal specifications. *Software Testing, Verification and Reliability*, 9(1):27–50, July 1999.

[12] J. Dick and A. Faivre. Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In J. C. P. Woodcock and P. G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, number 670 in LNCS, pages 268–284. Springer-Verlag, 1993.

[13] E. Krämer. A Generic Search Engine for a Java Constraint Kit. Diploma thesis, Ludwig Maximilians Universität München, January 2001.

[14] M.C. Gaudel. Testing can be formal, too. In *TAPSOFT '95: Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 82–96. Springer-Verlag, 1995.

[15] C.A.R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall International, 1998.

[16] B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from Z and B. In *Proc. of the Int. Conf. on Formal Methods Europe, FME'02*, volume 2391 of *LNCS*, pages 21–40. Springer, July 2002.

[17] P.A. Pari Salas and B.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.

[18] T. Srivatanakul, J. Clark, S. Stepney, and F. 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.

[19] P.A. Stocks. *Applying formal methods to software testing*. PhD thesis, Department of computer science, University of Queensland, 1993.

[20] T. Frühwirth and Slim Abdennadher. *Essentials of Constraint Programming*. Springer-Verlag, 2003.

[21] K.-C. Tai. Theory of fault-based predicate testing for computer programs. *IEEE Transactions on Software Engineering*, 22(8):552–562, 1996.

[22] 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.

[23] G. Wimmel and J. Jürjens. Specification-based test generation for security-critical systems using mutations. In C. George and M. Huaikou, editors, *Proc. of ICFEM'02, the International Conference of Formal Engineering Methods, October 21–25, 2002, Shanghai, China*, LNCS. Springer-Verlag, 2002.

[24] M. Woodward. Errors in algebraic specifications and an experimental mutation testing tool. *Software Engineering Journal*, pages 211–224, July 1993.