

Mutation Testing in the Refinement Calculus

Bernhard K. Aichernig

The United Nations University
International Institute for Software Technology (UNU/IIST)
Macau

Abstract. This article discusses mutation testing strategies in the context of refinement. Here, a novel generalization of mutation testing techniques is presented to be applied to contracts ranging from formal specifications to programs. It is demonstrated that refinement and its dual abstraction are the key notions leading to a precise and yet simple theory of mutation testing. The refinement calculus of Back and von Wright is used to express the concepts like contracts, useful mutations, test-cases and test-coverage.

Keywords: formal methods, formal specifications, mutation testing, refinement, abstraction, test-case generation, test-coverage.

1. Introduction

The synergy of combining formal methods and testing has become a popular area of research. In the last years, test-generation tools have been invented for almost every popular specification language. One reason is the industry's demand to cut the costs of software testing. Another is the academics' insight that testing is complementary to proving the correctness of a program (see e.g. Hoare's comments on testing in [Hoa03]).

However, only little research has been put into the question how the related development techniques such as program synthesis contribute to testing. Our current research addresses this open issue. In our previous work we have already demonstrated that test design can be viewed as a reverse program synthesis problem of finding adequate abstractions [Aic01b, Aic01a, Aic01c]. In this paper we focus on mutation testing and its relation to refinement.

1.1. Mutation Testing

Mutation testing is a fault-based testing technique introduced by Hamlet [Ham77] and DeMillo et al. [DLS78]. It is a means of assessing test suites. When a program passes all tests in a suite, mutant programs are

Correspondence and offprint requests to: Bernhard K. Aichernig, UNU/IIST, P.O. Box 3058, Macau (via Hong Kong), e-mail: bka@iist.unu.edu

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 this work the mutation testing strategy is extended to the general notion of contracts which includes executable programs as well as more abstract specifications. The motivation is to start the test-case design early in the specification phase and to check which kinds of wrongly implemented requirements, represented by specification mutations, can be found by the test-cases. It is important to stress the difference to conventional mutation testing of programs. The aim of specification mutation is not to find errors in the specification, but to find errors in the program caused by wrongly understood, or wrongly refined (implemented), specifications. Therefore, the mutated specifications represent a programmer’s possible misinterpretations of requirements, or an implementation error expressible in terms of the specification. The derived test-cases should reveal this kind of errors in the implementation. Thus, the hypothesis of this technique is that the programmer makes small errors in interpreting and implementing the specification.

Here, a test-case is assessed and designed relative to the mutated specification. However, since specifications are in general not executable, the check if a contract passes a test-case cannot be done by running the contract. A more general notion of *checking* must be developed. It will be seen that the notion of refinement is the key to this checking or assessment procedure. First, the role of test-cases in a formal development process has to be defined.

1.2. Test-Design as a Formal Synthesis Problem

Experience shows that test-cases for non-trivial systems are complex algorithms that have to be executed either by a tester manually, or by test drivers. The idea of our framework is the derivation of such test-cases T from a formal specification S . A test-case T should be correct with respect to S , and a program P that implements S should be tested with T . Thus, the derivation of T from S constitutes a formal synthesis problem. In order to formalize a certification criteria for this synthesis process, the relation between T , S and P must be clarified.

It is well-known that the relation between a specification S and its correct implementation P is called refinement. We write

$$S \sqsubseteq P$$

for expressing that P is a correct refinement (implementation) of S . The problem of deriving an unknown P from a given S is generally known as program synthesis:

$$S \sqsubseteq P?$$

In our previous work [Aic01b, Aic01a], we have shown that correct test-cases T can be defined as abstraction(s) of the specification S , or:

$$T \sqsubseteq S \sqsubseteq P$$

Consequently, test-case synthesis is a reverse refinement problem. The reverse refinement from a given S into an unknown test-case T ? is here called abstraction¹, and denoted as:

$$S \sqsupseteq T?$$

As a consequence, formal synthesis techniques can be applied for deriving test-cases from a formal specification. This idea is the base of our generalized view on testing techniques. In this paper it is shown how efficient test-cases T ? using mutation techniques can be designed. We use Back and von Wright’s refinement calculus [BvW98] in order to discuss the topic. The following Section 2 introduces the basic notation and the concepts of refinement and its dual abstraction. Next, in Section 3 it is shown that test-cases are in fact abstractions of formal specifications. Section 4 presents the new contributions to mutation testing. In

¹ Here, a test-case is abstract in the sense that an implementation (or specification) should refine the test-case if it passes the test. To avoid confusion, one could use the term anti-refinement instead of abstraction. However, the term abstraction is established in the field.

