DOI 10.1007/s00165-008-0083-6
BCS © 2008
Formal Aspects of Computing (2009) 21: 33–64

Formal Aspects of Computing

Mutation testing in UTP

Bernhard K. Aichernig[1,2] and He Jifeng[3]

[1]Institute for Software Technology, Graz University of Technology, Graz, Austria. E-mail: aichernig@ist.tugraz.at
[2]International Institute for Software Technology, United Nations University, Macao, S.A.R. China
[3]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 UTP design predicates, the other on the algebraic properties of a small programming language.

Keywords: Specification-based testing; Fault-based testing; Mutation testing; Unifying theories of programming; Refinement calculus; 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 of this paper to add a theory of testing to Hoare & He's Unifying Theories of Programming [HH98].

The theory we contribute was designed to be a complement to the existing body of knowledge. Traditionally, theories of programming focus on semantic 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.

Correspondence and offprint requests to: B. Aichernig, E-mail: aichernig@ist.tugraz.at

```
context Ttype(a: int, b: int, c: int): String
pre: 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
```

```
context Ttype(a: int, b: int, c: int): String
pre: 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
```

Fig. 1. *Left*: original specification of triangle types. *Right*: mutation with a faulty conditional

Our fault-based testing approach is based on a rather old technique from program testing called *mutation testing*. The original idea goes back to the late 1970s [Ham77, DLS78] and works as follows:

The tester injects faults into a program under test, by deliberately changing its source code. Then, test cases are executed on these faulty versions of the program. If all the faults are detected, the set of test cases can be considered adequate. If the tests fail this quality assessment, then additional test cases can be designed until all faults are detected. The assumption is that other faults in the original program can be caught as well, if it is able to detect the injected faults. The process of fault injection by changing the program text is called *program mutation*, giving the technique its name. Consequently, the altered program versions are called mutants. Usually, for each injected fault a separate mutation is generated. Hence, each mutation represents and contains a single fault.

We extend this technique by applying it to first-order predicates denoting program behaviour. This lifts mutation testing to the specification level. By using Unifying Theory of Programming's (UTP) design predicates, we have the full theory of programming available, including notions of correctness and refinement. We use this theory to define criteria for finding or even generating the test cases that would detect the injected errors (mutations). Hence, rather than inventing test cases and assessing them, we are interested in constructive methods of generating the adequate test cases. By test cases we mean program stimuli (inputs) together with predicted reactions (outputs).

An example will clarify the general idea. The Object Constraint Language (OCL) of UML 2.0 has been chosen to highlight the relevancy of our technique, but any specification language for expressing pre-postcondition specifications would do.

Example 1.1 Consider the well-known Triangle example, specified in Fig. 1. The operation **Ttype** on the left-hand side returns the type of a triangle represented by three given side-lengths a, b, c. The precondition restricts the problem specification to cases where the values of a, b, c form a valid triangle. The specification can be mutated (altered) in several ways. One common error made by programmers is to mess up variables in a condition. Such an error can be represented as early as on the specification level by mutating a specification variable as shown in the mutant on the right hand side (underlined condition). Note that this faulty version would pass a type checker, because the correct variable name has been replaced by a valid identifier. In general, we exclude such mutations from our testing strategy that can be trivially detected by static type checking.

The aim of our mutation testing strategy is to design test cases that will detect implementations of the faulty specification (mutant). The following three test cases covering each case of the original demonstrate that simple case analysis (branch coverage) is too weak.

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

None of these test cases would be able to distinguish the original from the mutant, since the predicted results ($result$) of the test cases and the actual results of a faulty implementation of the mutant would coincide. In contrast, the following test case is able to distinguish the original triangle operation from its mutant:

$$a = 1,\ b = 2,\ c = 2,\ result = isosceles$$

The predicted $result$ is in conflict with the actual outcome of the mutant ($isosceles \neq equilateral$). □

This paper is devoted to techniques for finding such fault-detecting test cases systematically.

Note that the strategy of mutation testing is not based on the syntactical structure of the source code, like statement or branch-coverage techniques. Its aim is to cover anticipated faults. This is why it is called a fault-based testing technique. It is this fundamentally different philosophy of our fault-based testing theory that adds a further dimension to the theories of programming (UTP). 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 (by running such tests) 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 linking theories of testing and programming.

The paper is structured as follows. After this general introduction, Sect. 2 introduces UTP [HH98], the mathematical framework used throughout this article. In particular, an overview of the theory of designs is presented. Section 3 briefly discusses the testing terminology and, based on the theory of designs, a series of formal definitions of concepts like test cases, test equivalence classes, faults, and a criterion for finding faults is presented. It is this section that highlights the important role of refinement in our testing theory. The next two sections include the main contributions of this paper. Section 4 contains a construction for test cases that will find anticipated errors in a design. This test case generation method works on the semantic level of designs. At the end of this section, a tool is briefly discussed. In Sect. 5 a purely algebraic (syntax-oriented) test case generation algorithm is presented. It is based on the algebraic properties of a small, but non-trivial, programming language. Finally, in Sect. 6 we discuss the results, give a review of related work, and present an overview of further research directions.

2. Unifying theories of programming

The present theory of testing is based on the work of Hoare and He on Unifying Theories of Programming (UTP) originally published in [HH98]. In the following we present a brief introduction of UTP and give motivations for its relevance to testing.

2.1. Unification of theories

In every scientific discipline phases of specialisation and unification can be observed. During specialisation scientists concentrate on some narrowly defined phenomenon and aim to discover the laws which govern it. Over time, it is realised that the laws are special cases of a more general theory and a unification process starts. The aim is a unifying theory that clearly and convincingly explains a broader range of phenomena. A proposed unification of theories often receives spectacular confirmation and reward by the prediction of new discoveries or by the development of new technologies. However, a unifying theory is usually complementary to the theories that it links, and does not seek to replace them.

In [HH98] Hoare and He are aiming at unification in computer science. They saw the need for a comprehensive theory of programming that

- includes a convincing approach to the study of a range of languages in which computer programs may be expressed,
- must introduce basic concepts and properties which are common to the whole range of programming methods and languages,
- must deal separately with the additions and variations which are particular to specific groups of related programming languages,
- should aim to treat each aspect and feature in the simplest possible fashion and in isolation from all the other features with which it may be combined or confused.

Our theory of testing originated out of these motivations for unification. With the advent of formal methods, formal verification and testing split into separate areas of research. Over time both areas advanced. In the 1990s a unification process started in the area of verification [Gau95]. As a result, most of today's formal method tools have a test case generator, acknowledging the fact that every proof needs systematic testing of its underlying assumptions. Furthermore, researchers in testing investigate the role of formal models in the automation of black-box testing. However, many results in testing and formal methods remain unrelated. In this work, we aim to contribute to a further unification. The notion of testing and test cases is added to the existing UTP. Remarkable is the fact that the concept of refinement is used to relate test cases with a theory of specifications and programs.

2.2. Theories of programming

An essential key to the success of natural sciences was the ability to formulate theories about observable phenomena. These observables are described in a specialised language, that name and relate the outcome of experiments. Often equations or inequations are used, in general mathematical relations. The same holds for the science of programming, where the descriptions of observables are called by their logical term *predicate*. In the following theory of testing, predicates are used in the same way as in a scientific theory, to describe the observable behaviour of a program when it is executed by a computer. In fact, we will define the meaning of a program, as well as the meaning of its test cases, as a predicate in first order logic.

Every scientific theory contains free variables that represent measurable quantities in the real world. In our theory of testing these free variables stand for program variables, conceptual variables representing a system's state, or observable input-output streams. The chosen collection of names is called the *alphabet*.

In engineering, predicates are not solely used to describe existing phenomena, but to specify desired properties. Such *requirements specifications* describe the behaviour of a device in all possible circumstances and are a starting point for developing a product. In addition, the test cases are derived from these requirements descriptions. It will be seen that test cases are actually a special form of requirements specification, designed for experimentation (or in computer science terms, for execution).

2.3. A theory of designs

In UTP by convention, observations made at the beginning of an experiment are denoted by *undecorated* variables (x, y), whereas observations made on later occasions will be *decorated* (x', y'). The set of these observation capturing variables is called the *alphabet*.

During experiments it is usual to wait for some initial transient behaviour to stabilise before making any further observation. In order to express this a Boolean variable ok and its decorated version ok' are introduced. Here, a true-valued variable ok stands for a successful initialisation and start of a program and $ok' = true$ denotes its successful *termination*.

In the theory of programming not every possible predicate is useful. It is necessary to restrict ourselves to predicates that satisfy certain *healthiness conditions*: e.g. a predicate describing a program that produces output without being started should be excluded from the theory ($\neg ok \wedge x' = 1$). In addition, the results of the theory must match the expected observations in reality, e.g. a program that fails to terminate sequentially composed with any program must always lead to non-termination of the whole composition (this is the technical motivation for introducing ok, ok'). We call the subset of predicates that meet our requirements *designs*. The following definitions and theorems are a reproduction of the original presentation of UTP [HH98].

Definition 2.1 (Design) Let p and Q be predicates not containing ok or ok' and p having only undecorated variables.

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

A *design* is a relation whose predicate is (or could be) expressed in this form.[1] □

As can be seen, a design predicate represents a pre-postcondition specification, a concept well-known from VDM [Jon86], RAISE [RAI95], B [Abr96] and more recently OCL [WK03].

Example 2.1 (Square Root) The following contract is a design of a square root algorithm using a program variable x for input and output. A constant e specifies the precision of the computation.

$$(x \geq 0 \wedge e > 0) \vdash (-e \leq x \; - \; x'^2 \leq e)$$

□

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 2.2 (Assignment) Given a program variable x and an expression e

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

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

Definition 2.3 (Conditional)

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

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 $wf =$ **true**.

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

Definition 2.4 (Sequential Composition)

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

□

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

Definition 2.5 (Demonic Choice)

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

□

UTP provides a series of theorems and lemmas expressing the basic algebraic properties of such programming constructs, e.g.:

Theorem 2.1

$$\begin{aligned}
(p_1 \vdash Q_1) \sqcap (p_2 \vdash Q_2) &= (p_1 \wedge p_2 \vdash Q_1 \vee Q_2) \\
(p_1 \vdash Q_1) \lhd b \rhd (p_2 \vdash Q_2) &= (p_1 \lhd b \rhd p_2 \vdash Q_1 \lhd b \rhd Q_2) \\
(p_1 \vdash Q_1);\; (p_2 \vdash Q_2) &= (p_1 \wedge \neg(Q_1;\; \neg p_2) \vdash Q_1;\; Q_2)
\end{aligned}$$

□

What keeps the theory simple and elegant is the fact that in UTP correctness is represented by logical implication. Hence, implication establishes a refinement order (actually a lattice) over designs. Thus, more concrete implementations imply more abstract specifications.

[1] This is a non-standard, more direct definition of designs. Our designs satisfy healthiness conditions H1–H3 in [HH98].

