Testing for Design Faults

Bernhard K. Aichernig\(^1\) and Jifeng He\(^2\)

United Nations University, International Institute for Software Technology (UNU-IIST), Macau

Abstract. Existing theories of testing focus on verification. Their strategy is to cover a specification or a program text to a certain degree in order to raise the confidence in the correctness of a system under test. We take a different approach in the sense that we present a theory of fault-based testing. Fault-based testing uses test data designed to demonstrate the absence of a set of pre-specified faults. Here, the focus is on falsification. The presented theory of testing is integrated into Hoare & He’s theory of programming. As a result, two new test case generation techniques for detecting anticipated faults are presented: one is based on the semantic level of design specifications, the other on the algebraic properties of a programming language.

Keywords: 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 [22].

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

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

---

\(^1\) On leave from Graz University of Technology, Austria.
\(^2\) On leave from East China Normal University, China.

Correspondence and offprint requests to: Bernhard K. Aichernig, UNU-IIST, P.O. Box 3058, Macau (via Hong Kong), e-mail: bka@iist.unu.edu
focusing on faults we will be 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 fault? 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.

The existing testing theories focus on verification. This restricts their use, either to the study of semantical issues (like the question of observable equivalence of processes [10]), or to the testing of very abstract (finite) models for which exhaustive testing is feasible (like in protocol testing [33]).

Fault-based testing was born in practice when testers started to assess the adequacy of their test cases by first injecting faults into their programs, and then by observing if the test cases could detect these faults. This technique of mutating the source code became well-known as mutation testing and goes back to the late 70-ies [19, 11]; since then it has found many applications and has become the major assessment technique in empirical studies on new test case selection techniques [37]. In the early 90-ies formal methods entered the testing stage [8, 12] and it took not long until mutation testing was applied to formal specifications [32]. Here, the idea is to model design errors or misinterpretations of requirements in a very early stage, and then design test cases to prevent such errors. Later, it was the first author's realization of the role of refinement in the process of test case generation, first published in [3], that motivated the present work. Then, a recent paper of the first author introduced mutation testing into the refinement calculus of Back and von Wright [4]. This work made a first step towards a theory of fault-based testing in defining the criteria for when a test case is able to detect a fault. In fact, refinement also helped to define what we mean by fault, since not all changes in a program represent faults that can be observed via testing. In mutation testing this is commonly known as equivalent mutant problem. Formally, we have an equivalent mutant if it refines its original. This motivates our definition of faults via non-refinement. The present paper translates the previous results into the Unifying Theory of Programming of Hoare and He, and contributes two new algorithms for generating test cases.

The paper is structured as follows. After this general introduction, Section 2 introduces the technical framework of [22] used throughout the article. In particular, an overview of the theory of designs is presented. Then, 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 Section 5 a purely algebraic (syntax-oriented) test case generation algorithm is presented. It is based on the algebraic properties of a small, but not trivial, programming language. Finally, in Section 6 we discuss the results, give a more technical review of the 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 [22]. In the following we present a brief introduction of UTP and give motivations for its relevancy to testing.

2.1. Unification of Theories

In every scientific discipline phases of specialization and unification can be observed. During specialization scientists concentrate on some narrowly defined phenomenon and aim to discover the laws which govern it. Over time, it is realized 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 [22] 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 90-ies a unification process started in the area of verification [16]. 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 specialized 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 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 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, conceptional 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 requirement specifications describe the behavior of a device in all possible circumstances and are a starting point for developing a product. But also the test cases are derived from these requirements descriptions. It will be seen that test cases are actually a special form of requirement 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 alphabet.

During experiments it is usual to wait for some initial transient behaviour to stabilize 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. 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. 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 [22].

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

\[
p \vdash Q =_{d} (\text{ok} \land p) \Rightarrow (\text{ok}' \land Q)
\]

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

As can be seen, a design predicate represents a pre-postcondition specification, a concept well-known from VDM [24], RAISE [18], B [1] and more recently OCL [35].

**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.

\[
(x \geq 0 \land e > 0) \vdash (-e \leq x - x'^2 \leq e)
\]

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

**Definition 2.2 (Abort).**

\[
\bot =_{d} \text{true} = \text{false} \vdash \text{false} = \text{false} \vdash \text{false} \vdash Q
\]

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 logical implication. Thus every program \( P \) that terminates is better than a non-terminating one

\[
\forall v, w, \cdots \in A \bullet P \Rightarrow \bot,
\]

for all \( P \) with alphabet \( A \).

Alternatively, following Dijkstra and Scholten [13] using square brackets to denote universal quantification over all variables in the alphabet, we write

\[
[P \Rightarrow \bot],
\]

for all \( P \) with alphabet \( A \).

Formally, we have refinement and say that every program \( P \) refines \( \bot \). In the interpretation of programs and specifications as single predicates, correctness is identified with implication. In other words, every observation that satisfies a program \( P \) must also satisfy its specification \( S \).

\[
[P \Rightarrow S]
\]

The following theorem shows that the implication order applies to designs, too:

**Theorem 2.1 (Refinement of Designs).**

\[
[(p_1 \vdash Q_1) \Rightarrow (p_2 \vdash Q_2)] \iff [p_2 \Rightarrow p_1] \text{ and } [(p_2 \land Q_1) \Rightarrow Q_2] \]

This represents the standard notion of specification refinement, like e.g. in VDM [24] and more recently in OCL [27, 35], where under refinement preconditions are weakened and postconditions are strengthened. Obviously, equivalence is defined as mutual implication. Hence, in UTP the standard refinement operator of the refinement calculus is defined via implication over the designs. This is in contrast to [5], where the weakest-preconditions of contracts are compared.

**Definition 2.3 (Refinement Operator).**

\[
S \sqsubseteq P =_{d} [P \Rightarrow S],
\]

for all \( P, S \) with alphabet \( A \).

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

**Definition 2.4 (Magic).**

\[
\top =_{d} (\text{true} \vdash \text{false})
\]
The program $\top$ is called magic since it miraculously 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 $\top = \neg \text{ok}$.

From the definitions above it follows that the meet and join operations in the lattice of designs are defined as

**Definition 2.5 (Choice).**

\[ P \sqcap Q =_{df} P \lor Q \]

\[ P \sqcup Q =_{df} P \land Q \]

representing internal (non-deterministic, demonic) and external (angelic) choice between two programs $P$ and $Q$.

