Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 187 (2007) 125–143
www.elsevier.com/locate/entcs

Refinement and Test Case Generation in UTP

Bernhard K. Aichernig[1]

Institute for Software Technology
Graz University of Technology
Graz, Austria

Jifeng He[2]

East China Normal University
Shanghai, China

Abstract

This paper presents a theory of testing that integrates into Hoare and He's Unifying Theory of Programming (UTP). We give test cases a denotational semantics by viewing them as specification predicates. This reformulation of test cases allows for relating test cases via refinement to specifications and programs. Having such a refinement order that integrates test cases, we develop a testing theory for fault-based testing.
Fault-based testing uses test data designed to demonstrate the absence of a set of pre-specified faults. A well-known fault-based technique is mutation testing. In mutation testing, first, faults are injected into a program by altering (mutating) its source code. Then, test cases that can detect these errors are designed. The assumption is that other faults will be caught, too. In this paper, we apply the mutation technique to both, specifications and programs.
Using our theory of testing, two new test case generation laws for detecting injected (anticipated) faults are presented: one is based on the semantic level of design specifications, the other on the algebraic properties of a programming language.

Keywords: formal methods; specification-based, model-based, fault-based testing; mutation testing; refinement; Unifying Theories of Programming; algebra of programming.

1 Introduction

A theory of programming explores the principles that underlie the successful practice of software engineering. Consequently, a theory of programming should not lack a theory of testing. Understanding of the fundamentals of software testing enables the experience gained in one language or application domain to be generalised rapidly to new applications and to new developments in technology. It is the contribution

[1] Email: aichernig@ist.tugraz.at
[2] Email: jifeng@sei.ecnu.edu.cn

doi:10.1016/j.entcs.2006.08.048

of this paper to add a theory of testing to Hoare & He's Unifying Theories of Programming (UTP) [11].

The theory we contribute was designed to be a complement to the existing body of knowledge. Traditionally, theories of programming focus on semantical issues, like correctness, refinement and the algebraic properties of a programming language. A complementary testing theory should focus on the dual concept of *fault*. The main idea of a fault-centred testing approach, also called *fault-based testing*, is to design test data to demonstrate the absence of a set of pre-specified faults.

It is this fundamentally different philosophy of our fault-based testing theory that adds a further dimension to the theories of programming. Rather than doing verification by testing, a doubtful endeavour anyway, here we focus on falsification. It is falsification, because the tester gains confidence in a system by designing test cases that would uncover an anticipated error. If the falsification fails, it follows that a certain fault does not exist. The fascinating point is that program refinement plays a key role in our theory of testing. However, due to the concentration on faults we are interested in the cases, where refinement does not hold — again falsification rather than verification.

The interesting questions that arise from focusing on faults are: Does an error made by a designer or programmer lead to an observable failure? Do my test cases detect such faults? How do I find a test case that uncovers a certain fault? What are the equivalent test cases that would uncover such a fault? Finally and most important: How to automatically generate test cases that will reveal certain faults? All these questions are addressed in this paper. They have been addressed before, but rarely on a systematic and scientifically defendable basis.

We assume some familiarity of the reader with UTP's theory of designs [11]. All theorems in this paper have been formally proven. The proofs will appear in an extended journal version of this paper.

The paper is structured as follows. After this general introduction, Section 2 gives a very brief introduction to the theory of designs of [11] and defines what we mean by a faulty design. The next two sections include the main contributions of this paper. Section 3 contains a construction for test cases that will find anticipated errors in a design. This test case generation technique works on the semantic level of designs. In Section 4, a purely algebraic (syntax-oriented) test case generation technique is presented. It is based on the algebraic properties of a small, but nontrivial, programming language. Finally, in Section 5 we discuss the results as well as its related work, and present an outlook on future research directions.

2 Preliminaries

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 behaviour 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 behaviour possibly causing a failure.

Here, we restrict ourselves to model-based (model-oriented) specifications. More precisely, we use the design calculus of UTP 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'.

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

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

Every program can be expressed as a design. This makes the theory of designs a tool for expressing specifications, programs, and, as it will be shown, test cases. In the following, some basic programming constructs are presented.

Definition 2.3 Assignment Given a program variable x and an expression e

$$x \;:=\; e \quad =_{df} \quad (wf(e) \vdash x' = e \wedge y' = y \wedge \cdots \wedge z' = z)$$

with wf being the predicate defining the well-formedness of expression e (e can be evaluated).

Definition 2.4 Conditional

$$P \lhd b \rhd Q \quad =_{df} \quad (wf(b) \vdash (b \wedge P \vee \neg b \wedge Q))$$

with wf being the predicate defining the well-formedness of the Boolean expression b.

In the further discussion we will maintain the simplifying assumption that all program expressions are everywhere well-formed (defined), thus $wf = \mathbf{true}$.