Definition 2.6 (Refinement)

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

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 [BvW98, Mor94] style $D_1 \sqsubseteq D_2$. □

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

Theorem 2.2 (Refinement of Designs)

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

□

In our theory, the worst of all programs is a non-terminating one, sometimes called *Abort* or *Chaos*:

Definition 2.7 (Abort)

$$\bot \;=_{df}\; \textbf{true} \;=\; \textbf{false} \vdash \textbf{true} \;=\; \textbf{false} \vdash \textbf{false} \;=\; \textbf{false} \vdash Q$$ □

The observations of non-terminating programs are completely unpredictable; anything is possible. Therefore programmers aim to deliver better programs. In our theory of designs the notion of "better" is captured by the implication ordering. Thus every program P that terminates is better than a non-terminating one.

$$\forall v, w, \cdots \in A \bullet P \Rightarrow \bot, \quad \text{for all } P \text{ with alphabet } A.$$

This refinement ordering defines a complete lattice over designs, with $\bot$ as the bottom element and $\top$, representing a non-implementable, magic-like, program as top element.

Definition 2.8 (Magic)

$$\top \;=_{df}\; (\textbf{true} \vdash \textbf{false})$$

□

The program $\top$ is called magic since it magically refines (implements) every possible design D:

$$[D \Leftarrow \top], \quad \text{for all } D \text{ with alphabet } A.$$

Another interpretation of magic is that it can never be started, since $\top = \neg ok$.

From the definitions above it follows that the *meet* operator of this lattice is the deterministic choice and its dual *join* operator is defined as

Definition 2.9 (Join)

$$P \sqcup Q \;=_{df}\; P \wedge Q$$

□

Theorem 2.3

$$(p_1 \vdash Q_1) \sqcup (p_2 \vdash Q_2) \;=\; (p_1 \vee p_2 \vdash ((p_1 \Rightarrow Q_1) \wedge (p_2 \Rightarrow Q_2)))$$

Finally, iteration is expressed by means of recursive definitions. Since designs form a complete lattice and the operators are monotonic, the weakest fixed point exists. This ensures that the result of any recursion is still a design.

3. Modelling faults in designs

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 the theory of refinement. However, first the vocabulary needs some clarification.

3.1. From errors via faults to failures

The vocabulary of computer scientists is rich with terms for naming the unwanted: bug, error, defect, fault, failure, etc. are commonly used without great care. However, in a discussion on testing it is necessary to differentiate between them in order to prevent confusion. Here, we adopt the standard terminology as recommended by the Institute of Electronics and Electrical Engineers (IEEE) Computer Society [Soc90]:

Definition 3.1 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 occurs when a fault executes.

In this work we aim to design 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 design descriptions. The faults will be introduced by deliberately changing a design, resulting in wrong behaviour possibly causing a failure.

What distinguishes the following theory from other testing theories is the fact that we define all test artifacts as designs. This means that we give test cases, test suites and even test equivalence classes a uniform (predicative) semantics in UTP. The fundamental insight behind this approach is that all these artifacts represent descriptions of a system to be built (or under test). They simply vary with respect to information content. A test case, for example, can be seen as a specification of a system's response to a single input. Consequently a test suite, being a collection of test cases, can also be considered as a (partial) specification. The same holds for test equivalence classes that represent a subset of a system's behaviour. Viewing testing artifacts as designs results in a very simple testing theory in which test cases, specifications and implementations can be easily related via the notion of refinement. A consequence of this semantic interpretation is that test cases are actually abstractions of a system specification. This seems often strange to people since a test case is experienced as something very concrete. However, from the information content point of view a test case is perfectly abstract: only for a given stimulus (input) the behaviour is defined. It is this limited information that makes test cases so easily understandable.

3.2. Test cases

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

Definition 3.2 (Test Case, deterministic) Let i be an *input vector* and o be an expected *output vector*, both being lists of values, having the same length as the variable lists v and v', respectively. Then, a test case T is defined being a design predicate:

$$T(i, o) \;=_{df}\; v = i \vdash v' = o$$

□

Sometimes test cases have to take non-determinism into account, therefore we define non-deterministic test cases as follows:

Definition 3.3 (Test Case, non-deterministic)

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

where c is a condition on the after state space defining the set of expected outcomes. □

Obviously, non-deterministic test cases having the same input can be compared regarding their strength. Thus, we have

Theorem 3.1

$$[T_{\sqcap}(i, c) \;\Rightarrow\; T_{\sqcap}(i, d)] \quad \textbf{iff} \quad [c \;\Rightarrow\; d]$$

□

This shows that non-deterministic test cases form a partial order. If we fix the input i the test case $T_{\sqcap}(i, \textbf{true})$ is the smallest test case and $T_{\sqcap}(i, \textbf{false})$ the largest. However, the question arises as to how to interpret these limits, and if they are useful as test cases. $T_{\sqcap}(i, \textbf{true})$ is a test case without any output prediction. It is useful in Robustness Testing, where i lies outside the specified input domain, or where one is just interested in exploring the reactions to different inputs. $T_{\sqcap}(i, \textbf{false})$ is equivalent to $\neg(ok \wedge v = i)$ and means that such programs cannot be started with input i; such tests are infeasible.

Definition 3.4 (Explorative Test Case)

$$T_{?}(i) \;=_{df}\; T_{\sqcap}(i, \textbf{true})$$

□

Definition 3.5 (Infeasible Test Case)

$$T_{\varnothing}(i) \;=_{df}\; T_{\sqcap}(i, \mathbf{false})$$

□

We get the following order of test cases:

Theorem 3.2 (Order of Test Cases) For a given input vector i, output vector o and condition c

$$\bot \;\sqsubseteq\; T_{?}(i) \;\sqsubseteq\; T_{\sqcap}(i, c) \;\sqsubseteq\; T(i, o) \;\sqsubseteq\; T_{\varnothing}(i) \;\sqsubseteq\; \top, \quad \text{provided } c(o) \text{ holds.}$$

□

A collection of test cases is called a *test suite* and is defined as the least upper bound of its test cases:

Definition 3.6 (Test Suite) Given a set s of test cases $t_1, \ldots, t_n$

$$TS(s) \;=_{df}\; t_1 \sqcup \cdots \sqcup t_n$$

□

The definition coincides with our intuition: an implementation has to pass all test cases in a test suite. In case of contradicting test cases, this is impossible, which is expressed by the test suite being equal to magic ($\top$).

From lattice theory it follows that adding test cases is refinement:

Theorem 3.3 Let T_1, T_2 be test cases of any type

$$T_i \;\sqsubseteq\; T_1 \sqcup T_2, \quad i \in \{1, 2\}$$

□

Given a program under test, we can talk about an exhaustive test suite, covering the whole input and output domain.

Definition 3.7 (Exhaustive Test Suite) Let D be a design, its set of *exhaustive test suites* is defined as

$$TS_{exhaustive} \;=_{df}\; \{TS(s) \mid TS(s) = D\}$$

□

In this definition the notion of exhaustiveness is based on designs, not on the program under test. Thus, an exhaustive test suite only needs to cover the defined (specified) input domain. The following theorem states this more explicitly:

Theorem 3.4 Given a design $D = p \vdash Q$ and one of its exhaustive test suites $ts_{exhaustive} \in TS_{exhaustive}$

$$ts_{exhaustive} \sqcup T_{?}(i) \;=\; ts_{exhaustive}, \quad \text{provided } p(i) \text{ holds.}$$

Proof. The proof uses the fact that test cases are designs. Therefore, lattice theory can be used.

$$\begin{array}{lll} ts_{exhaustive} \sqcup T_{?}(i) = & & \{\text{by definition of exhaustive test suites}\} \\ & D \sqcup T_{?}(i) = D & \\ = & & \{\text{by lattice theory}\} \\ & T_{?}(i) \sqsubseteq D & \\ = & & \{\text{by definition of refinement and Theorem 2.2}\} \\ & [v = i \Rightarrow p] \wedge [(v = i \wedge Q) \Rightarrow \mathbf{true}] & \\ = & & \{\text{since p(i) holds}\} \\ & true & \end{array}$$

□

This theorem says that explorative test cases for specified behaviour do not add new information (test cases) to the set of exhaustive test cases. Note however, that it might be useful to add explorative test cases with inputs outside the precondition p for exploring the unspecified behaviour of a program.

The last theorem leads us to a more general observation: Adding an additional test case to an exhaustive test suite is redundant.

Theorem 3.5 Given a design D and an exhaustive test suite $ts_{exhaustive} \in TS_{exhaustive}$. Furthermore, we have a test case $t \sqsubseteq D$ expressing the fact that t has been derived from D. Then,

$$ts_{exhaustive} \sqcup t \;=\; ts_{exhaustive}$$

Proof. The proof fully exploits the lattice properties of designs.

$$
\begin{array}{lll}
ts_{exhaustive} \sqcup t & = & \{\text{by definition of exhaustive test suites}\} \\
D \sqcup t & & \\
& = & \{\text{by lattice theory, since } t \sqsubseteq D\} \\
D & & \\
& = & \\
ts_{exhaustive} & &
\end{array}
$$

□

Having clarified the relations between different test cases, in the following, their relation to specifications and implementations is rendered more precisely.

Previous work of the first author [Aic01b] 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 (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 3.8 Let T be a test suite, S a specification, and I an implementation, all being designs, and

$$T \sqsubseteq S \sqsubseteq I$$

we define

- T as a *correct test suite* with respect to S,
- all test cases in T as *correct test cases* with respect to S,
- implementation I *passes* a test suite (test case) T,
- implementation I *conforms* to specification S.[2] □

The following theorem makes the relation of input and output explicit:

Theorem 3.6

$$
\begin{array}{lll}
T(i, o) \sqsubseteq D & \textbf{iff} & v := o \sqsubseteq (v := i;\ D) \\
T_{\sqcap}(i, c) \sqsubseteq D & \textbf{iff} & c(v') \sqsubseteq (v := i;\ D)
\end{array}
$$

□

So far the discussion has focused on correctness, thus when implementations are passing the test cases. However, the aim of testing is to find faults. In the following, we concentrate on faults and discuss how they are modelled in our theory of testing, leading to a fault-based testing strategy.

3.3. Faults

According to Definition 3.1, *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 be passed on during the development process; 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 are most suitable for representing faults.

Definition 3.9 (Faulty Design) Let D be a design, and D^m its mutated version, meaning that D^m has been produced by slightly altering D. Furthermore, let the mutation represent a fault model. Then, the mutated design D^m is defined to be a *faulty design* (or a *faulty mutation*) of D, if

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

(or $\neg(D \sqsubseteq D^m)$). □

[2] In testing, refinement between a specification and an implementation under test is called *conformance*.

Not all mutations (changes) of a design lead to observable failures. In mutation testing, mutants that behave equivalent to the original are called *equivalent mutants*. These mutants are excluded from the set of faulty designs. Strictly speaking, *equivalent mutants* are not observational equivalent and, therefore should be named *refining mutants*. They may produce additional observations outside the precondition of the original design D. However, since we ignore test cases outside the precondition of the original design, this additional behaviour cannot be detected by testing. Hence, we stick to the common notion of *equivalent mutant* keeping in mind its refinement semantics.