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.

**Definition 2.6 (Assignment).**

Given a program variable $x$ and an expression $e$

\[ x := e =_{df} (\text{wf}(e) \vdash x' = e \land y' = y \land \ldots \land z' = z) \]

with $\text{wf}$ being the predicate defining the well-formedness ($e$ can be evaluated) of expression $e$.

**Definition 2.7 (Conditional).**

\[ P \ll b \gg Q =_{df} (\text{wf}(b) \vdash (b \land P \lor \neg b \land Q)) \]

with $\text{wf}$ being the predicate defining the well-formedness of the Boolean expression $b$.

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

**Definition 2.8 (Sequential Composition).**

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

If we expand these basic operators to designs, we get the following laws.

**Theorem 2.2.**

\[
\begin{align*}
(p_1 \models Q_1) \sqcap (p_2 \models Q_2) &= (p_1 \land p_2 \models Q_1 \lor Q_2) \\
(p_1 \models Q_1) \sqcup (p_2 \models Q_2) &= (p_1 \lor p_2 \models ((p_1 \Rightarrow Q_1) \land (p_2 \Rightarrow Q_2))) \\
(p_1 \models Q_1) \ll b \gg (p_2 \models Q_2) &= (p_1 \ll b \gg p_2 \models Q_1 \ll b \gg Q_2) \\
(p_1; Q_1): (p_2; Q_2) &= (p_1 \land \neg (Q_1; \neg p_2) \models Q_1; Q_2)
\end{align*}
\]

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. Modeling Faults in Designs

#### 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:
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 behavior 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 behavior 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 semantics in UTP. The fundamental insight behind this approach is that all these artifacts represent descriptions of a system to be build (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 view 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. Furthermore, equality over value lists should be defined.

\[
T(i, o) =_{df} v = i \vdash v' = o
\]

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).

\[
T_v(i, c) =_{df} v = i \vdash c(v')
\]

where \( c \) is a condition on the after state space defining the set of possible expected outcomes.

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

Theorem 3.1.

\[
[T_v(i, c) \Rightarrow T_v(i, d)] \iff [c \Rightarrow d]
\]

This shows that non-deterministic test cases form a partial order. If we fix the input \( i \) the test case \( T_v(i, \text{true}) \) is the smallest test case and \( T_v(i, \text{false}) \) the largest. However, the question arises how to interpret these limits and if they are useful as test cases. \( T_v(i, \text{true}) \) 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. \( T_v(i, \text{false}) \) is equivalent to \( \neg(\text{ok} \wedge v = i) \) and means that such programs cannot be started with input \( i \); such tests are infeasible.

Definition 3.4 (Explorative Test Case).

\[
T_v(i) =_{df} T_v(i, \text{true})
\]
Definition 3.5 (Infeasible Test Case).

\[
T^\perp(i) \equiv q \ T_i(i, \false)
\]

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 \(c\)

\[
\bot \subseteq T_i(i) \subseteq T_i(i, c) \subseteq T(i, o) \subseteq T_o(i) \subseteq \top,
\]

provided \(c(o)\) holds. □

Next the collection of test cases deserves discussion. Such test suites are defined as a set of test cases a tester can choose from. From a program point of view this choice is made externally.

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

\[
TS(s) = q t_1 \uplus \ldots \uplus t_n
\]

The intuition behind this definition is that a test suite is a special form of design (specification) that consists of a number of test cases a tester can choose from. The angelic choice establishes the fact that the choice is made by the environment (here the tester) and that an implementation has to respect all of the test cases in a test suite. 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

\[
T_i \subseteq T_1 \cup T_2, \quad i \in \{1,2\}
\]

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 exhaustive test suite is defined as

\[
TS_{\text{exhaustive}} = q TS(s), \quad \text{such that } TS(s) = D
\]

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 its exhaustive test suite \(TS_{\text{exhaustive}}\)

\[
TS_{\text{exhaustive}} \uplus T_i(i) = TS_{\text{exhaustive}}, \quad \text{provided } p(i) \text{ holds.}
\]

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

\[
TS_{\text{exhaustive}} \uplus T_i(i) = \{\text{by definition of exhaustive test suites}\}
\]

\[
D \uplus T_i(i) = D
\]

\[
= \{\text{by lattice theory}\}
\]

\[
T_i(i) \subseteq D
\]

\[
= \{\text{by definition of refinement and Theorem 2.1}\}
\]

\[
[v = i \Rightarrow p] \land [(v = i \land Q) \Rightarrow \text{true}]
\]

\[
= \{\text{since } p(i) \text{ holds}\}
\]

\[
\text{true}
\]

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 its exhaustive test suite \(TS_{\text{exhaustive}}\). Furthermore, we have a test case \(t \subseteq D\) expressing the fact that \(t\) has been derived from \(D\). Then,

\[
TS_{\text{exhaustive}} \uplus t = TS_{\text{exhaustive}}
\]

Proof. The proof fully exploits the lattice properties of designs.
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 [3] 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 $T \subseteq S \subseteq I$

we define

• $T$ as the 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 $S$.
• when the role of a design is not of concern (specification or implementation), we say that a design satisfies the test cases.

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

Theorem 3.6.

$T(i, o) \subseteq D$ iff $v := o \subseteq (v := i; D)$
$T,(i, c) \subseteq D$ iff $c(v') \subseteq (v := i; D)$

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

$D \nsubseteq D^m$
(or $\neg(D \subseteq D^m)$). We also call $D^m$ a faulty design or a faulty mutation of $D$. 

$TS_{exhaustive} \sqsubset t = \{\text{by definition of exhaustive test suites}\}$
$D \sqsubset t$
$\{\text{by lattice theory, since } t \sqsubseteq D\}$
$D$
$\sqsubseteq TS_{exhaustive}$
Consequently, not all errors lead to faults. Here, for being a fault, a possible observation of this fault must exist. Note also, the use of the term intended (unintended) in the definition, instead of correct (incorrect). This is necessary, since, the latter is only defined with respect to a given specification, but faults can already be present in such specifications from the very beginning.