Sequential composition is defined in the obvious way, via the existence of an intermediate state v_0 of variables v. Here the existential quantification hides the intermediate observation v_0. In addition, the output alphabet of P ($out\alpha P$) and the input alphabet of Q (with all variables dashed, $in\alpha' Q$) must be identical.

Definition 2.5 Sequential Composition

$$P(v');Q(v) \quad =_{df} \quad \exists v_0 \bullet P(v_0) \wedge Q(v_0)\,, \qquad \text{provided } out\alpha P = in\alpha' Q = \{v'\}$$

Non-deterministic, demonic choice is defined as logical or:

Definition 2.6 (Demonic Choice)

$$P \sqcap Q \quad =_{df} \quad P \vee Q$$

UTP provides a series of theorems and lemmas expressing the basic algebraic properties of such programming constructs (see [11]).

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

Definition 2.7 (Refinement)

$$D_1 \;\sqsubseteq\; D_2 \quad =_{df} \quad \forall v, w, \cdots \in A \;\bullet\; D_2 \;\Rightarrow\; D_1\,, \text{ for all } D_1, D_2 \text{ with alphabet } A.$$

Alternatively, using square brackets to denote universal quantification over all variables in the alphabet, we write $[D_2 \;\Rightarrow\; D_1]$, or simply in refinement calculus style $D_1 \sqsubseteq D_2$.

Obviously, this gives the well-known properties that preconditions are weakened under refinement and postconditions are strengthened (become more deterministic):

Theorem 2.8 (Refinement of Designs)

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

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. 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 2.9 (Design Fault) Given a (intended) design D, and a (unintended) design D^m during which creation an *error* had been made (m stands for mutation). Then, we define a *design fault* in design D^m as the syntactical difference to D, if

$$D \;\not\sqsubseteq\; D^m$$

(or $\neg(D \sqsubseteq D^m)$). We call D^m a *faulty design* or a *faulty mutation* of D.

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 conditions 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.

3 Testing for Design Faults

In this section, we relate test cases via refinement to designs and programs. This is possible, since we give test cases a denotational semantics by viewing them as specification predicates. The result is a test case generation technique based on non-refinement.

3.1 Test Cases as Designs

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.

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

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

Definition 3.2 (Test Case, general)

$$t_\sqcap(i, c) \quad =_{df} \quad v = i \vdash c(v')$$

where c is a condition on the after state space defining the *possibly infinite* set of expected outcome vectors.

Previous work of the first author [1] 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

$$t \sqsubseteq S \sqsubseteq I, \qquad \text{for all } t \in T$$

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^m a mutant of D. Then, if the mutation in D^m 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-detecting Test Case) Let t be an input-output test case (possibly non-deterministic). Furthermore, D is a design and D^m its faulty version. Then, t is a fault-detecting test case when

$$t \sqsubseteq D \quad \wedge \quad (t \not\sqsubseteq D^m)$$

We say that t *detects* the fault in D^m. Alternatively we can say that the test case *distinguishes* D and D^m. In the language of mutation testing, t *kills* the mutant D^m. All the test cases that detect a certain fault form a *fault-detecting 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 Fault-detecting Equivalence Classes

A common technique in test case generation is *equivalence class testing* — the partitioning of the input domain (or output range) into equivalence classes. The motivation is the reduction of test cases, by identifying equivalently behaving sets of inputs. The rationale behind this strategy is a uniformity hypothesis assuming an equivalence relation over the behaviour of a program.

A well-known equivalence class testing technique regarding formal specification is *DNF partitioning*: the rewriting of a formal specification into its disjunctive normal form (see e.g. [9]). Usually DNF partitioning is applied to relational specifications, resulting in disjoint partitions of the relations (note that disjointness of the input domain is not guaranteed in DNF partitioning). We call such relational partitions *test equivalence classes.*

Definition 3.5 (Test Equivalence Class) Given a design $D = (p \vdash Q)$, we define a *test equivalence class* $T_\sim$ for testing D as a design of form $T_\sim = d_\perp ; D$ such that $[d \Rightarrow p]$.

The definition makes use of the *assertion* operator $b_\perp \quad =_{df} \quad (v = v') \lhd b \rhd \perp$, leading to a design which has no effect on variables v if the condition holds (skip), and behaves like abort ($\perp = \mathbf{true}$) otherwise.

Note that here a test equivalence class is a design denoting an input-output relation. It is defined via a predicate d that itself represents an equivalence class over input values. Given the definitions above, a design is obviously a refinement of its test equivalence class:

Theorem 3.6 *Given a design $D = p \vdash Q$ and one of its equivalence classes. Then,*

$$T_\sim \sqsubseteq D \qquad \square$$

Obviously, *DNF partitioning* can be applied to design predicates. However, in the following we focus on fault-detecting test equivalence classes. These are test equivalence classes where all test cases are able to detect a certain kind of error.