4. Designing test cases

It is common knowledge that exhaustive testing of software cannot be achieved in general. Therefore, the essential question of testing is the selection of adequate test cases. What is considered being adequate depends highly on the assumptions made—the *test hypothesis*. Typical types of test hypotheses are *regularity* and *uniformity* hypotheses. Example of the former is the assumption that if a sorting algorithm works for sequences up to 10 entries, it will also work for more; example of the latter is the assumption that certain input (or output) domains form equivalence partitions, and that consequently only one test case per partition is sufficient. In general, the stronger the hypothesis the fewer test cases are necessary.

The test hypothesis is closely related to the notion of *test coverage*. It defines a unit to measure the adequacy of a set of test cases based on a test hypothesis. Traditionally, test coverage has been defined based on program text, like statement, branch, and data-flow coverage. For example, aiming for statement coverage is based on the uniformity hypothesis that it is sufficient to execute every statement in a program once—a rather strong assumption.

Here, we take a fault-based approach: test cases will be designed according to their ability to detect anticipated faults.

4.1. Fault-based testing

In fault-based testing a test designer does not focus on a particular coverage of a program or its specification, but on concrete faults that should be detected. The focus on possible faults enables a tester to incorporate his expertise in both the application domain and the particular system under test. In testing the security or safety of a system typically a fault-based test design strategy is applied.

Perhaps, the most well-known fault-based strategy is mutation testing, where faults are modelled as changes in the program text. Mutation testing has been introduced by Hamlet [Ham77] and DeMillo et al. [DLS78]. Often it is used as a means of assessing test suites. When a program passes all tests in a suite, mutant programs are generated by introducing small errors into the source code of the program under test. The suite is assessed in terms of how many mutants it distinguishes from the original program. If some mutants pass the test suite, additional test cases are designed until all mutants that reflect errors can be distinguished. The number of mutant programs to be generated is defined by a collection of mutant operators that represent typical errors made by programmers. A hypothesis of this technique is that programmers only make small errors.

In previous work [Aic01a, Aic03] we have extended mutation testing to the notion of contracts in Back and von Wright's refinement calculus [BvW98]. In this section, we first translate these results to the theory of designs. Then, as a new contribution we provide a more constructive rule for designing fault-based test cases.

First, the following theorem links the existence of non-passing (or incorrect) tests to faulty designs.

Theorem 4.1 Given a design D, and a faulty design D^m, then there exists a test case t, with $t \sqsubseteq D$, such that $t \not\sqsubseteq D^m$.

Proof. Assume that such a test case does not exist and for all test cases $t \sqsubseteq D$ also $t \sqsubseteq D^m$ holds. Hence, for all test cases t_i in an exhaustive test suite $ts_{exhaustive} = t_1 \sqcup \cdots \sqcup t_n$ of D, also $t_i \sqsubseteq D^m$ holds. Given the fact [HH98] that for any design S

$$t_1 \sqcup \cdots \sqcup t_n \sqsubseteq S \quad \textbf{iff} \quad \forall\, i \bullet t_i \sqsubseteq S$$

holds, it follows that

$$ts_{exhaustive} \sqsubseteq D^m$$

By definition of exhaustive test suites (Definition 3.7) we have $ts_{exhaustive} = D$, hence it follows that

$$D \sqsubseteq D^m$$

This is a contradiction to our assumption that D^m is a faulty design (see Definition 3.9). Consequently, the theorem holds. □

Finding such a test case t 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-detecting test case as follows:

Definition 4.1 (Fault-detecting Test Case) Let t be either a deterministic or non-deterministic input-output test case. Furthermore, D is a design and D^m its faulty version. Then, t is a *fault-detecting test case* when

$$(t \sqsubseteq D) \quad \text{and} \quad (t \not\sqsubseteq D^m)$$

We say that a fault-detecting test case *detects* the fault in D^m. Alternatively we can say that the test case *distinguishes* D and D^m. In the context of mutation testing, one says that t *kills* the mutant D^m. □

It is important to point out that in case of a non-deterministic D^m, there is no guarantee that the fault-detecting test case will definitely kill the mutant. The mutant might always produce the output consistent with the test case. However, the test case ensures that whenever a wrong output is produced this will be detected. This is a general problem of testing non-deterministic programs.

We also want to remind the reader that our definitions solely rely on the lattice properties of designs. Therefore, our fault-detecting testing strategy scales up to other lattice-based test models as long as an appropriate refinement definition is used. More precisely, this means that the refinement notation must preserve the same algebraic laws. It is this lattice structure that enabled us to translate our previous results into the theory of designs. In [Aic03] we came to the same conclusions in a predicate transformer semantics, with refinement defined in terms of weakest preconditions.

The presented definition of a fault-detecting test case, able to detect a certain fault, presents a property that could be exploited by constraint solvers to search for a solution of such a test case in a finite domain. However, although feasible in principle, it is not the most efficient way to find such test cases. The reason is that the definition, because of its generality, does not exploit the refinement definition in the concrete test model. In the following we present a more constructive way to generate test cases for designs.

4.2. Fault-detecting test 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 (see, e.g. [Bei90, Jor02]). 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 popular equivalence class testing approach regarding formal specification is *DNF partitioning*—the rewriting of a formal specification into its disjunctive normal form (see, e.g. [DF93, Sto93, HNS97, BBH02]). 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*. In general for a test equivalence class $t_=$ and its associated design D, refinement holds: $t_= \sqsubseteq D$.

Definition 4.2 [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 condition d is called the *domain* of the test equivalence class. □

The definition makes use of the *assertion* operator $b_\perp =_{df} true \vdash ((v = v') \lhd b \rhd \perp)$, leading to a design which has no effect on variables v if the condition holds (skip), and behaves like abort (non-termination) 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 an associated test equivalence class:

Theorem 4.2 Given a design $D = p \vdash Q$ and one of its equivalence classes. Then,

$$T_\sim \sqsubseteq D$$

Proof. The proof uses the fact that an assertion in front of a design behaves like a precondition.

$$\begin{array}{lcl} T_\sim & = & d_\perp;\ D \\ & = & (d \wedge p) \vdash Q \\ & \Leftarrow & p \vdash Q \\ & = & D \end{array}$$

□

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

Definition 4.3 (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

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

□

This definition ensures that the output condition of a representative test case is not weaker than the 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. The rational behind this construction is the fact that, for a test case be able to distinguish a design D from its faulty sibling D^m, refinement between the two must not hold. Furthermore, for designs one may observe two places (cases) where refinement may be violated, the precondition and the postcondition. The domain d 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 test inputs which will produce different output values.

Theorem 4.3 (Fault-detecting Equivalence Class) Given a design $D = p \vdash Q$ and its faulty design $D^m = p^m \vdash Q^m$ with $D \not\sqsubseteq D^m$. For simplicity, we assume that $Q \equiv (p \Rightarrow Q)$. Then every representative test case of the test equivalence class

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

is able to detect the fault in D^m.

Proof. We first show that a representative test case $t = T_\sqcap(i, c)$ is a correct test case with respect to D:

$$\begin{array}{lcl} t \sqsubseteq D & = & \\ & & [(v = i) \Rightarrow p] \quad \wedge \quad [((v = i) \wedge Q) \Rightarrow c] \\ & = & \quad \{\text{since } p(i) \text{ holds by Definition 4.3}\} \\ & & \mathit{true} \quad \wedge \quad [((v = i) \wedge Q) \Rightarrow c] \\ & = & \quad \{\text{since } [Q(i) \equiv c] \text{ holds by Definition 4.3}\} \\ & & \mathit{true} \wedge \mathit{true} \\ & = & \\ & & \text{true} \end{array}$$

Next, we prove that a representative test case t covers the fault in the mutant $t \not\sqsubseteq D^m$: From the definition of the test equivalence class, we see that we have two cases for t.

Case 1 $t = T_{\sqcap}(i, c)$ and $(\neg p^m)(i)$:

$$
\begin{array}{lll}
t \not\sqsubseteq D^m & = & \\
 & & \neg[(v = i) \Rightarrow p^m] \quad \vee \quad \neg[((v = i) \wedge Q^m) \Rightarrow c] \\
 & = & \\
 & & \exists\ v \bullet ((v = i) \wedge \neg p^m) \quad \vee \quad \exists\ v, v' \bullet ((v = i) \wedge Q^m \wedge \neg c) \\
 & = & \quad \{i \text{ is a witness to 1st. disjunct, since } p^m(i) \text{ holds}\} \\
 & & \mathit{true} \quad \vee \quad \exists\ v, v' \bullet ((v = i) \wedge Q^m \wedge \neg c) \\
 & = & \\
 & & \text{true}
\end{array}
$$

Case 2 $t = T_{\sqcap}(i, c)$ and $(\exists\, v' \bullet (Q^m \wedge \neg Q))(i)$:

$$
\begin{array}{lll}
t \not\sqsubseteq D^m & = & \\
 & & \neg[(v = i) \Rightarrow p^m] \quad \vee \quad \neg[((v = i) \wedge Q^m) \Rightarrow c] \\
 & = & \\
 & & \exists\ v \bullet ((v = i) \wedge \neg p^m) \quad \vee \quad \exists\ v, v' \bullet ((v = i) \wedge Q^m \wedge \neg c) \\
 & = & \quad \{\text{by definition of the representative test case}\} \\
 & & \exists\ v \bullet ((v = i) \wedge \neg p^m) \quad \vee \quad \exists\ v, v' \bullet ((v = i) \wedge Q^m \wedge \neg Q(i)) \\
 & = & \quad \{t \text{ is a witness to 2nd disjunct}\} \\
 & & \exists\ v \bullet ((v = i) \wedge \neg p^m) \quad \vee \quad \mathit{true} \\
 & = & \\
 & & \text{true}
\end{array}
$$

□

4.3. Tool support

Nothing is as practical as a good theory. Hence, based on the presented theory we are currently working on fault-based test case generators. The first prototype tool we have developed in our group is a test case generator for the Object Constraint Language OCL. Here, the user either introduces faults interactively via a GUI or uses a set of standard mutation operators to generate mutant specifications automatically. The tool generates one test case out of the test equivalence class that will detect the error.

The theoretical foundation of the tool is Theorem 4.3. The automation exploits the fact that we are interested in non-refinement. Thus, instead of showing refinement where we need to demonstrate that the implication holds for all possible observations, here the existence of one (counter)example is sufficient. Hence, the problem of finding a test case can be represented as a constraint satisfaction problem (CSP).

A CSP 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.

We have developed such a constraint solver that searches for an input solution satisfying the domain of the fault-detecting test equivalence class. Here the CSP variables are the observation variables of an OCL specification. The constraints are obtained by applying Theorem 4.3 to the original and mutated specification. If an input able to kill the mutant has been found, then the complete test case is produced by generating the expected (set of) output values. Note that constraint solving operates on finite domains. Hence, in case the tool cannot find a test case it is unknown if the mutant refines the original or if a fault outside the search space exists. We say that the mutant refines the original specification in the context of the finite variable domains.

In order to compare our fault-based testing approach to more conventional techniques, the tool is also able to generate test cases using DNF partitioning. In this classical testing strategy, first, the disjunctive normal form (DNF) of a formal specification is generated and then, one representative test case from each disjunct is selected [DF93].