Section 5 an example is used to discuss the proposed mutation technique in more detail. An additional result is presented in Section 6: The mutation testing technique can be extended to decide which new test-cases should be added under refinement. Finally, we draw our conclusions in Section 7.

2. On Contracts, Refinement and Abstraction

2.1. Contracts

The prerequisite for testing is some form of contract between the user and the provider of a system that specifies what it is supposed to do. In case of system-level testing usually user and software requirement documents define the contract. Formal methods propose mathematics to define such a contract unambiguously and soundly. In the following the formal contract language of Back and von Wright [BvW98] is used. It is a generalization of the conventional pre- and post-condition style of formal specifications known from VDM, B and Z. The foundation of this refinement calculus is based on lattice-theory and classical higher-order logic (HOL).

A system, described by a contract, is modeled by a global state σ of type Σ , denoted by $\sigma : \Sigma$. The state has a number of program variables x_1, \dots, x_n , each of which can be observed and changed independently of the others. State changes, and thus functionality, is either expressed by functional state transformers f or relational updates R . A state transformer is a function $f : \Sigma \rightarrow \Gamma$ mapping a state space Σ to the same or another state space Γ .

A relational update $R : \Sigma \rightarrow \Gamma \rightarrow \text{Bool}$ specifies a state change by relating the state before with the state after execution. In HOL, relations are modeled by functions mapping the states to Boolean valued predicates. For convenience, a relational assignment ($x := x' | b$) is available and generalizes assignment statements. It sets a state variable x to a new state x' such that b , relating x and x' , holds.

The language further distinguishes between the responsibilities of communicating agents in a contract. Here, the contract models the viewpoint of one agent called the *angel* who interacts with the rest of the system called the *demon*. In our work following [BvW98, BMvW99], the user is considered the angel and the system under test the demon. Relational contract statements denoted by $\{R\}$ express relational updates under control of the angel (user). Relational updates of the demon are denoted by $[R]$ and express updates that are non-deterministic from the angel's point of view. Usually, we take the viewpoint of the angel.

The contract statement $\langle f \rangle$ denotes a functional update of the state, determined by a state transformer f . There is no choice involved here, neither for the angel nor the demon agent, since there is only one possible next state for a given state.

Two contracts can be combined by sequential composition $C_1; C_2$ or choice operators. The angelic choice $C_1 \sqcup C_2$ and the demonic choice $C_1 \sqcap C_2$ define non-deterministic choice of the angel or demon between two contracts C_1 and C_2 . Furthermore, predicate assertions $\{p\}$ and assumptions $[p]$ define conditions the angel, respectively the demon, must satisfy. In this language of contract statements $\{p\}; \langle f \rangle$ denotes partial functions and $\{p\}; [R]$ pre-postcondition specifications. Furthermore, recursive contracts, using the least fix-point operator μ and the greatest fix-point operator ν , are possible for expressing several patterns of iteration.

The core contract language used in this work can be summarized by the following BNF grammar, where p is a predicate and R a relation.

$$C := \{p\} \mid [p] \mid \{R\} \mid [R] \mid C; C \mid C \sqcup C \mid C \sqcap C \mid \mu X \cdot C$$

To simplify matters, we will extend this core language by our own contract statements. However, all new statements will be defined by means of the above core language. Thus, our language extensions are conservative. This means that our new definitions do not introduce any inconsistencies into the existing theory of the refinement calculus.

2.2. Example Contracts

A few simple examples should illustrate the contract language. The following contract is a pre- postcondition specification of a square root algorithm:

$$\{x \geq 0 \wedge e > 0\}; [x := x' \mid -e \leq x - x'^2 \leq e]$$

The precondition is an assertion about an input variable x and a precision e . A relational assignment expresses the demonic update of the variable x to its new value x' . The contract would be breached by the user, if $x \geq 0 \wedge e > 0$ does not hold initially. If this precondition is true, then x is assigned some value x' such that the postcondition $-e \leq x - x'^2 \leq e$ holds.

Consider the following version of the square root contract that uses both kinds of non-determinism:

$$\{x, e := x', e' \mid x' \geq 0 \wedge e' > 0\}; [x := x' \mid -e \leq x - x'^2 \leq e]$$

In this contract the interaction of two agents is specified explicitly. This contract requires that our agent, called the angel, first chooses new values for x and e . Then the other agent, the demon, is given the task of computing the square-root in the variable x .

The following example should demonstrate that programming constructs can be defined by means of the basic contract statements. A conditional statement can be defined by an angelic choice as follows:

$$\text{if } P \text{ then } S_1 \text{ else } S_2 \text{ fi} \triangleq \{P\}; S_1 \sqcup \{\neg P\}; S_2$$

Thus, the angel agent can choose between two alternatives. The agent will, however, always choose only one of these, the one for which the assertion is true, because choosing the alternative where the guard is false would breach the contract. Hence, the agent does not have a real choice if he wants to satisfy the contract.