Definition 3.7 (Representative Test Case) A test case $t = t_\sqcap(i, c)$ is a *representative test case* of a test equivalence class $T_\sim = d_\perp ; D$, with $D = p \vdash Q$, if and only if

$$d(i) \quad \wedge \quad p(i) \quad \wedge \quad [Q(i) \equiv c] \qquad \square$$

This definition ensures that the output condition of a representative test case is not weaker than its test equivalence class specifies.

The following theorem provides an explicit construction of a test equivalence class that represents a set of test cases that are able to detect a particular fault in a design.

Theorem 3.8 (Fault-detecting Equivalence Class) *Given a design $D = p \vdash Q$ and its faulty design $D^m = p^m \vdash Q^m$, thus $D \not\sqsubseteq D^m$. For simplicity, we assume that $Q \equiv (p \Rightarrow Q)$. Then, every representative test case of the test equivalence class*

$$T_\sim \quad =_{df} \quad d_\perp ; D\,, \qquad \textit{with } d = \neg p^m \ \vee \ \exists v' \bullet (Q^m \wedge \neg Q)$$

is able to detect the fault in D^m. $\square$

The rational behind this construction is the fact that, for a test case being able to distinguish a design D from its faulty sibling D^m, refinement between the two must not hold. For designs one may observe two places (cases) where refinement may be violated, the precondition and the postcondition. The domain of $T_\sim$ represents these two classes of test inputs. The first class are test inputs that work for the correct design, but cause the faulty design to abort. The second class are the common input states that produce different output values. The two conditions are derived from negating the refinement law for pre- and postconditions (Theorem 2.8).

We have developed a constraint solver for generating test cases from mutated OCL specifications based on Theorem 3.8 [3].

4 Testing for Program Faults

So far our discussion on testing has focused on the semantical model of designs. In this section, we turn from semantics to syntax. The motivation is to restrict ourselves to a subclass of designs that are expressible, or at least implementable, in a certain programming language. Thus, we define a program as a predicate expressed in the limited notations (syntax) of a programming language. From the predicate semantics of the programming language operators, algebraic laws can be derived (see [11]). In the following, we will use this *algebra of programs* as a means to reason about faults in a program on a purely syntactical basis. The result is a test case generation algorithm for fault-based testing that works solely on the syntax of a programming language. We define the syntax as follows:

$$
\begin{aligned}
\langle \text{program} \rangle ::= \ & \mathbf{true} \\
| \ & \langle \text{variable list} \rangle \ := \ \langle \text{expression list} \rangle \\
| \ & \langle \text{program} \rangle \ \lhd \ \langle \text{Boolean Expression} \rangle \ \rhd \ \langle \text{program} \rangle \\
| \ & \langle \text{program} \rangle \ ; \ \langle \text{program} \rangle \\
| \ & \langle \text{program} \rangle \ \sqcap \ \langle \text{program} \rangle \\
| \ & \langle \text{recursive identifier} \rangle \\
| \ & \mu \ \langle \text{recursive identifier} \rangle \ \bullet \ \langle \text{program} \rangle
\end{aligned}
$$

The semantics of the operators follows the definitions in Section 2. The last recursive statement has the standard least fix-point interpretation.

4.1 Finite Normal Form

Algebraic laws, expressing familiar properties of the operators in the language, can be used to reduce every expression in the restricted notation to an even more restricted notation, called a *normal form*. Normal forms play an essential role in an algebra of programs: They can be used to compare two programs, as well as to give an algebraic semantics to a programming language.

Our idea is to use a normal form to decide if two programs, the original one and the faulty one (also called the mutant) can be distinguished by a test case. When the normal forms of both are equivalent, then the error did not lead to an (observable) fault. This solves the problem of equivalent mutants in mutation testing. Furthermore, the normal form will be used for deriving fault-detecting equivalence classes on a purely algebraic (syntactic) basis. Our normal form has been designed for this purpose: in contrast to the normal form in [11], we push

the conditions outwards. All the laws have been formally proven. The following assignment normal is taken from [11].

Definition 4.1 (Assignment Normal Form) The normal form for assignments is the total assignment, in which all the variables of the program appear on the left hand side in some standard order.

$$x, y, \ldots, z \ := \ e, f, \ldots, g$$

The assignments $v \ := \ g$ or $v \ := \ h(v)$ will be used to express the total assignment; thus the vector variable v is the list of all variables and g, h denote the list of expressions.

A non-total assignment can be transformed to a total assignment by, (1) addition of identity assignments $(a, \ldots \ := \ a, \ldots)$, (2) reordering of the variables with their associated expressions. The law that eliminates sequential composition between normal forms is

$$(v \ := \ g; \ v \ := \ h(v)) \quad = \quad (v \ := \ h(g)) \tag{L1}$$

where $h(g)$ is calculated by substituting the expressions in g for the corresponding variables in v.

Since our language includes non-determinism, we translate conditionals to non-deterministic choices of guarded commands.

$$(P \lhd c \rhd Q) \quad = \quad (c \wedge P) \sqcap (\neg c \wedge Q) \tag{L2}$$

This law follows from the definition of conditional and non-deterministic choice. With this elimination rule at hand we are able to define a non-deterministic normal form.