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 [19] and DeMillo et al. [11]. 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 [2, 4] we have extended mutation testing to the notion of contracts in Back and von Wright’s refinement calculus. 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 explicitly states that test cases can only uncover errors made that are real faults in the sense that it is possible to observe them.

**Theorem 4.1.** Given a design $D$, and a faulty design $D^m$, then there exists a test case $t$, with $t \subseteq D$, such that $t \nsubseteq D^m$.

**Proof.** Assume that such a test case does not exists and for all test cases $t \subseteq D$ also $t \subseteq D^m$ holds. From this follows that $D \subseteq D^m$. This is a contradiction to our assumption that $D^m$ is a faulty design. Consequently, the theorem holds.

Finding 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-based test case as follows:

**Definition 4.1 (Fault-based 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-based test case
when

\[(t \subseteq D) \quad \text{and} \quad (t \not\subseteq D^m)\]

We say that a fault-based 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-based test case will 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-based 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 [4] 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-based test case, able to detect a certain fault, presents a necessary and sufficient property for such test cases. It 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-based 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. [6, 25]). 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. [12, 32, 20, 7]). 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 \(t\sim\) and its associated design \(D\), refinement holds: \(t\sim \subseteq D\).

**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 \(T\sim = d_\bot; D\) such that \([d \Rightarrow p]\). □

The definition makes use of the assertion operator \(b_\bot = (v = v') < b > \bot\), 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 its test equivalence class:

**Theorem 4.2.** Given a design \(D = p \vdash Q\) and one of its equivalence classes. Then, \(T\sim \subseteq D\).

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

\[
T\sim = d_\bot; D \\
= (d \land p) \vdash Q \\
= p \vdash Q \\
= D
\]

Obviously, DNF partitioning can be applied to design predicates. However, in the following we focus on fault-based 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 \( t = T_c(i, c) \) is a representative test case of a test equivalence class \( T_\sim = d_\perp; D \), with \( D = p \vdash Q \), if and only if
\[
d(i) \land p(i) \land [Q(i) \equiv c]
\]
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 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-based Equivalence Class). Given a design \( D = p \vdash Q \) and its faulty design \( D^m = p^m \vdash Q^m \), thus \( D \not\subseteq D^m \). For simplicity, we assume that \( Q \equiv (p \Rightarrow Q) \). Then every representative test case of the test equivalence class
\[
T_\sim = d_\perp; D, \quad \text{with } d = \neg p^m \lor \exists v' \bullet (Q^m \land \neg Q)
\]
is able to detect the fault in \( D^m \).

Proof. From the definition of the test equivalence class, we see that we have two cases for a representative test case \( t \):

Case 1 \( t = T_c(i, c) \) and \((\neg p^m)(i)\):
\[
t \not\subseteq D^m = \neg[(v = i) \Rightarrow p^m] \lor \neg[((v = i) \land Q^m) \Rightarrow c]
\]
\[
= \exists v \bullet ((v = i) \land \neg p^m) \lor \exists v, v' \bullet ((v = i) \land Q^m \land \neg c)
\]
\[
= \{ i \text{ is a witness to 1st. disjunct, since } p^m(i) \text{ holds} \}
\]
\[
\text{true} \lor \exists v, v' \bullet ((v = i) \land Q^m \land \neg c)
\]
\[
= \text{true}
\]

Case 2 \( t = T_c(i, c) \) and \((\exists v' \bullet (Q^m \land \neg Q))(i)\):
\[
t \not\subseteq D^m = \neg[(v = i) \Rightarrow p^m] \lor \neg[((v = i) \land Q^m) \Rightarrow c]
\]
\[
= \exists v \bullet ((v = i) \land \neg p^m) \lor \exists v, v' \bullet ((v = i) \land Q^m \land \neg c)
\]
\[
= \{ \text{by definition of the representative test case} \}
\]
\[
\exists v \bullet ((v = i) \land \neg p^m) \lor \exists v, v' \bullet ((v = i) \land Q^m \land \neg Q(i))
\]
\[
= \{ t \text{ is a witness to 2nd disjunct} \}
\]
\[
\exists v \bullet ((v = i) \land \neg p^m) \lor \text{true}
\]
\[
= \text{true}\]

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

Fig. 1. Original specification of a triangle in UML’s OCL.

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. Two mutated specifications for the triangle example.

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-based 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 [12].

A classical example from the testing literature serves to illustrate the tools functionality.

**Example 4.1.** Consider the well-known Triangle example, specified in Figure 1. The function `Ttype` returns the type of a triangle represented by three given lengths. The precondition restricts the problem specification to cases where the values of `a`, `b`, `c` form a valid triangle. The specification in Figure 1 can be mutated in several ways. Two possibilities are shown in Figure 2. The left hand side models the possible error of changing a variable’s name into any other valid name such that the mutant passes type checking. The right hand side mutant reflects the error, where a designer got the order of the nested `if-statements` wrong. The two test cases generated by the tool are

\[ a = 1, \ b = 2, \ c = 2, \ \text{result} = "isosceles" \]

for the mutant on the left hand side and

\[ a = 1, \ b = 1, \ c = 1, \ \text{result} = "equilateral" \]
for the right hand side mutant. 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 prevent such faults to reside in any implementation of \textit{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:

\begin{align*}
a = 2, & \quad b = 2, \quad c = 1, \quad \text{result} = \text{isosceles} \\
a = 2, & \quad b = 1, \quad c = 2, \quad \text{result} = \text{isosceles} \\
a = 1, & \quad b = 2, \quad c = 2, \quad \text{result} = \text{isosceles} \\
a = 2, & \quad b = 3, \quad c = 4, \quad \text{result} = \text{scalene} \\
a = 1, & \quad b = 1, \quad c = 1, \quad \text{result} = \text{equilateral}
\end{align*}

Analyzing 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 Figure 2. Therefore, one could argue that the fault-based test cases do not add further value. However, in general the fault-based strategy has a higher fault-detecting capability. Consider the two additional mutated specifications shown in Figure 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 for unrevealing the faults in Figure 3:

\begin{align*}
a = 2, & \quad b = 2, \quad c = 2, \quad \text{result} = \text{equilateral} \\
a = 3, & \quad b = 2, \quad c = 4, \quad \text{result} = \text{scalene} \\
a = 1, & \quad b = 3, \quad c = 3, \quad \text{result} = \text{isosceles}
\end{align*}

covers the left hand mutant, and

\begin{align*}
a = 3, & \quad b = 2, \quad c = 4, \quad \text{result} = \text{scalene} \\
a = 1, & \quad b = 3, \quad c = 3, \quad \text{result} = \text{isosceles}
\end{align*}

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

\begin{align*}
a = 1, & \quad b = 3, \quad c = 3, \quad \text{result} = \text{isosceles}
\end{align*}

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-based test cases for them is a valuable vehicle for validation. The details of the tool and a rigorous empirical analysis are subjects for future publications.