The tool is able to generate test cases for the triangle example above (Example 1.1).

```
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. 2. A triangle with mutated if-statements

```
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. 3. Two more mutations of the triangle specification

Example 4.1 The specification of Fig. 1 can be mutated in several ways. In addition to variable name mutations as shown in Fig. 1, a designer might get the order of the nested *if-statements* wrong. This is modelled in Fig. 2. The two test cases generated by the tool are

$$a = 1,\ b = 2,\ c = 2,\ result = \text{“}isosceles\text{”}$$

for the mutant in Fig. 1 and

$$a = 1,\ b = 1,\ c = 1,\ result = \text{“}equilateral\text{”}$$

for the mutant in Fig. 2. One can easily see that each test case is able to distinguish its mutant from the original, since the mutants would produce different results. Hence, these test cases are sufficient to detect such faults in any implementation of `Ttype`.

Alternatively, by choosing the DNF partitioning strategy the tool returns five test cases, one for each partition. Note that the tool partitions the isosceles case into three cases:

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

Analysing these test cases generated by the DNF partitioning strategy one observes that the five test cases are also able to detect the faults presented in Fig. 2. Therefore, one could argue that the fault-based test cases do not add further value. However, in general DNF partitioning may not detect all possible faults. Consider the two additional mutated specifications shown in Fig. 3. One can easily see that the five DNF test cases are not able to reveal these faults, but the fault-based strategy generates precisely the following test cases that are needed to reveal the faults in Fig. 3:

$$a = 2,\ b = 2,\ c = 2,\ result = \text{“}equilateral\text{”}$$

covers the left hand mutant, and

$$a = 3,\ b = 2,\ c = 4,\ result = \text{“}scalene\text{”}$$

covers the mutant on the right hand side. It is also possible to integrate the DNF approach and ask the tool

to generate all fault-based test cases for every domain partition. Then, the additional test case

$a = 1,\ b = 3,\ c = 3,\ result = \text{“}isosceles\text{”}$

for the mutant on the right hand side is returned as well. □

This example, although trivial, demonstrates the automation of our 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. It should be added that the test case generator also helps in understanding the specification. Experimenting with different mutations and generating fault-detecting test cases for them is a valuable vehicle for validation. For further details of the tool's search algorithm we refer to [AS05].

5. Testing for program faults

So far our discussion on testing has focused on the semantic 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 [HH98]). 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:

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

The semantics of the operators follows the definitions in Sect. 2.3. The recursive statement using the least fix-point operator μ will be discussed separately in Sect. 5.4.

5.1. 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 study properties of existing semantics given by equations.

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 does not lead to an (observable) fault. This solves the problem of equivalent mutants in mutation testing. Furthermore, the normal form will be used to derive test equivalence classes on a purely algebraic (syntactic) basis. Our normal form has been designed for this purpose: In contrast to the normal form in [HH98], we push the conditions outwards. The proofs of the new laws are given in the Appendix. The following assignment normal form is taken from [HH98].

Definition 5.1 (*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 and h denote lists 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

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

where $h(g)$ is calculated by substituting the expressions in g for the corresponding variables in v (see [HH98]).

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

Theorem 5.1 (Conditional Elimination)

$$(P \lhd c \rhd Q) = (c \wedge P) \sqcap (\neg c \wedge Q)$$

Proof. By definition of conditional and non-deterministic choice. □

With this elimination rule at hand we are able to define a non-deterministic normal form.

Definition 5.2 (*Non-deterministic Normal Form*) A *non-deterministic normal form* is defined to be a non-deterministic choice of guarded total assignments.

$$(g_1 \wedge v := f) \sqcap (g_2 \wedge v := g) \sqcap \cdots \sqcap (g_n \wedge v := h)$$

with g_i being conditions such that

$$(g_1 \vee \cdots \vee g_n) = \textbf{true}$$

and ok, ok' do not occur in g_i. 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

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

The easiest operators to eliminate is disjunction itself (see [HH98])

$$\left(\bigsqcap A\right) \sqcap \left(\bigsqcap B\right) = \bigsqcap(A \cup B)\} \quad \textbf{(L2)}$$

and the conditional

$$\left(\bigsqcap A\right) \lhd d \rhd \left(\bigsqcap B\right) = \left(\bigsqcap\{((d \wedge b) \wedge P) \mid (b \wedge P) \in A\}\right) \sqcap \left(\bigsqcap\{((\neg d \wedge c) \wedge Q) \mid (c \wedge Q) \in B\}\right) \quad \textbf{(L3)}$$

is eliminated by splitting each guarded assignment in two cases (proof in Appendix). Note, this law is the only one introducing non-trivial (unequal true) guards when transforming a program to normal form. Hence, all non-trivial guards appear in their non-negated and negated form, ensuring that the disjunction of all guards is true.

Sequential composition is reduced by

$$\left(\bigsqcap A\right);\ \left(\bigsqcap B\right) = \bigsqcap\{(b(v) \wedge c(f(v))) \wedge v := f;\ v := g \mid (b \wedge v := f) \in A \wedge (c \wedge v := g) \in B\} \quad \textbf{(L4)}$$

Here, all non-deterministic combinations of sequential composition are formed (proof in Appendix).

The following lemma shows that our non-deterministic normal form is a design, given the fact that total assignments are designs.

Lemma 5.1

$$\bigsqcap_i (g_i \wedge (p_i \vdash Q_i)) = \left(\bigwedge_i (g_i \Rightarrow p_i)\right) \vdash \left(\bigvee_i (g_i \wedge Q_i)\right), \quad \text{provided } \bigvee_i g_i = true$$

Proof. See Appendix. □

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

$$b \lor P$$

where b is a condition for non-termination not containing ok, ok', and P a non-deterministic normal form. □

Any previous normal form P that terminates can be expressed as

$$\textbf{false} \lor P$$

and the constant **true** as

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

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

$$(b \lor P) \sqcap (c \lor Q) = (b \lor c) \lor (P \sqcap Q) \quad \textbf{(L5)}$$

$$(b \lor P) \lhd d \rhd (c \lor Q) = ((b \land d) \lor (c \land \neg d)) \lor (P \lhd d \rhd Q) \quad \textbf{(L6)}$$

$$(b \lor P);\ (c \lor Q) = (b \lor (P;\ c)) \lor (P;\ Q) \quad \textbf{(L7)}$$

(Laws **L5** and **L7** are taken from [HH98]; The proof of Law **L6** is in the Appendix.) 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 (proof in Appendix).

$$\left(\bigsqcap A\right);\ c = \bigvee\{(g \land (P;\ c)) \mid (g \land P) \in A\} \quad \textbf{(L8)}$$

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

$$b \lor \bigsqcap_i \{(g_i \land v := e_i) \mid 1 \le i \le n\}$$

The following lemmas show that our non-termination normal form is a design, given the fact that the non-deterministic normal form is a design

Lemma 5.2

$$b \lor (p \vdash Q) = (\neg b \land p) \vdash Q$$

Proof. See Appendix. □

Lemma 5.3

$$b \lor \bigsqcap_i \{(g_i \land v := e_i) \mid 1 \le i \le n\} = \left(\neg b \land \bigwedge_i (g_i \Rightarrow wf(e_i))\right) \vdash \left(\bigvee_i (g_i \land v' = e_i)\right), \text{ provided } \bigvee_i g_i = true$$

Proof. Follows directly from Lemma 5.1 and Lemma 5.2 and the definition of assignments. □

Next, it is shown how this normal form facilitates the generation of fault-detecting test cases. The technique is to introduce faults into the normal form and then search for test cases that are able to detect these faults.

5.2. Introducing faults

In the discussion so far, we have always assumed that faults are observable, i.e. $D \not\sqsubseteq D^m$. However, a well-known practical problem is the introduction of such faults that do not lead to refinement. In mutation testing of programs this is called the problem of equivalent mutants.

The problem of equivalent mutants can be simplified by reducing any non-recursive program into our finite normal form. More precisely, both the original program and the mutated one (the mutant) are transformed into normal form. Then, refinement can be checked by the following laws.

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 (see [HH98]).

$$(v := g) = (v := h) \quad \textbf{iff} \quad [g = h] \tag{L9}$$

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

$$R \sqsubseteq \left(\bigsqcap A\right) \quad \textbf{iff} \quad \forall P : P \in A \bullet (R \sqsubseteq P) \tag{L10}$$

$$((g_1 \wedge P_1) \sqcap \cdots \sqcap (g_n \wedge P_n)) \sqsubseteq (b \wedge Q) \quad \textbf{iff} \quad [\exists i \bullet ((g_i \wedge P_i) \Leftarrow (b \wedge Q))] \tag{L11}$$

$$[(g \wedge v := f) \Leftarrow (b \wedge v := h)] \quad \textbf{iff} \quad [b \Rightarrow (g \wedge (f = h))] \tag{L12}$$

The first law (see [HH98]) enables a non-deterministic normal form to be split into its component guarded assignments, which are then decided individually by the second law (proofs of **L11**, **L12** in Appendix). Note that **L12** is not decidable, in general. However, a combination of symbolic simplifiers and constraint solvers may deal with this reduced problem in practice.

Example 5.1 Consider the following example of a program *Min* for computing the minimum of two numbers.

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

In mutation testing, the assumption is made that programmers make small errors. A common error is to mix operators. The mutant Min_1 models such an error.

$$Min_1 \;=_{df}\; z := x \lhd x \boxed{\geq} y \rhd z := y$$

By means of the normal form it is now possible to show that this mutation represents a fault. Thus, we have to prove that

$$Min \not\sqsubseteq Min_1$$

Proof. In the following derivations, we will skip trivial simplification steps.

$$\begin{aligned} Min &= x, y, z := x, y, x \lhd x \leq y \rhd x, y, z := x, y, y && \{\text{adding identity assignments}\} \\ &= ((x \leq y) \wedge x, y, z := x, y, x) \sqcap \neg(x \leq y) \wedge x, y, z := x, y, y) && \{\text{by } \textbf{L3}\} \end{aligned}$$

Next, we reduce Min_1 to normal form

$$\begin{aligned} Min_1 &= x, y, z := x, y, x \lhd x \geq y \rhd x, y, z := x, y, y && \{\text{adding identity assignments}\} \\ &= ((x \geq y) \wedge x, y, z := x, y, x) \sqcap (\neg(x \geq y) \wedge x, y, z := x, y, y) && \{\text{by } \textbf{L3}\} \end{aligned}$$

Assume $Min \sqsubseteq Min_1$ then according to **L10** we must show that the two refinements hold

$$((x \leq y) \wedge x, y, z := x, y, x) \sqcap \neg(x \leq y) \wedge x, y, z := x, y, y) \sqsubseteq (x \geq y) \wedge x, y, z := x, y, x \tag{Case 1}$$

$$((x \leq y) \wedge x, y, z := x, y, x) \sqcap \neg(x \leq y) \wedge x, y, z := x, y, y) \sqsubseteq \neg(x \geq y) \wedge x, y, z := x, y, y \tag{Case 2}$$

We start checking the cases with law **L11** and **L12**.

$$\begin{aligned} \text{Case 1 } \textbf{iff } & [((x \leq y \wedge (x, y, x) := (x, y, x)) \Leftarrow (x \geq y \wedge (x, y, x) := (x, y, x))) && \{\text{by } \textbf{L11}\} \\ & \quad \vee \\ & \quad ((\neg(x \leq y) \wedge (x, y, x) := (x, y, y)) \Leftarrow (x \geq y \wedge (x, y, x) := (x, y, y)))] \\ & = [(x \geq y \Rightarrow (x \leq y \wedge \mathit{true})) && \{\text{by } \textbf{L12}\} \\ & \quad \vee \\ & \quad (x \geq y \Rightarrow (x > y \wedge x = y))] \\ & = [(x \geq y \Rightarrow x \leq y) \vee (x \geq y \Rightarrow \mathit{false})] \\ & = [x \leq y \vee x < y] \\ & = \mathit{false} \end{aligned}$$

It follows that refinement does not hold and that the mutation introduces an observable fault. □

The next example demonstrates the detection of an equivalent mutant.