Definition 4.2 (Non-deterministic Normal Form) A *non-deterministic normal form* is defined to be a non-deterministic choice of guarded total assignments.

$$(g_1 \wedge \ v \ := \ f_1) \ \sqcap \ (g_2 \wedge v \ := \ f_2) \ \sqcap \ \ldots \ \sqcap \ (g_n \wedge v \ := \ f_n)$$

Let A be a set of guarded total assignments, then we write the normal form as $\bigsqcap \ A$.

The previous assignment normal form can be easily expressed in this new normal form as disjunction over the unit set

$$v := g \quad = \quad \bigsqcap\{(\mathbf{true} \wedge v := g)\}$$

The easiest operators to eliminate is choice itself

$$(\bigsqcap A) \sqcap (\bigsqcap B) \quad = \quad \bigsqcap(A \cup B)\} \tag{L3}$$

and the conditional

$$\begin{aligned}(\bigsqcap A) \lhd d \rhd (\bigsqcap B) \quad = \quad & (\bigsqcap\{((d \wedge b) \wedge P) \mid (b \wedge P) \in A\}) \sqcap \\ & (\bigsqcap\{((\neg d \wedge c) \wedge Q) \mid (c \wedge Q) \in B\})\end{aligned} \tag{L4}$$

Sequential composition is reduced by

$$(\bigsqcap A);(\bigsqcap B) \quad = \quad \bigsqcap\{((b \wedge (P;c)) \wedge (P;Q)) \mid (b \wedge P) \in A \ \wedge \ (c \wedge Q) \in B\} \quad \text{(L5)}$$

In order to reduce $P;c$ in law L5 we need an additional law

$$(v := e);b(v) \quad = \quad b(e) \qquad \text{(L6)}$$

The program constant **true** is not an assignment and cannot in general be expressed as a finite disjunction of guarded assignments. Its introduction into the language requires a new normal form.

Definition 4.3 (Non-termination Normal Form) A *Non-termination Normal Form* is a program represented as a disjunction

$$b \ \vee \ P$$

where b is a condition for non-termination and P a non-deterministic normal form.

Any previous normal form P that terminates can be expressed as

$$\textbf{false} \ \vee \ P$$

and the constant **true** as

$$\textbf{true} \ \vee \ v := v$$

The other operators between the new normal forms can be eliminated by the following laws

$$\begin{aligned}
(b \vee P) \sqcap (c \vee Q) &= (b \vee c) \ \vee \ (P \sqcap Q) && \text{(L7)}\\
(b \vee P) \lhd d \rhd (c \vee Q) &= ((b \wedge d) \vee (c \wedge \neg d)) \vee (P \lhd d \rhd Q) && \text{(L8)}\\
(b \vee P);(c \vee Q) &= (b \vee (P;c)) \vee (P;Q) && \text{(L9)}
\end{aligned}$$

The occurrences of each operator on the right hand side can be further reduced by the laws of the previous sections. Again for reducing $(P;c)$ an additional law is needed; this time for the previous non-deterministic normal form.

$$(\bigsqcap A);c \quad = \quad \bigvee\{(g \wedge (P;c)) \mid (g \wedge P) \in A\} \qquad \text{(L10)}$$

The algebraic laws above allow any non-recursive program in our language to be reduced to a finite normal form

$$b \ \vee \ \bigsqcap_i \{(g_i \wedge v := e_i) \mid 1 \leq i \leq n\}$$

4.2 Refinement Laws for the Normal Forms

In the previous section, it was shown that refinement (or non-refinement) is at the heart of our testing theory. As for general designs, the equivalence class for detecting

faults in normal form programs is inspired by their refinement laws. Next, we present these refinement laws for our program normal form.

For assignments that are deterministic, the question of refinement becomes a simple question of equality. Two assignment normal forms are equal, if and only if all the expressions in the total assignment are equal.

$$(v \;:=\; g) \quad = \quad (v := h) \qquad \textbf{iff} \qquad [g = h] \tag{L11}$$

The laws which permit detection of refining mutants for the non-deterministic normal form are:

$$R \;\sqsubseteq\; (\bigsqcap A) \quad \textbf{iff} \quad \forall P \;:\; P \in A \bullet (R \;\sqsubseteq\; P) \tag{L12}$$

$$((g_1 \wedge P_1) \sqcap \ldots \sqcap (g_n \wedge P_n)) \;\sqsubseteq\; (b \wedge Q) \quad \textbf{iff} \quad [\exists i \bullet ((g_i \wedge P_i) \Leftarrow (b \wedge Q))] \tag{L13}$$

$$[(g \wedge v := f) \Leftarrow (b \wedge v := h)] \quad \textbf{iff} \quad [b \;\Rightarrow\; (g \wedge (f = h))] \tag{L14}$$

The first law enables a non-deterministic normal form to be split into its component guarded assignments, which are then decided individually by the second law.