5. Testing for Program Faults

So far our discussion on testing has focused on the semantical model of designs. In this section we turn from semantics to syntax. The motivation is to restrict ourselves to a subclass of designs that are expressible,
or at least implementable, in a certain programming language. Thus, we define a program as a predicate expressed in the limited notations (syntax) of a programming language. From the predicate semantics of the programming language operators, algebraic laws can be derived (see [22]). In the following, we will use this algebra of program 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:

\[
\langle \text{program} \rangle ::= \text{true} \\
\langle \text{variable list} \rangle ::= \langle \text{expression list} \rangle \\
\langle \text{program} \rangle \triangleleft \langle \text{Boolean Expression} \rangle \triangleright \langle \text{program} \rangle \\
\langle \text{program} \rangle ; \langle \text{program} \rangle \\
\langle \text{program} \rangle \sqcap \langle \text{program} \rangle \\
\langle \text{recursive identifier} \rangle \\
\mu \langle \text{recursive identifier} \rangle \bullet \langle \text{program} \rangle
\]

The semantics of the operators follows the definitions in Section 2.3. The recursive statement using the least fix-point operator \(\mu\) will be discussed separately in Section 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 give an algebraic semantics to a programming language.

Our idea is to use a normal form to decide if two programs, the original one and the faulty one (also called the mutant) can be distinguished by a test case. When the normal forms of both are equivalent, then the error did not lead to an (observable) fault. This solves the problem of equivalent mutants in mutation testing. Furthermore, the normal form will be used 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 [22], we push the conditions outwards. The proofs of the new laws are given in the appendix. The following assignment normal is taken from [22].

**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 \(g, h\) denote the list of expressions. \(\square\)

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

\[(v ::= g; v ::= h(v)) = (v ::= h(g))\]  \hspace{1cm} \text{(L1)}

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

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

**Theorem 5.1 (Conditional Elimination).**

\[(P \triangleleft c \triangleright Q) = (c \wedge P) \sqcap (\neg c \wedge Q)\]

**Proof.** By definition of conditional and non-deterministic choice. \(\square\)

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.

\[(g_1 \wedge v ::= f) \sqcap (g_2 \wedge v ::= g) \sqcap \ldots \sqcap (g_n \wedge v ::= h)\]
Let $A$ be a set of guarded total assignments, then we write the normal form as $\prod A$.

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

$$v := g = \prod \{(true \land v := g)\}$$

The easiest operators to eliminate is disjunction itself

$$(\prod A) \bot (\prod B) = \prod (A \cup B) \quad \text{(L2)}$$

and the conditional

$$(\prod A) \triangleleft d \triangleright (\prod B) = \prod \{((d \land b) \land P) \mid (b \land P) \in A\} \cap \prod \{((\neg d \land c) \land Q) \mid (c \land Q) \in B\} \quad \text{(L3)}$$

Sequential composition is reduced by

$$(\prod A); (\prod B) = \prod \{(b \land (P; c)) \land P \mid (b \land P) \in A \land (c \land Q) \in B\} \quad \text{(L4)}$$

In order to reduce $P; c$ in law L4 we need an additional law

$$(v := e); b(v) = b(e) \quad \text{(L5)}$$

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 and $P$ a non-deterministic normal form.

Any previous normal form $P$ that terminates can be expressed as

$$false \lor P$$

and the constant $true$ as

$$true \lor v := v$$

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

$$(b \lor P) \cap (c \lor Q) = (b \lor c) \lor (P \cap Q) \quad \text{(L6)}$$

$$(b \lor P) \triangleleft d \triangleright (c \lor Q) = ((b \land d) \lor (c \land \neg d)) \lor (P \triangleleft d \triangleright Q) \quad \text{(L7)}$$

$$(b \lor P); (c \lor Q) = (b \lor (P; c)) \lor (P; Q) \quad \text{(L8)}$$

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.

$$(\prod A); c = \bigvee \{(g \land (P; c)) \mid (g \land P) \in A\} \quad \text{(L9)}$$

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

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

Next, it is shown how this normal form facilitates the generation of fault-based 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. $D \nsubseteq D^m$. 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 good news is that the problem is decidable for non-recursive programs.

The problem of equivalent mutants can be solved 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 easily decided 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.

\[(v := g) = (v := h) \iff [g = h]\]  \hspace{1cm} (L10)

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

\[R \subseteq \{\bigcap A\} \iff \forall P : P \in A \land (R \subseteq P)\]  \hspace{1cm} (L11)

\[((g_1 \land P_1) \cap \ldots \cap (g_n \land P_n)) \subseteq (b \land Q) \iff [\exists i \cdot ((g_i \land P_i) \lneq (b \land Q))]\]  \hspace{1cm} (L12)

\[[(g \land v := f) \lneq (b \land v := h)] \iff [b \Rightarrow (g \land (f = h))]\]  \hspace{1cm} (L13)

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

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

\[\text{Min} =_u z := x < x \leq y \triangleright z := y\]

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.

\[\text{Min}_1 =_u z := x < x \geq y \triangleright z := y\]

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

\[\text{Min} \not\subseteq \text{Min}_1\]

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

\[
\begin{align*}
\text{Min} & = x, y, z := x, y, x < x \leq y \triangleright x, y, z := x, y, y & \{\text{adding identity assignments}\} \\
& = (x \leq y) \land x, y, z := x, y, y & \{\text{by L3}\}
\end{align*}
\]

Next, we reduce Min$_1$ to normal form

\[
\begin{align*}
\text{Min}_1 & = x, y, z := x, y, x < x \geq y \triangleright x, y, z := x, y, y & \{\text{adding identity assignments}\} \\
& = (x \geq y) \land x, y, z := x, y, x \cap \neg(x \geq y) \land x, y, z := x, y, y & \{\text{by L3}\}
\end{align*}
\]

Assume Min $\subseteq$ Min$_1$ then according to L11 we must show that the two refinements hold

\[
\begin{align*}
(x \leq y) \land x, y, z := x, y, x \cap \neg(x \leq y) \land x, y, z := x, y, y & \subseteq (x \geq y) \land x, y, z := x, y, x & \{\text{Case 1}\} \\
(x \leq y) \land x, y, z := x, y, x \cap \neg(x \leq y) \land x, y, z := x, y, y & \subseteq \neg(x \geq y) \land x, y, z := x, y, y & \{\text{Case 2}\}
\end{align*}
\]

We start checking the cases with law L12 and L13.