Example 5.2 Consider again the program *Min* for computing the minimum of two numbers of Example 5.1. Another mutation regarding the comparison operator is produced

$$Min_2 \;=_{df}\; z := x \lhd x \;\boxed{<}\; y \rhd z := y$$

By means of normal form reduction it is now possible to show that this mutation does not represent a fault. Thus, we show that

$$Min \sqsubseteq Min_2$$

Proof. Since the normal form of *Min* has already been computed, we start with normalising Min_2.

$$\begin{array}{lcll} Min_2 & = & x, y, z := x, y, x \lhd x < y \rhd x, y, z := x, y, y & \{\text{adding identity assignments}\} \\ & = & ((x < y) \wedge x, y, z := x, y, x) \sqcap (\neg(x < y) \wedge x, y, z := x, y, y) & \{\text{by } \mathbf{L3}\} \end{array}$$

Again, two refinements must hold according to **L10**

$$((x \leq y) \wedge x, y, z := x, y, x) \sqcap \neg(x \leq y) \wedge x, y, z := x, y, y) \sqsubseteq (x < y) \wedge x, y, z := x, y, x \qquad \text{(Case 1)}$$

$$((x \leq y) \wedge x, y, z := x, y, x) \sqcap \neg(x \leq y) \wedge x, y, z := x, y, y) \sqsubseteq \neg(x < y) \wedge x, y, z := x, y, y \qquad \text{(Case 2)}$$

We check the cases

$$\begin{array}{lll} \text{Case 1 } \mathbf{iff} & [((x \leq y \wedge (x, y, x) := (x, y, x)) \Leftarrow ((x < y) \wedge x, y, z := x, y, x)) & \{\text{by } \mathbf{L11}\} \\ & \quad \vee & \\ & \quad ((\neg(x \leq y) \wedge (x, y, x) := (x, y, y)) \Leftarrow ((x < y) \wedge x, y, z := x, y, x))] & \\ & = [x < y \Rrightarrow x \leq y & \{\text{by } \mathbf{L12}\} \\ & \quad \vee & \\ & \quad x < y \Rrightarrow (x > y \wedge x = y)] & \\ & = [x \geq y \vee x \leq y & \\ & \quad \vee & \\ & \quad x \geq y \vee \mathit{false}] & \\ & = [\mathit{true} \vee x \geq y \vee \mathit{false}] & \\ & = \mathit{true} & \end{array}$$

$$\begin{array}{lll} \text{Case 2 } \mathbf{iff} & [((x \leq y \wedge (x, y, x) := (x, y, x)) \Leftarrow (\neg(x < y) \wedge x, y, z := x, y, y)) & \{\text{by } \mathbf{L11}\} \\ & \quad \vee & \\ & \quad ((\neg(x \leq y) \wedge (x, y, x) := (x, y, y)) \Leftarrow (\neg(x < y) \wedge x, y, z := x, y, y))] & \\ & = [x \geq y \Rrightarrow (x \leq y \wedge x = y) & \{\text{by } \mathbf{L12}\} \\ & \quad \vee & \\ & \quad x \geq y \Rrightarrow x > y] & \\ & = [x < y \vee x = y & \{\text{by } \mathbf{L12}\} \\ & \quad \vee & \\ & \quad x < y \vee x > y] & \\ & = [x \leq y \vee x > y] & \\ & = \mathit{true} & \end{array}$$

Since, both cases are true, we have refinement and the error made, represented by the mutation, cannot be detected. Such, mutations must be excluded from the fault-detecting test case generation process. □

These examples demonstrate how normal forms can be used to exclude equivalent mutants from the test case generation process. In the following, we are going to extend the laws to cover non-termination as well.

For non-termination normal form the laws for testing the refinement are

$$(c \vee Q) \sqsubseteq (b \vee P) \quad \textbf{iff} \quad [b \Rightarrow c] \text{ and } (c \vee Q) \sqsubseteq P \qquad \textbf{(L13)}$$
$$(c \vee (g_1 \wedge P_1) \sqcap \cdots \sqcap (g_n \wedge P_n)) \sqsubseteq (b \wedge Q) \quad \textbf{iff} \quad [c \vee (\exists\, i \bullet (g_i \wedge P_i) \Leftarrow (b \wedge Q))] \qquad \textbf{(L14)}$$

(Law **L13** is taken from [HH98]; proof of **L14** in Appendix). Again an example serves to illustrate the rules for non-termination.

Example 5.3 Let us again consider the simple problem of returning the minimum of two numbers. If both inputs are natural numbers, the following program computes the minimum of x, y in x.

$$MinNat =_{df} (x < 0 \vee y < 0) \vee (x := x \lhd (x - y) < 0 \rhd x := y)$$

First, an equivalent mutant is produced that can be detected by a derivation on the normal form

$$MinNat_1 =_{df} (x < 0 \vee y < 0) \vee (x := x \lhd (x - y) < \boxed{1} \rhd x := y)$$

Proof. First, both normal forms are derived.

$$\begin{array}{rcl} MinNat & = & (x < 0 \vee y < 0) \vee ((x, y := x, y) \lhd (x - y) < 0 \rhd (x, y := y, y)) \\ & = & (x < 0 \vee y < 0) \vee (((x - y) < 0 \wedge x, y := x, y) \sqcap (\neg((x - y) < 0) \wedge x, y := y, y)) \\ MinNat_1 & = & (x < 0 \vee y < 0) \vee (((x - y) < 1 \wedge x, y := x, y) \sqcap (\neg((x - y) < 1) \wedge x, y := y, y)) \end{array}$$

Since, both have the same non-termination condition, we have to check according to law **L13** that

$$MinNat \sqsubseteq ((x - y) < 1 \wedge x, y := x, y) \sqcap (\neg((x - y) < 1) \wedge x, y := y, y)$$

According to law **L10** we have to show two refinements

$$MinNat \sqsubseteq ((x - y) < 1 \wedge x, y := x, y) \qquad \text{(Case 1)}$$
$$MinNat \sqsubseteq (\neg((x - y) < 1) \wedge x, y := y, y) \qquad \text{(Case 2)}$$

We verify the cases

$$\begin{array}{lll} \text{Case 1 } \textbf{iff} & [(x < 0 \vee y < 0) & \{\text{by } \textbf{L14}\} \\ & \quad \vee & \\ & \quad (((x - y) < 0 \wedge x, y := x, y) \Leftarrow ((x - y) < 1 \wedge x, y := x, y)) & \\ & \quad \vee & \\ & \quad ((\neg((x - y) < 0) \wedge x, y := y, y) \Leftarrow ((x - y) < 1 \wedge x, y := x, y))] & \\ & = [(x < 0 \vee y < 0) & \{\text{by } \textbf{L12}\} \\ & \quad \vee & \\ & \quad ((x - y) < 1) \Rightarrow (x - y < 0 \wedge \mathit{true}) & \\ & \quad \vee & \\ & \quad ((x - y) < 1) \Rightarrow (x - y \geq 0 \wedge x = y)] & \\ & = [(x < 0 \vee y < 0) & \{\text{by } \textbf{L12}\} \\ & \quad \vee & \\ & \quad ((x - y) \geq 1) \vee (x - y < 0) & \\ & \quad \vee & \\ & \quad ((x - y) \geq 1) \vee x = y] & \\ & = [(x < 0 \vee y < 0) & \\ & \quad \vee & \\ & \quad x > y \vee x < y \vee x = y] & \\ & = \mathit{true} & \end{array}$$

The fact that Case 2 holds can be shown by a similar derivation. □

It has been shown that the presented refinement laws can be used to automatically detect equivalent mutants for non-recursive programs. Next, test case generation is discussed.

5.3. Test case generation

The presented normal form has been developed to facilitate the automatic generation of test cases that are able to detect anticipated faults. Above, it has been demonstrated that algebraic refinement laws solve the problem of equivalent mutants that have an alternation not representing a fault. The above laws also build the foundation of our test case generation process. The following theorem defines the test equivalence class that will detect an error.

Theorem 5.2 Let $P = (p \vdash Q)$ be a program and $P^m = (p^m \vdash Q^m)$ a faulty mutation of this program with normal forms as follows

$$P = c \lor \prod_j \{(a_j \land v := f_j) \mid 1 \leq j \leq m\}$$
$$P^m = c^m \lor \prod_k \{(b_k \land v := h_k) \mid 1 \leq k \leq n\}$$

For simplicity, we assume the well-formedness of the total assignments ($wf(f_j) = wf(h_k) = true$). Then, every representative test case of the test equivalence class

$$T_\sim \;=_{df}\; d_\bot;\; P\,, \quad \text{with } d = (\neg c \land c^m) \lor \bigvee_k (\neg c \land b_k \land \bigwedge_j (\neg a_j \lor (f_j \neq h_k)))$$

is able to detect the fault in D^m.

Before presenting the formal proof, we present an informal explanation: 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.

1. Those test cases have to be added where the mutant does not terminate, but the original program does. That is when $(\neg c \land c^m)$ holds.
2. In the terminating case, by the two laws **L10** and **L11**, 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 **L12** tells us that refinement between two guarded commands holds iff $[b_k \;\Rightarrow\; (a_j \land (f_j = h_k))]$. Negating this gives $\exists\, v, v' \bullet b_k \land (\neg a_j \lor (f_j \neq h_k))$. Since we are only interested in test cases that terminate, 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 ($\bigvee_k$).