Next, it is shown how this normal form facilitates the generation of fault-based test cases.

4.3 Test Case Generation from Normal Forms

The presented normal form has been developed to facilitate the automatic generation of test cases that are able to detect anticipated faults. The algebraic refinement laws solve the problem of equivalent mutants (more precisely, refining mutants). The above laws were also the inspiration for our next test case generation theorem: it calculates the test equivalence class detecting a given fault.

Theorem 4.4 *Let P be a program and P^m a faulty mutation of this program with normal forms as follows*

$$P = c \;\vee\; \bigsqcap_j \{(a_j \wedge v := f_j) \mid 1 \le j \le m\}$$

$$P^m = d \;\vee\; \bigsqcap_k \{(b_k \wedge v := h_k) \mid 1 \le k \le n\}$$

Then, every representative test case of the test equivalence class

$$T_{\sim} \quad =_{df} \quad d_{\perp}; P\,, \qquad \textit{with } d = (\neg c \wedge d) \;\vee\; \bigvee_k (\neg c \wedge b_k \wedge \bigwedge_j (\neg a_j \vee (f_j \neq h_k)))$$

is able to detect the fault in D^m. □

Instead of presenting the formal proof, we give an informal argument: In order to detect an error, the domains of the test equivalence classes must contain these input values where refinement does not hold. We have two cases of non-refinement: (1) P^m does not terminate but P does; (2) both are terminating but with different results.

(i) Those test cases have to be added where the mutant does not terminate, but the original program does. That is when $(\neg c \wedge d)$ holds.

(ii) In the terminating case, by the two laws L12 and L13, it follows that all combinations of guarded commands must be tested regarding refinement of the original one by the mutated one. Those, where this refinement test fails contribute to the test equivalence class. Law L14 tells us that refinement between two guarded commands holds iff $[b_k \Rightarrow (a_j \wedge (f_j = h_k))]$. Negating this gives $\exists v, v' \bullet b_k \wedge (\neg a_j \vee (f_j \neq h_k))$. Since we are only interested in test cases where the output is defined, we add the constraint $\neg c$. We see that this condition is at the heart of our test domain. Since we have to show non-refinement, this must hold for all the non-deterministic choices of P $(\bigwedge_j)$. Finally, each non-deterministic choice of P^m may contribute to non-refinement $(\bigvee_k)$.

Example 4.5 Consider the following example of a program *Min* for computing the minimum of two numbers x, y.

$$Min \quad =_{df} \quad z := x \lhd x \leq y \rhd z := y$$

In mutation testing, the assumption is made that programmers make small errors. A common error is to mix operators. The mutant Min^m models such an error.

$$Min^m \quad =_{df} \quad z := x \lhd x \boxed{\geq} y \rhd z := y$$

Their normal forms can be easily derived:

$$\begin{array}{ll} & Min \\ = & \{\text{adding identity assignments}\} \\ & x, y, z := x, y, x \lhd x \leq y \rhd x, y, z := x, y, y \\ = & \{\text{by L2}\} \\ & ((x \leq y) \wedge x, y, z := x, y, x) \sqcap \neg(x \leq y) \wedge x, y, z := x, y, y) \end{array}$$

$$\begin{array}{ll} & Min^m \\ = & \{\text{adding identity assignments}\} \\ & x, y, z := x, y, x \lhd x \geq y \rhd x, y, z := x, y, y \\ = & \{\text{by L2}\} \\ & ((x \geq y) \wedge x, y, z := x, y, x) \sqcap (\neg(x \geq y) \wedge x, y, z := x, y, y) \end{array}$$

According to Theorem 4.4 we have

$$
\begin{aligned}
d = & \\
& (\neg \mathbf{false} \wedge \mathbf{false}) \ \vee \ \bigvee_{k\in\{1,2\}} (\neg \mathbf{false} \wedge b_k \wedge \bigwedge_{j\in\{1,2\}} (\neg a_j \vee (f_j \neq h_k))) \\
= & \\
& (x \geq y \wedge (x > y \vee \mathbf{false}) \wedge (x \leq y \vee x \neq y)) \\
& \vee \\
& (x < y \wedge (x > y \vee x \neq y) \wedge (x \leq y \vee \mathbf{false})) \\
= & \\
& x > y \vee x < y
\end{aligned}
$$

Note that the case where $x = y$ has been correctly excluded from the fault-detecting equivalence class, since such test cases cannot distinguish the two versions of the program.

4.4 Recursion

Both, theory and intuition tell us that recursive programs cannot be represented as a finite normal form. The degree of non-determinism of a recursion cannot be expressed by a finite disjunction, because it depends on the initial state. Kleene's Theorem tells us that the normal form of a recursive program is the least upper bound of an infinite series of program approximations $\bigsqcup S^0, S^1, \ldots$ where each approximation is a refinement of its predecessor, thus $S^i \sqsubseteq S^{i+1}$.