**Case 1** \[\iff [(x \leq y) \land (x, y, x) := (x, y, x)] \iff (x \geq y) \land (x, y, x) := (x, y, x))\]  \hspace{1cm} (by L12)

\[
\begin{align*}
& \lor \\
& [(\neg(x \leq y) \land (x, y, x) := (x, y, y)) \iff (x \geq y) \land (x, y, x) := (x, y, y))] \\
& = [(x \geq y \Rightarrow (x \leq y \land \text{true})) \\
& \lor \\
& (x \geq y \Rightarrow (x > y \land x = y))] \\
& = \text{false \lor false} \\
& = \text{false}
\end{align*}
\]

It follows that refinement does not hold and that the mutation introduces an observable fault. \[\square\]
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

\[ \text{Min}_2 =_{\theta} z := x < x \mathbin{\square} y \mathbin{\triangleright} z := y \]

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

\[ \text{Min} \sqsubseteq \text{Min}_2 \]

Proof. Since the normal form of Min has already been computed, we start with normalizing \( \text{Min}_2 \).

\[ \text{Min}_2 = x, y, z := x, y, x < x \mathbin{\triangleright} y, z := x, y, y \quad \{ \text{adding identity assignments} \} \\
= ((x < y) \land x, y, z := x, y, x) \sqcap (\neg(x < y) \land x, y, z := x, y, y) \quad \{ \text{by L3} \} \]

Again, two refinements must hold according to L11

\[ ((x \leq y) \land x, y, z := x, y, x) \sqcap (\neg(x \leq y) \land x, y, z := x, y, y) \sqsubseteq (x < y) \land x, y, z := x, y, x \quad \text{(Case 1)} \]

\[ ((x \leq y) \land x, y, z := x, y, x) \sqcap (\neg(x \leq y) \land x, y, z := x, y, y) \sqsubseteq (x < y) \land x, y, z := x, y, y \quad \text{(Case 2)} \]

We check the cases

Case 1 iff \[ [(x \leq y \land (x, y, x) := (x, y, x))] \leq [(x < y) \land (x, y, z := x, y, x))] \]

\[ \lor \]

\[ [(\neg(x \leq y) \land (x, y, x) := (x, y, y))] \leq [(x < y) \land (x, y, z := x, y, x))] \]

\[ = [x < y \Rightarrow x \leq y] \quad \{ \text{by L13} \} \]

\[ \lor \]

\[ x < y \Rightarrow (x > y \land x = y) ] \]

\[ = [\text{true} \vee \text{false}] \]

\[ = \text{true} \]

Case 2 iff \[ [(x \leq y \land (x, y, x) := (x, y, x))] \leq [(\neg(x < y) \land (x, y, z := x, y, y))] \]

\[ \lor \]

\[ [(\neg(x \leq y) \land (x, y, x) := (x, y, y))] \leq [(\neg(x < y) \land (x, y, z := x, y, y))] \]

\[ = [x \geq y \Rightarrow (x \leq y \land x = y)] \quad \{ \text{by L13} \} \]

\[ \lor \]

\[ x \geq y \Rightarrow x > y ] \]

\[ = [x \geq y \Rightarrow (x = y \lor x > y)] \]

\[ = \text{true} \]

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-based test case generation process. \( \square \)

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

\[ (c \lor Q) \sqsubseteq (b \lor P) \quad \text{iff} \quad [b \Rightarrow c] \quad \text{and} \quad (c \lor Q) \sqsubseteq P \quad \text{(L14)} \]

\[ (c \lor (g_1 \land P_1) \sqcap \ldots \sqcap (g_n \land P_n)) \sqsubseteq (b \land Q) \quad \text{iff} \quad [c \lor (\exists i \cdot (g_i \land P_i) \Leftarrow (b \land Q))] \quad \text{(L15)} \]

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 \( x \).

\[ \text{MinNat} =_{\theta} (x < 0 \lor y < 0) \lor (x := x < (x - y) < 0 \lor x := y) \]
First, an equivalent mutant is produced that can be detected by a derivation on the normal form
\[ MinNat_1 =_{eq} (x < 0 \lor y < 0) \lor (x := x < (x - y) < 1) \lor x := y) \]

**Proof.** First, both normal forms are derived.

\[ MinNat = (x < 0 \lor y < 0) \lor ((x, y := x, y) < (x - y) < 0) \lor (x, y := y, y) \]
\[ MinNat_1 = (x < 0 \lor y < 0) \lor ((x - y) < 0 \land x := x, y := x) \lor (\neg((x - y) < 0) \land x := y, y) \]

Since, both have the same non-termination condition, we have to check according to law L14 that
\[ MinNat \sqsubseteq ((x - y) < 1 \land x, y := x, y) \land (\neg((x - y) < 1) \land x, y := y, y) \]

According to law L11 we have to show two refinements
\[ MinNat \sqsubseteq ((x - y) < 1 \land x, y := x, y) \]
\[ MinNat \sqsubseteq (\neg((x - y) < 1) \land x, y := y, y) \]

We verify the cases

Case 1 iff \[ [(x < 0 \lor y < 0) \lor (\neg((x - y) < 0) \land x, y := y, y) \]

\[ (x < 0 \lor y < 0) \lor (\neg((x - y) < 0) \land x, y := y, y) \]

\[ = [(x < 0 \lor y < 0) \lor (\neg((x - y) < 0) \land x, y := y, y) \]

\[ = [(x < 0 \lor y < 0) \lor (x < 0 \lor y < 0) \lor (\neg((x - y) < 0) \land x, y := y, y) \]

\[ = (x < 0 \lor y < 0) \lor (x < 0 \lor y < 0) \lor (\neg((x - y) < 0) \land x, y := y, y) \]

\[ = (x < 0 \lor y < 0) \lor (\neg((x - y) < 0) \land x, y := y, y) \]

\[ = (x < 0 \lor y < 0) \lor (\neg((x - y) < 0) \land x, y := y, y) \]

\[ = (x < 0 \lor y < 0) \lor (\neg((x - y) < 0) \land x, y := y, y) \]

\[ = \text{true} \]