Proof. The formal proof uses Theorem 4.3 to derive the test domain $d^{12} = d_1 \lor d_2$. This is possible, since according to Lemma 5.3 the normal forms represent designs: the postcondition is a non-deterministic choice of assignments (restricted by guards), and the preconditions are the negated non-termination conditions c and c^m.

$$\begin{aligned} d_1 &= \quad \{\text{by Theorem 4.3}\} \\ &\quad\; \neg p^m \\ &= \quad \{\text{by Lemma 5.3}\} \\ &\quad\; \neg(\neg c^m \land \textstyle\bigwedge_k (b_k \Rightarrow wf(h_k)))) \\ &= \quad \{\text{by assumption that } wf(h_k) = true\} \\ &\quad\; c^m \end{aligned}$$

$$
\begin{array}{ll}
d_2 = & \{\text{by Theorem 4.3}\} \\
& \exists\, v' \bullet (Q^m \wedge \neg Q) \\
= & \{\text{by Lemma 5.3}\} \\
& \exists\, v' \bullet ((\bigvee_k (b_k \wedge v' = h_k)) \wedge \neg(\bigvee_j (a_j \wedge v' = f_j))) \\
= & \{\text{by de Morgan's law}\} \\
& \exists\, v' \bullet ((\bigvee_k (b_k \wedge v' = h_k)) \wedge \bigwedge_j \neg(a_j \wedge v' = f_j)) \\
= & \{\text{by predicate calculus}\} \\
& \exists\, v' \bullet (\bigvee_k (b_k \wedge v' = h_k \wedge \bigwedge_j \neg(a_j \wedge v' = f_j))) \\
= & \{\text{by predicate calculus}\} \\
& \exists\, v' \bullet (\bigvee_k (b_k \wedge \bigwedge_j (v' = h_k \wedge \neg(a_j \wedge v' = f_j)))) \\
= & \{\text{by de Morgan's law}\} \\
& \exists\, v' \bullet (\bigvee_k (b_k \wedge \bigwedge_j (v' = h_k \wedge (\neg a_j \vee v' \neq f_j)))) \\
= & \{\text{by distributive law}\} \\
& \exists\, v' \bullet (\bigvee_k (b_k \wedge \bigwedge_j ((v' = h_k \wedge \neg a_j) \vee (v' = h_k \wedge v' \neq f_j)))) \\
= & \{\text{by equality of } v' \text{ and } h_k\} \\
& \exists\, v' \bullet (\bigvee_k (b_k \wedge \bigwedge_j ((v' = h_k \wedge \neg a_j) \vee (v' = h_k \wedge h_k \neq f_j)))) \\
= & \{\text{by simplification}\} \\
& \exists\, v' \bullet (\bigvee_k (b_k \wedge v' = h_k \wedge \bigwedge_j (\neg a_j \vee h_k \neq f_j))) \\
= & \{\text{by one point rule of predicate calculus}\} \\
& \bigvee_k (b_k \wedge \bigwedge_j (\neg a_j \vee h_k \neq f_j))
\end{array}
$$

Thus,

$$T_\sim = d^{12}_\perp;\ P \ =\ d^{12}_\perp;\ (c \vee Q) \ =\ (d^{12} \wedge \neg c)_\perp;\ P \ =\ ((\neg c \wedge d_1) \vee (\neg c \wedge d_2))_\perp;\ P \ =\ d;\ P$$

The last derivation on the test equivalence class shows that the test domain can be safely strengthened by $\neg c$ due to the termination condition c in P. □

Note, that in case of true non-determinism, which means some guards are **true**, detection of the errors can only happen if the faulty part is chosen to be executed. Since, by definition of non-determinism a tester has no means to influence this decision, it may go undetected for a while. However, under the assumption of a fair selection policy, the fault will eventually be detected. Thus, when we say a test case (or its equivalence class) will detect an error, we really mean it is able to do so over a period of time.

Example 5.4 Consider the program and its mutant in Example 5.1. According to Theorem 5.2 we have the fault-detecting domain

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

Note that the case where $x = y$ has been correctly excluded from the domain of the test equivalence class, since it is unable to distinguish the two versions of the program. □

5.4. 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 5.3 (Kleene) If F is continuous then

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

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

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 5.4 (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

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

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 [HH98] tells us that we can decompose the problem.

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

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 computations. Such a decision represents a test hypothesis (i.e. a regularity hypothesis according to [GJ98]), where the tester assumes, that if n iterations did not reveal a fault, an equivalent mutant has been produced.

An example shall illustrate the approximation.

Example 5.5 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:

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

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.

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

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

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

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.

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

The third approximation describes MIN up to two iterations, leading to more choices.

$$
\begin{aligned}
S^3 \;=_{df}\; F(S^2) \;=\; & (k+1 < n) \vee \\
& \quad (((k+1 = n \wedge A[k] < A[t] \wedge A[k+1] < A[t]) \wedge (k, t := k+2, k+1)) \\
& \quad \sqcap \\
& \quad ((k+1 = n \wedge A[k] < A[t] \wedge A[k+1] \ge A[t]) \wedge (k, t := k+2, k)) \\
& \quad \sqcap \\
& \quad ((k+1 = n \wedge A[k] \ge A[t] \wedge A[k+1] < A[t]) \wedge (k, t := k+2, k+1)) \\
& \quad \sqcap \\
& \quad ((k+1 = n \wedge A[k] \ge A[t] \wedge A[k+1] \ge A[t]) \wedge (k, t := k+2, t)) \\
& \quad \sqcap \\
& \quad ((k = n \wedge A[k] < A[t]) \wedge (k, t := k+1, k)) \\
& \quad \sqcap \\
& \quad ((k = n \wedge A[k] \ge A[t]) \wedge (k, t := k+1, t)) \\
& \quad \sqcap \\
& \quad ((k > n) \wedge k, t := k, t)))
\end{aligned}
$$

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:

$$MIN_1 \;=_{df}\; k := 2;\;\; t := 1;\;\; \mu\, X \bullet ((B;\;\; X) \lhd k \;\boxed{<}\; n \rhd k, t := k, t)$$

Its first approximation gives

$$S_1^1 \;=_{df}\; F(\mathbf{true}) \;=\; (k < n) \;\vee ((k \ge n) \wedge k, t := k, t)$$

By applying Theorem 5.2 to find test cases that can distinguish the two first approximations, we realise that such a test case does not exist, because $S^1 \;\sqsubseteq\; S_1^1$. The calculation of the test equivalence class domain predicate d^1 gives $false$:

$$
\begin{aligned}
d^1 \;=\; & \quad \{\text{by Theorem 5.2 }\} \\
& (\neg(k \le n) \wedge k < n) \;\vee\; (\neg(k \le n) \wedge k \ge n \wedge (\neg(k > n) \vee false)) \\
= \; & \\
& false \vee false \;=\; false
\end{aligned}
$$

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

$$
\begin{aligned}
S_1^2 \;=_{df}\; F(S_1) \;=\; & (k+1<n) \lor \\
& (((k+1=n \land A[k]<A[t]) \land (k,t := k+1,k)) \\
& \sqcap \\
& ((k+1=n \land A[k] \geq A[t]) \land (k,t := k+1,t)) \\
& \sqcap \\
& ((k \geq n) \land k,t := k,t))
\end{aligned}
$$

This time test cases exist. By applying Theorem 5.2 we get the test equivalence class that can find the error.

$$
\begin{aligned}
d(k,t) \;=\; & \quad \{\text{by Theorem 5.2}\} \\
& (\neg(k \leq n) \land k < n) \\
& \lor \\
& (k \geq n \land k+1 = n \land A[k] < A[t] \land \ldots \\
& \lor \\
& (k \geq n \land k+1 = n \land A[k] \geq A[t] \land \ldots \\
& \lor \\
& (k \geq n \land k \geq n \\
& \quad \land (\neg(k = n \land A[k] < A[t]) \lor \mathit{true}) \\
& \quad \land (\neg(k = n \land A[k] \geq A[t]) \lor \mathit{true}) \\
& \quad \land (\neg(k > n) \lor \mathit{false})) \\
= & \\
& \mathit{false} \\
& \lor \\
& (k \geq n \land k \leq n) \\
= & \\
& (k = n)
\end{aligned}
$$

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

$$T_{\sim 1} \;=\; (n=2)_{\perp};\; \mathit{MIN}$$

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 designed.

Note also that the test equivalence class $T_{\sim 1}$ is just an approximation of the complete test equivalence class. More precisely, it has to be an approximation, since the complete test equivalence class is infinite. Next we investigate an error, where the programmer forgets to increase the index variable k.

$$
\begin{aligned}
\mathit{MIN}_2 \;&=_{df}\; k := 2;\;\; t := 1;\;\; \mu\, X \bullet \Big(\big(\boxed{B_2} \;;\;\; X \big) \lhd k \leq n \rhd k, t := k, t \Big) \\
B_2 \;&=_{df}\; (t := k;\;\; k := k+1) \lhd A[k] < A[t] \rhd k := \boxed{k}
\end{aligned}
$$

Obviously, $S^1 = S_2^1$ since the error has been made inside the loop. Therefore, immediately the second approximation of the mutant S_2^2 is presented:

$$\begin{aligned} S_2^2 \ =_{df}\ & (k+1 \leq n \wedge A[k] < A[t]) \vee (((k \leq n \wedge k+1 > n \wedge A[k] < A[t]) \wedge (k, t := k+1, k)) \\ & \sqcap \\ & (k \leq n \wedge A[k] \geq A[t]) \vee ((k \leq n \wedge k > n \wedge A[k] \geq A[t]) \wedge (k, t := k, t)) \\ & \sqcap \\ & (\mathit{false}) \vee ((k > n) \wedge k, t := k, t) \end{aligned}$$

We see that the second case becomes infeasible (the guard equals false), and that consequently the non-termination condition is weakened:

$$\begin{aligned} S_2^2 \ = \ & (k < n \vee (k = n \wedge A[k] \geq A[t])) \vee \\ & (((k = n \wedge A[k] < A[t]) \wedge (k, t := k+1, k)) \\ & \sqcap \\ & ((k > n) \wedge k, t := k, t)) \end{aligned}$$

Clearly, a weaker non-termination condition leads to non-refinement. Therefore, Theorem 5.2 gives us for this case the test equivalence class representing the cases where MIN terminates and MIN_2 does not.

$$\begin{aligned} T_{\sim 2}(k, t) \ &= (k = n \wedge A[k] \geq A[t])_{\perp};\ MIN) \\ T_{\sim 2} \ &= (n = 2 \wedge A[2] \geq A[1])_{\perp};\ MIN) \end{aligned}$$

The calculated test cases are indeed those, where MIN_2 fails to terminate, due to the missing incrementation of k. □

The example demonstrated how to calculate test cases for detecting faulty designs even when recursion is present. However, in cases where refinement cannot be falsified, we have to stop the approximation process at a certain point. An upper limit n must be chosen by the tester, to determine how many approximation steps should be computed.

6. Conclusions

Summary. 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 [HH98]. It extends 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 test case generation methods could be developed.

The first test case generation method (Definition 4.1) is a general criterion for fault-detecting test cases. It is not completely new, but has been translated from our previous work [Aic03] 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). In [AD06] we applied this technique to labelled transition systems for testing web-servers. This demonstrates the generality of our refinement-based testing theory.

The second test case generation method (Theorem 4.3) 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. An alternative implementation technology would be SAT-solving as it is used in Daniel Jackson's Alloy Analyzer [Jac00].

The third approach (Theorem 5.2) 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.

Motivations for using UTP. UTP's aim is simplicity and our work definitely benefitted from its simple predicative semantics. Having implication as the refinement order made the theories simpler and the proofs shorter, than, e.g. using a weakest-precondition semantics. Furthermore, the relational design predicates can be directly fed into a constraint solving system, or a BDD checker. Most important, UTP's links to other programming paradigms, like, e.g. parallel programming or object-orientation, keep our testing theory open for extensions.

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 1970s [Ham77, DLS78]; since then it has found many applications and has become the major assessment technique in empirical studies on new test case selection techniques [Won01].

To our present knowledge Budd and Gopal were the first who mutated specifications [BG85]. They applied a set of mutation operators to specifications given in predicate calculus form.