Theorem 4.6 (Kleene) *If F is continuous then*

$$\mu X \bullet F(X) \quad = \quad \bigsqcup_n F^n(\mathbf{true})$$

where $F^0(X) \quad =_{df} \quad \mathbf{true}$*, and* $F^{n+1}(X) \quad =_{df} \quad F(F^n(X))$ □

Operators that distribute through least upper bounds of descending chains are called *continuous*. Fortunately, all operators in our language are continuous and, therefore, this normal form transformation can be applied. Unfortunately, this infinite normal form can never be computed in its entirety; however, for each n, the finite normal form can be readily computed. The normal form for our full programming language is, thus, defined as follows

Definition 4.7 (Infinite Normal Form) An *infinite normal form* for recursive programs is a program theoretically represented as least upper bound of descending chains of *finite normal forms*. Formally, it is of form

$$\bigsqcup S \qquad \text{with } S = \langle (c_n \vee Q_n) \mid n \in \mathbb{N} \rangle$$

S being a descending chain of approximations and Q being a non-deterministic normal form, i.e. a disjunction of guarded commands.

For test case generation, again, refinement between the original and the mutant must be checked. Fortunately, the following law from [11] tells us that we can decompose the problem.

$$(\bigsqcup S) \sqsubseteq (\bigsqcup T) \quad \textbf{iff} \quad \forall i : i \in \mathbb{N} \bullet S_i \sqsubseteq (\bigsqcup T) \tag{L15}$$

The central idea to deal with recursive programs in our test case generation approach is to approximate the normal form of both the program and the mutant until non-refinement can be detected. For equivalent mutants an upper limit n will determine when to stop the search. An example shall illustrate this.

Example 4.8 Assume that we want to find an index t pointing to the smallest element in an array $A[1..n]$, where n is the length of the array and $n > 0$. A program for finding such a minimum can be expressed in our programming language as follows:

$$\begin{array}{rcl} MIN & =_{df} & k := 2;\ t := 1;\ \mu X \bullet ((B;\ X) \lhd k \leq n \rhd k, t := k, t) \\ B & =_{df} & (t := k;\ k := k + 1) \lhd A[k] < A[t] \rhd k := k + 1 \end{array}$$

Since, the normal form of $\mu X \bullet F(X)$ is infinite and has to be approximated, we first convert $F(X)$ into a (finite) normal form.

$$\begin{array}{rcl} F(X) & = & ((k \leq n \wedge A[k] < A[t]) \wedge (k, t := k + 1, k;\ X)) \\ & & \sqcap \\ & & ((k \leq n \wedge A[k] \geq A[t]) \wedge (k, t := k + 1, t;\ X)) \\ & & \sqcap \\ & & ((k > n) \wedge k, t := k, t) \end{array}$$

Next, the first elements in the approximation chain are computed. According to Kleene's theorem we have

$$S^1 \quad =_{df} \quad F(\textbf{true}) \quad = \quad (k \leq n) \ \vee ((k > n) \wedge k, t := k, t)$$

The first approximation describes the exact behaviour only if the iteration is not entered. The second approximation describes the behaviour already more appropriately, taking one iteration into account. Note how the non-termination condition

gets stronger.

$$
\begin{array}{lcl}
S^2 =_{df} F(S^1) & = & (k+1 \leq n \wedge A[k] < A[t]) \vee \\
& & (((k \leq n \wedge k+1 > n \wedge A[k] < A[t]) \wedge (k,t := k+1, k)) \\
& & \sqcap \\
& & (k+1 \leq n \wedge A[k] \geq A[t]) \\
& & \vee ((k \leq n \wedge k+1 > n \wedge A[k] \geq A[t]) \wedge (k,t := k+1, t)) \\
& & \sqcap \\
& & (\mathbf{false}) \vee ((k > n) \wedge k,t := k,t) \\
& = & (k < n) \vee \\
& & \quad (((k = n \wedge A[k] < A[t]) \wedge (k,t := k+1, k)) \\
& & \quad \sqcap \\
& & \quad ((k = n \wedge A[k] \geq A[t]) \wedge (k,t := k+1, t)) \\
& & \quad \sqcap \\
& & \quad ((k > n) \wedge k,t := k,t))
\end{array}
$$

It can be seen from the first three approximations that our normal form approximations represent computation paths as guarded commands. As the approximation progresses, more and more paths are included. Obviously, the normal form approximations of the whole program, including the initialisations of k and t, can be easily obtained by substituting 2 for k and 1 for t in $S_1, S_2, \ldots$.