The fact that Case 2 holds can be shown by a similar derivation. \[ \square \]

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 \( P \) be a program and \( P^m \) a faulty mutation of this program with normal forms as follows

\[ P = c \lor \bigcap_j \{(a_j \land v := f_j) \mid 1 \leq j \leq m\} \]
\[ P^m = d \lor \bigcap_k \{(b_k \land v := h_k) \mid 1 \leq k \leq n\} \]
Then, every representative test case of the test equivalence class
\[ T_\sim =_d d_\perp' P \],
with \( d' = (\neg c \land d) \lor \bigvee_k (\neg c \land b_k \land \bigwedge_j (\neg a_j \lor (f_j \neq h_k))) \)
is able to detect the fault in \( D_m \).

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

1. Those test cases have to be added where the mutant does not terminate, but the original program does. That is when \((\neg c \land d)\) holds.
2. In the terminating case, by the two laws \( L11 \) and \( L12 \), 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 \( L13 \) 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 where the output is defined, we add the constraint \( \neg c \). We see that this condition is at the heart of our test domain. Since we have to show non-refinement, this must hold for all the non-deterministic choices of \( P (\bigwedge_j) \). Finally, each non-deterministic choice of \( P^m \) may contribute to non-refinement \( (\bigvee_k) \).

The formal proof uses Theorem 4.3 to derive the test domain \( d'' = d'_1 \lor d'_2 \).

\[ d'_1 = \begin{cases} \text{(by Theorem 4.3)} \\ \neg P^m \end{cases} \]
\[ = \begin{cases} \text{(by definition of } P^m \text{)} \\ d \end{cases} \]
\[ d'_2 = \begin{cases} \text{(by Theorem 4.3)} \\ \exists v' \bullet (Q^m \land \neg Q) \end{cases} \]
\[ = \begin{cases} \text{(by definitions of } P \text{ and } P^m \text{)} \\ \exists v' \bullet \bigcap_k \{ (b_k \land v := h_k) \mid 1 \leq k \leq n \} \land \neg \bigcap_j \{ (a_j \land v := f_j) \mid 1 \leq j \leq m \} \end{cases} \]
\[ = \begin{cases} \text{(by definition of } \bigcap \text{)} \\ \exists v' \bullet \bigvee_k (b_k \land v := h_k) \land \bigwedge_j \neg (a_j \land v := f_j) \end{cases} \]
\[ = \begin{cases} \text{(by distributive law)} \\ \exists v' \bullet \bigvee_k (b_k \land v := h_k) \land \bigwedge_j \neg (a_j \land v := f_j) \end{cases} \]
\[ = \begin{cases} \text{(by definition of assignment and distribution)} \\ \exists v' \bullet \bigvee_k (b_k \land \bigwedge_j (v := h_k \land (\neg a_j \lor \neg (v := f_j)))) \end{cases} \]
\[ = \begin{cases} \text{(by simplification)} \\ \exists v' \bullet \bigvee_k (b_k \land \bigwedge_j ((v' = h_k \land \neg a_j) \lor (v' = h_k \land v' \neq f_j))) \end{cases} \]
\[ = \text{(by definition of } h_k \text{ in the terminating case)} \]
\[ \bigvee_k (b_k \land \bigwedge_j (\neg a_j \lor (h_k \neq f_j))) \]

Thus,
\[ T_\sim = d'_2; P = d'_2; (c \lor Q) = (d'' \land \neg c) \perp P = ((\neg c \land d'_1) \lor (\neg c \land d'_2)) \perp P = d'; P \]

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 \textbf{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

\[ d = (\neg \text{false} \land \text{false}) \lor \bigvee_{k \in \{1,2\}} (\neg \text{false} \land b_k \land \bigwedge_{j \in \{1,2\}} (\neg a_j \lor (f_j \neq h_k))) \]

\[ = (x \geq y \land (x > y \lor \text{false}) \land (x \leq y \lor x \neq y)) \lor (x < y \land (x > y \lor x \neq y) \land (x \leq y \lor \text{false})) \]

\[ = (x \geq y \land x > y \land x \neq y) \lor (x < y \land x \neq y \land (x \leq y) \]

\[ = x > y \lor x < y \]

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

\[ \square \]

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 \( \bigcup S^0, S^1, \ldots \) where each approximation is a refinement of its predecessor, thus \( S^0 \subseteq S^{i+1} \).

Theorem 5.3 (Kleene). If \( F \) is continuous then

\[ \mu X \bullet F(X) = \bigcup_n F^n(\text{true}) \]

where \( F^0(X) =_d \text{true} \), and \( F^{n+1}(X) =_d F(F^n(X)) \)

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

\[ \bigcup S \quad \text{with} \quad S = \{(c_n \lor Q_n) \mid n \in \mathbb{N}\} \]

\( S \) 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 [22] tells us that we can decompose the problem.

\[ (\bigcup S) \subseteq (\bigcup T) \quad \text{iff} \quad \forall i : i \in \mathbb{N} \bullet S_i \subseteq (\bigcup T) \quad (L16) \]

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. An example shall illustrate this.