Tai and Su [TS87] 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 [Tai96] 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.

Until a few years ago, most of the research on testing from formal specifications has widely ignored fault-based testing. The current approaches generate test cases according to the structural information of a model in a formal specification language, like for example VDM [DF93], Z, B [LPU02], or Lotos [GJ98]. Only few noticed the relevance of a fault-based strategy on the specification level.

Stocks applied mutation testing to Z specifications [Sto93]. In his work he extends mutation testing to model-based specification languages by defining a collection of mutation operators for Z's specification language. An example of his specification mutations is the exchange of the join operator $\cup$ of sets with intersection $\cap$. He presented the criteria to generate test cases to discriminate mutants, but did not automate his approach.

More recently, Simon Burton presented a similar technique as part of his test case generator for Z specifications [Bur00]. 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 formulating different testing strategies). This is in contrast to our implementation of the OCL test case generator, where Constraint Handling Rules [FA03] 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.

Fault-based testing has also been discovered by the security community. Wimmel and Jürjens [WJ02] use mutation testing on specifications to extract those interaction sequences that are most likely to find vulnerabilities. Here, mutants of an Autofocus model are generated. Then, a constraint solver is used to search for a test sequence that satisfies a mutated system model (a predicate over traces) and that does not satisfy a security requirement. If a test case able to kill the mutant can be found, then the mutation introduces a vulnerability and the test case shows how it can be exploited. Again, this approach is a special instantiation of our more general refinement technique supporting our proposal that a general theory of fault-based testing should be based on refinement.

Black et al. showed how model checkers (e.g. SMV) can be applied in mutation testing [BOY01]. There are basically two methods. The first is similar to our approach: The original and a mutated model are integrated as SMV models and then a temporal formula stipulates that the output variables of the models must always be equivalent. A counter-example in the form of a trace through the state space serves as the test case distinguishing the mutation. In the second method, the temporal formulas themselves are mutated.

In [FAW07] we have demonstrated how this technique can be extended and optimised for fast regression testing. The limitation of this model-checking approach is that it only works for deterministic models. In the general case of nondeterminism, tree-shaped test cases are needed as it is the case in tools like TGV [AD06]. Our general observation regarding testing and model checking is that in most cases the work lacks the support of a precise testing theory, e.g. the definition of a conformance relation.

A group in York has started to use fault-based techniques for validating their CSP models [Sri01, SCSP03]. Their aim is not to generate test cases, but to study the equivalent mutants. Their work demonstrates that semantic issues of complex concurrent models can be detected, by understanding why alternated (mutated) models are observationally equivalent. Their reported case study in the security domain indicates the relevance of fault-based testing in this area. Similar research is going on in Brazil with an emphasis on protocol specifications written in the Estelle language [SMFS99].

Our testing theory relies on the notion of refinement. Of course, the relation between testing and refinement is not completely new. Hennessy and de Nicola [DNH84] developed a testing theory that defines the equivalence and refinement of processes in terms of testers. Similarly, the failure-divergency refinement of CSP [Hoa85] is inspired by testing, since it is defined via the possible observations of a tester. Later, these theories led to Tretmans' work on conformance testing based on labelled transition systems [Tre92, Tre99]. They are the foundations of Peleska's work on testing, as well [PS96]. However, these theories do not focus on the use of abstraction (the reverse of refinement) in order to select a subset of test cases. Furthermore, these existing testing theories focus on verification. This restricts their use, either to the study of semantic issues (like the question of observable equivalence of processes [DNH84]), or to the testing of very abstract (finite) models for which exhaustive testing is feasible (like in protocol testing [Tre92]). In contrast, this work focuses on falsification.

It was Stepney in her work on Object-Z, who first promoted explicitly the use of abstraction for designing test cases [Ste95]. The application of a refinement calculus to define different test-selection strategies is a contribution of the first author's doctoral thesis [Aic01a]. It was in this thesis, where the idea of applying refinement to mutation testing has been presented the first time. Although others worked on specification-based mutation testing before, the use of a refinement relation and a normal form is completely new.

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. For example, the relationship between the abstraction in model checking and the abstraction techniques in test case selection deserves a careful study. A further research area where a unifying theory might be applied is the combination of testing and formal proofs. This is related to the highly controversial question discussed in philosophy of science, how theories can be confirmed by observations. Our next steps will be to include models of concurrency, work out the difference between a test case and a test configuration (the first is a kind of specification, the latter the synchronous product of a tester and a system under test), and to translate the previously obtained results on test sequencing to our new theory.

Another branch of future work is automation. We are currently working on extensions of the prototype test case generator discussed in Sect. 4.3. Especially, the proper sequencing of test cases for bringing a system into a target state has to be addressed. [LPU02] demonstrates the use of constraint solving to master this problem. Besides the study of algorithms for automation, another research agenda is language design. We believe that the design of future specification and programming languages will be highly influenced by tool support for static and dynamic analysis, including test case generation and automatic verification. Therefore, a careful study of the properties of programming languages with respect to automation will gain importance.

All in all, we believe this is a quite challenging and exciting area of research. The authors hope that the presented theory of fault-based testing will inspire the reader to new contributions to testing, and verification in general.

Acknowledgments

This research is being carried out as part of the EU funded research project in Framework 6: IST-33826 CREDO (Modelling and analysis of evolutionary structures for distributed services). The test case generator discussed in Sect. 4.3 was implemented by Percy Antonio Pari Salas during his fellowship at UNU-IIST. The authors wish to thank the three anonymous referees for their detailed and constructive feedback in order to improve the paper.

Appendix

In this appendix the proofs of the algebraic laws that have been newly introduced in Sect. 5 are presented. Those not listed below are taken from [HH98].

Proof of L3

$$
\begin{array}{lll}
(\bigsqcap A) \lhd d \rhd (\bigsqcap B) & = & \{\text{by L1, Sect. 5.2 in [HH98]}\} \\
 & & \bigsqcap\{(b \wedge P) \lhd d \rhd (c \wedge Q) \mid (b \wedge P) \in A \ \wedge (c \wedge Q) \in B\} \\
 & = & \{\text{by Theorem 5.1}\} \\
 & & \bigsqcap\{(d \wedge b \wedge P) \sqcap (\neg d \wedge c \wedge Q) \mid (b \wedge P) \in A \ \wedge (c \wedge Q) \in B\} \\
 & = & \{\text{by L2}\} \\
 & & \bigsqcap(\{((d \wedge b) \wedge P) \mid (b \wedge P) \in A\} \cup \{((\neg d \wedge c) \wedge Q) \mid (c \wedge Q) \in B\}) \\
 & = & \{\text{by L2}\} \\
 & & (\bigsqcap\{((d \wedge b) \wedge P) \mid (b \wedge P) \in A\}) \sqcap (\bigsqcap\{((\neg d \wedge c) \wedge Q) \mid (c \wedge Q) \in B\})
\end{array}
$$

Proof of L4

$$
\begin{array}{lll}
(\bigsqcap A)\,;\,(\bigsqcap B) & = & \{\text{by L2, Sect. 5.2 in [HH98]}\} \\
 & & \bigsqcap\{(b \wedge P);\ (c \wedge Q) \mid (b \wedge P) \in A \ \wedge (c \wedge Q) \in B\} \\
 & = & \{\text{by definition of normal form}\} \\
 & & \bigsqcap\{(b \wedge v := f);\ (c \wedge v := g) \mid (b \wedge v := f) \in A \ \wedge (c \wedge v := g) \in B\} \\
 & = & \{\text{by definition of :=}\} \\
 & & \bigsqcap\{(b \wedge v' = f(v));\ (c \wedge v' = g(v)) \mid (b \wedge v := f) \in A \ \wedge (c \wedge v := g) \in B\} \\
 & = & \{\text{by definition of ; }\} \\
 & & \bigsqcap\{(\exists\, v_0 \bullet (b(v) \wedge v_0 = f(v) \wedge c(v_0) \wedge v' = g(v_0))) \mid (b \wedge v := f) \in A \ \wedge (c \wedge v := g) \in B\} \\
 & = & \{\text{by equality}\} \\
 & & \bigsqcap\{(\exists\, v_0 \bullet (b(v) \wedge c(f(v)) \wedge v_0 = f(v) \wedge v' = g(v_0))) \mid (b \wedge v := f) \in A \ \wedge (c \wedge v := g) \in B\} \\
 & = & \{\text{by simplification}\} \\
 & & \bigsqcap\{(b(v) \wedge c(f(v)) \wedge \exists\, v_0 \bullet (v_0 = f(v) \wedge v' = g(v_0))) \mid (b \wedge v := f) \in A \ \wedge (c \wedge v := g) \in B\} \\
 & = & \{\text{by definition of ; }\} \\
 & & \bigsqcap\{(b(v) \wedge c(f(v)) \wedge (v := f;\ v = g)) \mid (b \wedge v := f) \in A \ \wedge (c \wedge v := g) \in B\}
\end{array}
$$

Proof of L6

$$
\begin{array}{lll}
(b \vee P) \lhd d \rhd (c \vee Q) & = & \{\text{by L7, Sect. 5.3 in [HH98]}\} \\
 & & (b \lhd d \rhd c) \vee (P \lhd d \rhd Q) \\
 & = & \{\text{by definition of conditional}\} \\
 & & ((b \wedge d) \vee (c \wedge \neg d)) \vee (P \lhd d \rhd Q)
\end{array}
$$

Proof of L8

$$
\begin{array}{lll}
(\bigsqcap A)\,;\ c & = & \{\text{by L4, Sect. 5.3 in [HH98]}\} \\
 & & \bigvee\{(g \wedge P);\ c) \mid (g \wedge P) \in A\} \\
 & = & \{g \text{ is a predicate over } v\} \\
 & & \bigvee\{(g \wedge (P;\ c)) \mid (g \wedge P) \in A\}
\end{array}
$$

Proof of L11

$$
\begin{array}{lll}
((g_1 \wedge P_1) \sqcap \cdots \sqcap (g_n \wedge P_n)) \sqsubseteq (b \wedge Q) & = & \{\text{by definitions of } \sqcap \text{ and } \sqsubseteq\} \\
 & & [((g_1 \wedge P_1) \vee \cdots \vee (g_n \wedge P_n) \Leftarrow (b \wedge Q)] \\
 & = & \{\text{by distribution of } \Leftarrow\} \\
 & & [((g_1 \wedge P_1) \Leftarrow (b \wedge Q)) \vee \cdots \vee (g_n \wedge P_n) \Leftarrow (b \wedge Q))] \\
 & = & \{\text{by definition of existential quantification}\} \\
 & & [\exists\, i \bullet ((g_i \wedge P_i) \Leftarrow (b \wedge Q))]
\end{array}
$$

Proof of L12

$$
\begin{aligned}
[(g \wedge v := f) \Leftarrow (b \wedge v := h)] = \quad & \{\text{by definition of total assignment}\} \\
& [(g \wedge v' = f) \Leftarrow (b \wedge v' = h)] \\
= \quad & \\
& [(g \Leftarrow (b \wedge v' = h)) \wedge (v' = f \Leftarrow (b \wedge v' = h))] \\
= \quad & \\
& [(g \Leftarrow b) \wedge (v' = f \Leftarrow v' = h) \Leftarrow b)] \\
= \quad & \\
& [(g \Leftarrow b) \wedge ((f = h) \Leftarrow b)] \\
= \quad & \\
& [(g \wedge (f = h)) \Leftarrow b]
\end{aligned}
$$

Proof of L14

L14 follows directly from L11.