Next, we illustrate our fault-based testing technique, which first introduces a mutation, and then tries to approximate the mutant until refinement does not hold. A common error is to get the loop termination condition wrong. We can model this by the following mutant:

$$MIN^m =_{df} k := 2;\ t := 1;\ \mu X \bullet ((B;\ X) \lhd k \boxed{<} n \rhd k,t := k,t)$$

Its first approximation gives

$$S^{m1} =_{df} F(\mathbf{true}) = (k < n) \vee ((k \geq n) \wedge k,t := k,t)$$

By applying Theorem 4.4 to find test cases that can distinguish the two first approximations, we realize that such a test case does not exist, because $S^1 \sqsubseteq S^{m1}$. The calculation of the test equivalence class domain predicate $d^1(k,t)$ gives **false**:

$$
\begin{array}{l}
d^1(k,t) = \quad \{\text{by Theorem 4.4 }\} \\
\qquad (\neg(k \leq n) \wedge k < n) \ \vee \ (\neg(k \leq n) \wedge k \geq n \wedge (\neg(k > n) \vee \mathbf{false})) \\
\quad = \\
\qquad \mathbf{false} \vee \mathbf{false} \ = \ \mathbf{false}
\end{array}
$$

It is necessary to consider the second approximation of the mutant:

$$\begin{aligned} S^{m2} \;=_{df}\; F(S^{m1}) \;=\; & (k+1<n) \vee \\ & (((k+1=n \wedge A[k]<A[t]) \wedge (k,t:=k+1,k)) \\ & \sqcap \\ & ((k+1=n \wedge A[k]\geq A[t]) \wedge (k,t:=k+1,t)) \\ & \sqcap \\ & ((k\geq n) \wedge k,t:=k,t)) \end{aligned}$$

This time test cases exist. By applying Theorem 4.4 we get the test equivalence class that can find the error.

$$\begin{aligned} d^2(k,t) = \quad & \{\text{by Theorem 4.4}\} \\ & (\neg(k\leq n) \wedge k<n) \\ & \vee \\ & (k\geq n \wedge k+1=n \wedge A[k]<A[t] \wedge \ldots \\ & \vee \\ & (k\geq n \wedge k+1=n \wedge A[k]\geq A[t] \wedge \ldots \\ & \vee \\ & (k\geq n \wedge k\geq n \\ & \quad \wedge (\neg(k=n \wedge A[k]<A[t]) \vee \mathbf{true}) \\ & \quad \wedge (\neg(k=n \wedge A[k]\geq A[t]) \vee \mathbf{true}) \\ & \quad \wedge (\neg(k>n) \vee \mathbf{false})) \\ = & \\ & \mathbf{false} \vee (k\geq n \wedge k\leq n) \;=\; (k=n) \end{aligned}$$

By substituting the initialisation values ($k = 2$ and $t = 1$) the concrete fault-detecting test equivalence class is:

$$T_{\sim}{}^2 \;=\; (n=2)_{\perp}; MIN$$

The result is somehow surprising. The calculated test equivalence class says that every array with two elements can serve as a test case to detect the error. One might have expected that the error of leaving the loop too early could only be revealed if the minimum is the last element ($A[2] < A[1]$) resulting in different values for t (2 vs. 1). However, this condition disappears during the calculation. The reason is that the counter variable k is observable and that the two program versions can be distinguished by their different values for k (3 vs. 2).

In practice, k will often be a local variable and not part of the alphabet of the program. In such a case a stronger test equivalence class will be obtained. This illustrates the fact that it is important to fix the alphabet (the observables), before test cases are being designed.

Note also that the test equivalence class $T_\sim{}^2$ is just a 2-step approximation of the complete test equivalence class $T_\sim = (n \geq 2)_\perp ; MIN$.

5 Conclusions

Summary and Discussion.

The paper presented a novel theory of testing with a focus on fault detection. This fault-based testing theory is a conservative extension of the existing Unifying Theories of Programming [11]. It extents the application domain of Hoare & He's theory of programming to the discipline of testing. It has been demonstrated that the new theory enables the formal reasoning about test cases, more precisely about the fault detecting power of test cases. As a consequence, new laws for generating test cases could be developed.

The first test case generation law (Definition 3.4) is a general criterion for fault-based test cases. It is not completely new, but has been translated from our previous work [1] to the theory of designs. It states that a test case in order to find a fault in a design (which can range from specifications to programs) must be an abstraction of the original design, and in addition, it must not be an abstraction of the faulty design. No such test cases exist if the faulty design is a refinement of the original one. Note that the translation of this criterion from a different mathematical framework was straightforward. Since our previous definition was solely based on the algebraic properties of refinement, we just had to change the definition of refinement (from weakest precondition inclusion to implication). This demonstrates the generality of our refinement-based testing theory. In [2] we applied this technique to labelled transition systems.

The second test case generation law (Theorem 3.8) is more constructive and specialised for designs. It can be applied to specification languages that use pre- and postconditions, including VDM-SL, RSL, Z, B and OCL. Its finding is based on the conditions, when refinement between designs does not hold. It uses the operations on predicates (conditions and relations) to find the test cases. This approach forms the basis for our constraint solving approach to generate test cases from OCL specifications in [3].