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:

\[
\begin{align*}
MIN & =_\omega k := 2; \ t := 1; \ \mu X \bullet ((B; \ X) \sqsubseteq k \leq n \triangleright k, t := k, t) \\
B & =_\omega (t := k; \ k := k + 1) \sqsubseteq A[k] < A[t] \triangleright k := k + 1
\end{align*}
\]

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.

\[
F(X) = ((k \leq n \land A[k] < A[t]) \land (k, t := k + 1, k; \ X))
\]

\[
\bigcap ((k \leq n \land A[k] \geq A[t]) \land (k, t := k + 1, t; \ X))
\]

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

\[
S^1 =_\omega F(\text{true}) = (k \leq n) \lor ((k > n) \land k, t := k, t)
\]

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.

\[
S^2 =_\omega F(S^1) = (k + 1 \leq n \land A[k] < A[t]) \lor ((k \leq n \land k + 1 > n \land A[k] < A[t]) \land (k, t := k + 1, k))
\]

\[
\bigcap (k + 1 \leq n \land A[k] \geq A[t]) \lor ((k \leq n \land k + 1 > n \land A[k] \geq A[t]) \land (k, t := k + 1, t))
\]

\[
\bigcap (false) \lor ((k > n) \land k, t := k, t)
\]

\[
= (k < n) \lor ((k = n \land A[k] < A[t]) \land (k, t := k + 1, k))
\]

\[
\bigcap ((k = n \land A[k] \geq A[t]) \land (k, t := k + 1, t))
\]

\[
\bigcap ((k > n) \land k, t := k, t))
\]

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

\[
S^3 =_\omega F(S^2) = (k + 1 < n) \lor ((k + 1 = n \land A[k] < A[t] \land A[k + 1] < A[t]) \land (k, t := k + 2, k + 1))
\]

\[
\bigcap ((k + 1 = n \land A[k] < A[t] \land A[k + 1] \geq A[t]) \land (k, t := k + 2, k))
\]

\[
\bigcap ((k + 1 = n \land A[k] \geq A[t] \land A[k + 1] < A[t]) \land (k, t := k + 2, k + 1))
\]

\[
\bigcap ((k + 1 = n \land A[k] \geq A[t] \land A[k + 1] \geq A[t]) \land (k, t := k + 2, t))
\]

\[
\bigcap ((k = n \land A[k] < A[t]) \land (k, t := k + 1, k))
\]

\[
\bigcap ((k = n \land A[k] \geq A[t]) \land (k, t := k + 1, t))
\]

\[
\bigcap ((k > n) \land k, t := k, t))
\]

It can be seen from the first three approximations that our normal form approximations represent compu-
tation 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:

\[
\begin{align*}
\text{MIN}_1 &= \mu X \cdot ((B; X) \triangleleft k \triangleleft n \triangleright k; t := k, t) \\
S_1^1 &= \mu F(\text{true}) = (k < n) \lor ((k \geq n) \land k; t := k, t)
\end{align*}
\]

Its first approximation gives

\[
S_1^1 = F(\text{true}) = (k < n) \lor ((k \geq n) \land k; t := k, t)
\]

By applying Theorem 5.2 to find test cases that can distinguish the two first approximations, we realize that such a test case does not exist, because \( S_1^1 \subseteq S_1^1 \). The calculation of the test equivalence class domain predicate \( d_1 \) gives \( \text{false} \):

\[
d_1 = \{ \text{by Theorem 5.2} \} \\
\quad = \text{false} \lor \text{false} = \text{false}
\]

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

\[
S_2^1 = \mu F(S_1) = (k + 1 < n) \lor (((k + 1 = n \land A[k] < A[t]) \land (k, t := k + 1, k)) \\
\quad \lor ((k + 1 = n \land A[k] > A[t]) \land (k, t := k + 1, t)) \\
\quad \lor ((k \geq n) \land k, t := k, t))
\]

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

\[
d(k, t) = \{ \text{by Theorem 5.2} \} \\
\quad = (k < n) \land k < n \\
\quad \lor (k \geq n \land k + 1 = n \land A[k] < A[t] \land \ldots \\
\quad \lor (k \geq n \land k + 1 = n \land A[k] \geq A[t] \land \ldots \\
\quad \lor (k \geq n \land k \geq n \\
\quad \land (\neg(k = n \land A[k] < A[t]) \lor \text{true}) \\
\quad \land (\neg(k = n \land A[k] \geq A[t]) \lor \text{true}) \\
\quad \land (\neg(k > n) \lor \text{false}))
\]

\[
\quad = \text{false} \\
\quad \lor (k \geq n \land k \leq n) \\
\quad = (k = n)
\]

By substituting the initialization values \((k = 2\text{ and } t = 1)\) the concrete fault-based test equivalence class is:

\[
T_{\sim 1} = (n = 2) \perp; \text{MIN}
\]

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

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

Note also that the test equivalence class $T_{\sim 1}$ 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 $k$.

$$MIN_2 := k := 2; t := 1; \mu X \bullet ((B_2; X) < k \leq n \triangleright k, t := k, t)$$

$$B_2 := t := k; k := k + 1 \triangleright A[k] < A[t] \triangleright k := k$$

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

$$S^2_2 = (k + 1 \leq n \wedge A[k] < A[t]) \lor ((k \leq n \wedge k + 1 > n \wedge A[k] < A[t]) \wedge (k, t := k + 1, k))$$

$$\cap$$

$$((k \leq n \wedge k > n \wedge A[k] > A[t]) \wedge (k, t := k, t))$$

$$\cap$$

$$(false) \lor ((k > n) \wedge k, t := k, t)$$

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

$$S^2_2 = (k < n \lor (k = n \wedge A[k] \geq A[t])) \lor$$

$$(((k = n \wedge A[k] < A[t]) \wedge (k, t := k + 1, k))$$

$$\cap$$

$$((k > n) \wedge k, t := k, t))$$

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.

$$T_{\sim 2}(k, t) = (k = n \wedge A[k] \geq A[t]) \perp MIN$$

$$T_{\sim 2} = (n = 2 \wedge A[2] \geq A[1]) \perp MIN$$

We convince ourselves that our method works, since the calculated test cases are indeed those, where $MIN_2$ fails to terminate, due to the missing incrementation of $k$. □

The example demonstrated that it is possible 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. Such a decision represents a test hypothesis (i.e. a regularity hypothesis according to [17]), where the tester assumes, that if $n$ iterations did not reveal a fault, an equivalent mutant has been produced.

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 [22]. It extents the application domain of Hoare & He’s theory of programming to the discipline of testing. It has been demonstrated that the new theory enables the formal reasoning about test cases, more precisely about the fault detecting power of test cases. As a consequence, new test case generation methods could be developed.