Proof of Lemma 5.1

$$
\begin{aligned}
& \prod_i (g_i \wedge (p_i \vdash Q_i)) \\
= \quad & \{\text{by definitions of meet and design}\} \\
& \bigvee_i (g_i \wedge ((ok \wedge p_i) \Rightarrow (ok' \wedge Q_i))) \\
= \quad & \\
& \bigvee_i (g_i \wedge (\neg(ok \wedge p_i) \vee (ok' \wedge Q_i))) \\
= \quad & \\
& \bigvee_i ((g_i \wedge \neg ok) \vee (g_i \wedge \neg p_i) \vee (ok' \wedge g_i \wedge Q_i)) \\
= \quad & \\
& \bigvee_i (g_i \wedge \neg ok) \vee \bigvee_i (g_i \wedge \neg p_i) \vee \bigvee_i (ok' \wedge g_i \wedge Q_i) \\
= \quad & \\
& (\neg ok \wedge \bigvee_i g_i) \vee \bigvee_i (g_i \wedge \neg p_i) \vee (ok' \wedge \bigvee_i (g_i \wedge Q_i)) \\
= \quad & \{\text{by assumption } \bigvee_i g_i = true\} \\
& (\neg ok \vee \bigvee_i (g_i \wedge \neg p_i) \vee (ok' \wedge \bigvee_i (g_i \wedge Q_i)) \\
= \quad & \\
& \neg(ok \wedge \bigwedge_i (\neg g_i \vee p_i)) \vee (ok' \wedge \bigvee_i (g_i \wedge Q_i)) \\
= \quad & \{\text{by definition of design}\} \\
& \left(\bigwedge_i (g_i \Rightarrow p_i)\right) \vdash \left(\bigvee_i (g_i \wedge Q_i)\right)
\end{aligned}
$$

Proof of Lemma 5.2

$$
\begin{aligned}
& b \vee (p \vdash Q) \\
& \quad = \{\text{by definition of design}\} \\
& \qquad b \vee ((ok \wedge p) \Rightarrow (ok' \wedge Q)) \\
& \quad = \\
& \qquad \neg b \Rightarrow ((ok \wedge p) \Rightarrow (ok' \wedge Q)) \\
& \quad = \\
& \qquad (ok \wedge \neg b \wedge p) \Rightarrow (ok' \wedge Q) \\
& \quad = \{\text{by definition of design}\} \\
& \qquad (\neg b \wedge p) \vdash Q
\end{aligned}
$$

References

[Abr96] Abrial J-R (1996) The B Book, assigning programs to meanings. Cambridge University Press, Cambridge

[AD06] Aichernig BK, Delgado CC (2006) From faults via test purposes to test cases: on the fault-based testing of concurrent systems. In: Baresi L, Heckel R (eds) Proceedings of FASE'06, Fundamental Approaches to Software Engineering, Vienna, Austria, March 27–29, 2006. Lecture Notes in Computer Science, vol 3922. Springer, Berlin, pp 324–338

[Aic01a] Aichernig BK (2001) Systematic black-box testing of computer-based systems through formal abstraction techniques. Ph.D. thesis, Institute for Software Technology, TU Graz, Austria, January 2001 Supervisor: Peter Lucas

[Aic01b] Aichernig BK (2001) Test-design through abstraction: a systematic approach based on the refinement calculus. J Universal Comput Sci 7(8):710–735

[Aic03] Aichernig BK (2003) Mutation testing in the refinement calculus. Formal Asp Comput J 15(2):280–295

[AS05] Aichernig BK, Antonio Pari Salas P (2005) Test case generation by OCL mutation and constraint solving. In: Cai K-Y, Ohnishi A, Lau MF (eds) QSIC 2005, Proceedings of the Fifth International Conference on Quality Software, Melbourne, Australia, September 19–21, 2005. IEEE Computer Society Press, New York, pp 64–71

[BBH02] Benattou M, Bruel J-M, Hameurlain N (2007) Generating Test Data from OCL Specifications (position paper). In: ECOOP'2002 Workshop on Integration and Transformation of UML Models (WITUML02), Malaga, Spain, June 2002. http://ctp.di.fct.unl.pt/~ja/wituml/bruel.pdf (last visited Sep. 1st, 2007)

[Bei90] Beizer B (1990) Software testing techniques, 2nd edn. Van Nostrand Reinhold, New York

[BG85] Budd TA, Gopal AS (1985) Program testing by specification mutation. Comput Lang 10(1):63–73

[BOY01] Black PE, Okun V, Yesha Y (2001) Mutation of model checker specifications for test generation and evaluation. In: Mutation testing for the new century. Kluwer, Dordrecht, pp 14–20

[Bur00] Burton S (2000) Automated testing from Z specifications. Technical report YCS 329, Department of Computer Science University of York

[BvW98] Back R-J, von Wright J (1998) Refinement Calculus, a Systematic Introduction. Graduate Texts in Computer Science. Springer, Berlin

[DF93] Dick J, Faivre A (1993) Automating the generation and sequencing of test cases from model-based specifications. In: Woodcock JCP, Larsen PG (eds) Proceedings of FME'93: Industrial-Strength Formal Methods, International Symposium of Formal Methods Europe, April 1993, Odense, Denmark. Springer, Berlin, pp 268–284

[DLS78] DeMillo R, Lipton R, Sayward F (1978) Hints on test data selection: Help for the practicing programmer. IEEE Comput 11(4):34–41

[DNH84] De Nicola R, Hennessy MCB (1984) Testing equivalences for processes. Theor Comput Sci 34:83–133

[FA03] Frühwirth T, Abdennadher S (2003) Essentials of constraint programming. Springer, Berlin

[FAW07] Fraser G, Aichernig BK, Wotawa F (2007) Handling model changes: Regression testing and test-suite update with model-checkers. Electronic Notes in Theoretical Computer Science, vol 190. Elsevier, Amsterdam, pp 33–46

[Gau95] Gaudel M-C (1995) Testing can be formal, too. In: TAPSOFT '95: Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development. Springer, Berlin, pp 82–96

[GJ98] Gaudel M-C, James PR (1998) Testing algebraic data types and processes: a unifying theory. Formal Asp Comput 10(5, 6):436–451

[Ham77] Hamlet RG (1977) Testing programs with the aid of a compiler. IEEE Trans Softw Eng 3(4):279–290

[HH98] Hoare CAR, He J (1998) Unifying theories of programming. Prentice-Hall, Englewood Cliffs, NJ

[HNS97] Helke S, Neustupny T, Santen T (1997) Automating test case generation from Z specifications with Isabelle. In: Bowen JP, Hinchey MG, Till D (eds) Proceedings of ZUM'97, the 10th International Conference of Z Users, April 1997, Reading, UK. Lecture Notes in Computer Science, vol 1212. Springer, Berlin, pp 52–71

[Hoa85] Hoare CAR (1985) Communicating sequential processes. International Series in Computer Science. Prentice-Hall, Englewood Cliffs, NJ

[Jac00] Jackson D (2000) Automating first-order relational logic. In: Proceedings of SIGSOFT FSE 2000: ACM SIGSOFT Symposium on Foundations of Software Engineering, November 6–10, 2000, San Diego, California, USA. ACM, New York, pp 130–139

[Jon86] Jones CB (1986) Systematic software development using VDM. International Series in Computer Science. Prentice-Hall, Englewood Cliffs, NJ
[Jor02] Jorgensen PC (2002) Software testing: a craftsman's approach, 2nd edn. CRC Press, Boca Raton, FL
[LPU02] Legeard B, Peureux F, Utting M (2002) Automated boundary testing from Z and B. In: Eriksson L-H, Lindsay PA (eds) Proceedings of FME 2002: Formal Methods—Getting IT Right, International Symposium of Formal Methods Europe, July 2002, Copenhagen, Denmark. Lecture Notes in Computer Science, vol 2391. Springer, Berlin, pp 21–40
[Mor94] Morgan C (1994) Programming from specifications, 2nd edn. International Series In Computer Science. Prentice-Hall, Englewood Cliffs, NJ
[PS96] Peleska J, Siegel M (1996) From testing theory to test driver implementation. In: Gaudel M-C, Woodcock J (eds) FME'96: Industrial Benefit and Advances in Formal Methods, March 1996, Springer-Verlag, pp 538–556
[RAI95] RAISE Development Group (1995) The RAISE development method. Prentice-Hall, UK
[SCSP03] Srivatanakul T, Clark J, Stepney S, Polack F (2003) 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. IEEE, New York, pp 340–351
[SMFS99] Simone do Rocio Senger de Souza, Jose Carlos Maldonado, Sandra Camargo Pinto Ferraz Fabbri, Wanderley Lopes de Souza (1999) Mutation testing applied to Estelle specifications. Softw Q J 8:285–301
[Soc90] IEEE Computer Society (1990) Standard glossary of software engineering terminology, Standard 610.12. IEEE Press, New York
[Sri01] Srivatanakul T (2001) Mutation testing for concurrency. Master's thesis, Department of Computer Science, University of York, UK, September 2001
[Ste95] Stepney S (1995) Testing as abstraction. In: Bowen JP, Hinchey MG (eds) ZUM '95: 9th International Conference of Z Users, Limerick 1995. Lecture Notes in Computer Science, vol 967. Springer, Berlin, pp 137–151
[Sto93] Stocks PA (1993) Applying formal methods to software testing. Ph.D. thesis, Department of computer science, University of Queensland
[Tai96] Tai K-C (1996) Theory of fault-based predicate testing for computer programs. IEEE Trans Softw Eng 22(8):552–562
[Tre92] Tretmans J (1992) A formal approach to conformance testing. Ph.D. thesis, Universiteit Twente
[Tre99] Tretmans J (1999) Testing concurrent systems: A formal approach. In: Baeten JCM, Mauw S (eds) CONCUR'99. Lecture Notes in Computer Science, vol 1664. Springer, Berlin, pp 46–65
[TS87] Tai K-C, Su H-K (1987) Test generation for Boolean expressions. In: Proceedings of the Eleventh Annual International Computer Software and Applications Conference (COMPSAC), pp 278–284
[WJ02] Wimmel G, Jürjens J (2002) Specification-based test generation for security-critical systems using mutations. In: George C, Huaikou M (eds) Proceedings of ICFEM'02, the International Conference of Formal Engineering Methods, October 21–25, 2002, Shanghai, China, Lecture Notes in Computer Science. Springer, Berlin, pp 471–482
[WK03] Warmer J, Kleppe A (2003) The object constraint language: getting your models ready for MDA, 2nd edn. Addison-Wesley, Reading, MA
[Won01] Eric Wong W (ed) (2001) Mutation Testing for the New Century. Kluwer, Dordrecht

Received 5 February 2007
Accepted in revised form 11 April 2008 by E. A. Boiten, M. J. Butler, J. Derrick, L. Groves and J. C. P. Woodcock
Published online 5 June 2008