The third law (Theorem 4.4) lifts the test case generation process to the syntactical level. By using a normal form representation of a given program (or specification), equivalence classes of test cases can be generated or, in the case of recursive programs, approximated. This is the technique, which is most likely to scale up to more complex programming and design languages. We have demonstrated the approach by using a small and simple programming language. However, the language is not trivial. It includes non-determinism and general recursion. A tool that uses this technique will combine constraint solving and symbolic manipulation.

Related Work

Fault-based testing was born in practice when testers started to assess the adequacy of their test cases by first injecting faults into their programs, and then by observing if the test cases could detect these faults. This technique of mutating the source code became well-known as mutation testing and goes back to the late 70-ies [10,8]; since then it has found many applications and has become the major assessment technique in empirical studies on new test case selection techniques [17].

To our present knowledge Budd and Gopal were the first who mutated specifications [5]. They applied a set of mutation operators to specifications given in predicate calculus form.

Tai and Su [15] proposed algorithms for 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 [14] 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 [13]. He presented the criteria to generate test cases to discriminate mutants, but did not use refinement. Woodward investigated mutation operators for algebraic specifications [18].

Burton presented a fault-based test case generator for Z specifications [6]. He uses a combination of a theorem prover and a collection of constraint solvers. The theorem prover generates a disjunctive normal form, simplifies the formulas and helps in formulating different testing strategies.

Black et al. studied mutation operators using the SMV model checker [4]. A group in York has recently started to use fault-based techniques for validating their CSP models [12]. 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 [7].

Wimmel and Jürjens [16] use mutation testing on specifications to extract those interaction sequences that are most likely to find security issues.

Future Work.

The presented theory is far from being final or stable. It is another step in our research aim to establish a unifying theory of testing. Such a theory will provide semantic links between different testing theories and models. These links will facilitate the systematic comparison of the results in different areas of testing, hopefully leading to new advances in testing.

References

[1] Aichernig, B. K., *Mutation Testing in the Refinement Calculus*, Formal Aspects of Computing Journal **15** (2003), pp. 280–295.

[2] Aichernig, B. K. and C. C. Delgado, *From faults via test purposes to test cases: on the fault-based testing of concurrent systems*, in: L. Baresi and R. Heckel, editors, *Proceedings of FASE'06, Fundamental Approaches to Software Engineering, Vienna, Austria, March 27–29, 2006*, Lecture Notes in Computer Science **3922** (2006), pp. 324–338.

[3] Aichernig, B. K. and P. A. P. Salas, *Test case generation by OCL mutation and constraint solving*, in: K.-Y. Cai, A. Ohnishi and M. Lau, editors, *QSIC 2OO5, Proceedings of the Fifth International Conference on Quality Software, Melbourne, Australia, September 19-21, 2005* (2005), pp. 64–71.

[4] Black, P., V. Okun and Y. Yesha, *Mutation of model checker specifications for test generation and evaluation*, in: *Mutation testing for the new century*, Kluwer Academic Publishers, 2001 pp. 14–20.

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

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

[7] de Souza, S., J. M. S. Fabbri and W. de Souza, *Mutation testing applied to Estelle specifications*, Software Quality Journal **8** (1999), pp. 285–301.

[8] DeMillo, R., R. Lipton and F. Sayward, *Hints on test data selection: Help for the practicing programmer*, IEEE Computer **11** (1978), pp. 34–41.

[9] Dick, J. and A. Faivre, *Automating the generation and sequencing of test cases from model-based specifications*, in: J. Woodcock and P. Larsen, editors, *Proceedings of FME'93: Industrial-Strength Formal Methods, International Symposium of Formal Methods Europe, April 1993, Odense, Denmark* (1993), pp. 268–284.

[10] Hamlet, R. G., *Testing programs with the aid of a compiler*, IEEE Transactions on Software Engineering **3** (1977), pp. 279–290.

[11] Hoare, C. and H. Jifeng, "Unifying Theories of Programming," Prentice-Hall International, 1998.

[12] Srivatanakul, T., 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* (2003), pp. 340–351.

[13] Stocks, P. A., "Applying formal methods to software testing," Ph.D. thesis, Department of computer science, University of Queensland (1993).

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

[15] Tai, K.-C. and H.-K. Su, *Test generation for Boolean expressions*, in: *Proceedings of the Eleventh Annual International Computer Software and Applications Conference (COMPSAC)*, 1987, pp. 278–284.

[16] Wimmel, G. and J. Jürjens, *Specification-based test generation for security-critical systems using mutations*, in: C. George and M. Huaikou, editors, *Proceedings of ICFEM'02, the International Conference of Formal Engineering Methods, October 21–25, 2002, Shanghai, China*, LNCS (2002), pp. 471–482.

[17] Wong, W. E., editor, "Mutation Testing for the New Century," Kluwer Academic Publishers, 2001.

[18] Woodward, M., *Errors in algebraic specifications and an experimental mutation testing tool.*, Software Engineering Journal (1993), pp. 211–224.