The first test case generation method (Definition 4.1) is a general criterion for fault-based test cases. It is not completely new, but has been translated from our previous work [4] to the theory of designs. It states that a test case in order to find a fault in a design (which can range from specifications to programs) must be an abstraction of the original design; and in addition it must not be an abstraction of the faulty design.
No such test cases exist if the faulty design is a refinement of the original one. Note that the translation of this criterion from a different mathematical framework was straightforward. Since our previous definition was solely based on the algebraic properties of refinement, we just had to change the definition of refinement (from weakest precondition inclusion to implication). This demonstrates the generality of our refinement-based testing theory.

The second test case generation method (Theorem 4.3) is more constructive and specialized 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 [23].

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 nondeterminism and general recursion. A tool that uses this technique will combine constraint solving and symbolic manipulation.

Related Work. Until recently, most of the research on specification-based testing 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 [12], Z, B [26], or Lotos [17]. Only few noticed the relevance of a fault-based strategy on a specification level.

Perhaps the first one was Stocks, who applied mutation testing to Z specifications [32]. 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 [9]. He uses a combination of a theorem prover and a collection of constraint solvers. The theorem prover generates the DNF, simplifies the formulas (and helps formulating different testing strategies). This is in contrast to our implementation of the OCL test case generator, where Constraint Handling Rules [15] 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 [36] 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.

A group in York has recently started to use fault-based techniques for validating their CSP models [29, 30]. Their aim is not to generate test cases, but to study the equivalent mutants. Their work demonstrates that semantical 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 [14].

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 [10] developed a testing theory that defines the equivalence and refinement of processes in terms of testers. Similarly, the failure-divergency refinement of CSP [21] 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 labeled transition systems [33, 34]. They
are the foundations of Peleska's work on testing, as well [28]. However, these theories do not focus on the use of abstraction (the reverse of refinement) in order to select a subset of test cases.

It was Stepney in her work on Object-Z, who first promoted explicitly the use of abstraction for designing test cases [31]. The application of a refinement calculus to define different test-selection strategies is a contribution of the first author's doctoral thesis [2]. 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 help is the combination of testing and formal proofs. This question is related to the highly controversial question in the 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 later 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 Section 4.3. 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.

Acknowledgements. The test case generator discussed in Section 4.3 was implemented by Percy Antonio Pari Salas during his fellowship at UNU-IIST.

References

Appendix

In this appendix the proofs of the algebraic laws that have been newly introduced in Section 5 are presented. Those not listed below are taken from [22].
Proof of L3

\[(\bigcap A) \triangleright d \triangleright (\bigcap B) = \{ \text{by L1, Sect. 5.2 in [22]} \}\]
\[\bigcap \{(b \land P) \triangleright d \triangleright (c \land Q) \mid (b \land P) \in A \land (c \land Q) \in B\}\]
\[= \{ \text{by Theorem 5.1} \}\]
\[\bigcap \{(d \land b \land P) \triangleright (\neg d \land c \land Q) \mid (b \land P) \in A \land (c \land Q) \in B\}\]
\[= \{ \text{by L2} \}\]
\[\bigcap \{{(d \land b \land P) \mid (b \land P) \in A} \cup \{(\neg d \land c \land Q) \mid (c \land Q) \in B\}\}
\[= \{ \text{by L2} \}\]
\[\bigcap \{(d \land b \land P) \mid (b \land P) \in A\} \cap \{(\neg d \land c \land Q) \mid (c \land Q) \in B\}\]

Proof of L4

\[(\bigcap A) ; (\bigcap B) = \{ \text{by L2, Sect. 5.2 in [22]} \}\]
\[\bigcap \{(b \land P) ; (c \land Q) \mid (b \land P) \in A \land (c \land Q) \in B\}\]
\[= \{ \text{by definition of} ; \}\]
\[\bigcap \{(\exists v_0 \bullet b(v) \land P(v, v_0) \land c(v_0)) \land Q(v_0, v') \mid (b \land P) \in A \land (c \land Q) \in B\}\]
\[= \{ \text{by definition of} ; \}\]
\[\bigcap \{(b(v) \land (\exists v_0 \bullet P(v, v_0)) \land c(v_0))\land (\exists v_0 \bullet P(v, v_0) \land Q(v_0, v')) \mid (b \land P) \in A \land (c \land Q) \in B\}\]
\[= \{ \text{by definition of} ; \}\]
\[\bigcap \{(b \land (P; c)) \land P; Q \mid (b \land P) \in A \land (c \land Q) \in B\}\]

Proof of L7

\[(b \lor P) \triangleright d \triangleright (c \lor Q) = \{ \text{by L7, Sect. 5.3 in [22]} \}\]
\[\triangleright d \triangleright (P \lor d \lor Q)\]
\[= \{ \text{by definition of conditional} \}\]
\[\triangleright d \triangleright Q\]

Proof of L9

\[(\bigcap A) ; c = \{ \text{by L4, Sect. 5.3 in [22]} \}\]
\[\bigvee \{(g \land (P; c)) \mid (g \land P) \in A\}\]
\[= \{g \text{ is a predicate over} v\}\]
\[\bigvee \{(g \land (P; c)) \mid (g \land P) \in A\}\]

Proof of L12

\[(g_1 \land P_1) \cap \ldots \cap (g_n \land P_n) \subseteq (b \land Q) = \{ \text{by definitions of} \cap \text{and} \subseteq \}\]
\[\{ [(g_1 \land P_1) \lor \ldots \lor (g_n \land P_n)] \subseteq (b \land Q) \}\]
\[= \{ \text{by distribution of} \subseteq \}\]
\[\{(g_1 \land P_1) \subseteq (b \land Q) \lor \ldots \lor (g_n \land P_n) \subseteq (b \land Q) \}\]
\[= \{ \text{by definition of existential quantification} \}\]
\[\exists i \bullet ((g_i \land P_i) \subseteq (b \land Q))\]
Proof of L13

\[(g \land v := f) \iff (b \land v := h)] = \{\text{by definition of total assignment}\} \quad (g \land v' = f) \iff (b \land v' = h]\]

\[= (g \iff (b \land v' = h)) \land (v' = f \iff (b \land v' = h))] \quad = (g \iff b) \land (v' = f \iff v' = h) \iff b)]

\[= (g \iff b) \land ((f = h) \iff b)] \quad = ((g \land (f = h)) \iff b] \quad = ((g \land (f = h)) \iff b]

Proof of L15

L15 follows directly from L12.