Alternatively, we could also define the conditional in terms of choices made by the other agent (demon) as follows:

$$\text{if } P \text{ then } S_1 \text{ else } S_2 \text{ fi} \triangleq [P]; S_1 \sqcap [\neg P]; S_2$$

These two definitions are equivalent. The choice of the demon agent is not controllable by our agent, thus to achieve some desired condition, our agent has to be prepared for both alternatives. If the demon agent is to carry out the contract without violating our agent's assumptions, it has to choose the first alternative when P is true and the second alternative when P is false.

In general, iterations can be expressed by recursive contracts. One possibility of specifying a recursive contract is to use the form $(\mu X \cdot C)$. Here X is a variable that ranges over contract statements, while $(\mu X \cdot C)$ is the contract statement C , where each occurrence of X in C is interpreted as a recursive invocation of the contract C . The operator μ defines a least fix-point interpretation of this recursion. For example, the standard while loop is defined as follows:

$$\text{while } g \text{ do } S \text{ od} \triangleq (\mu X \cdot \text{if } g \text{ then } S; X \text{ else skip fi})$$

We write $\text{skip} \triangleq \langle id \rangle$ for the action that applies the identity function to the present state.

2.3. Semantics

The semantics of the contract statements is defined by weakest-precondition predicate transformers. A predicate transformer $C : (\Gamma \rightarrow \text{Bool}) \rightarrow (\Sigma \rightarrow \text{Bool})$ is a function mapping postcondition predicates to precondition predicates. The set of all predicate transformers from Σ to Γ is denoted by $\Sigma \mapsto \Gamma \triangleq (\Gamma \rightarrow \text{Bool}) \rightarrow (\Sigma \rightarrow \text{Bool})$.

The different roles of the angel and the demon are reflected in the weakest-precondition semantics in Figure 1. Here q denotes a postcondition predicate and σ a particular state, p is an arbitrary predicate, and R a relation. Following an established tradition, we identify contract statements with predicate transformers that they determine. The notation $f.x$ is used for function application instead of the more common form $f(x)$. In this semantics, the breaching of a contract by our angel agent, means that the weakest-precondition is **false**. If a demon agent breaches a contract, the weakest-precondition is trivially **true**. The semantics of the specification constructs above can be interpreted as follows:

- The weakest-precondition semantics of an *assertion* contract reflects the fact that, if the final state of the contract should satisfy the post-condition q , then in addition, the assertion predicate p must hold. It can be seen that the global state is not changed by an assertion statement. Consequently, the angel breaches this contract if $p \cap q$ evaluates to **false**.
- The semantics of an *assumption* shows that the demon is responsible for satisfying an assumption predicate p . If the assumption does not hold, the demon breaches the contract and the angel is released from the contract. In this case, the weakest-precondition trivially evaluates to **true**.

$\{p\}.q$	\triangleq	$p \cap q$	(<i>assertion</i>)
$[p].q$	\triangleq	$\neg p \cup q$	(<i>assumption</i>)
$\{R\}.q.\sigma$	\triangleq	$(\exists \gamma \in \Gamma . R.\sigma.\gamma \wedge q.\gamma)$	(<i>angelic update</i>)
$[R].q.\sigma$	\triangleq	$(\forall \gamma \in \Gamma . R.\sigma.\gamma \Rightarrow q.\gamma)$	(<i>demonic update</i>)
$(C_1; C_2).q$	\triangleq	$C_1.(C_2.q)$	(<i>sequential composition</i>)
$(C_1 \sqcup C_2).q$	\triangleq	$C_1.q \cup C_2.q$	(<i>angelic choice</i>)
$(C_1 \sqcap C_2).q$	\triangleq	$C_1.q \cap C_2.q$	(<i>demonic choice</i>)

Fig. 1. Weakest-precondition semantics of the contract statements.

- The *angelic update* definition says that a final state γ must exist in the relation R , such that the postcondition q holds. The existential quantifier in the weakest-precondition shows that the angel has control of this update. The angel can satisfy the contract, as long as one update exists that satisfies the postcondition. In the set notation this update is defined as $\{R\}.q.\sigma \triangleq R.\sigma \cap q \neq \emptyset$.
- This is in contrast to the definition of the *demonic update*. Here, all possible final states γ have to satisfy the postcondition. The reason is that the demonic update is out of our control. It is not known, to which of the possible states, described by the relation R , the state variables will be set. In the set notation this update is defined as $[R].q.\sigma \triangleq R.\sigma \subseteq q$.
- The weakest-precondition of two sequentially combined contracts is defined by the *composition* of the two weakest-preconditions.
- The *angelic choice* definition shows that the weakest-precondition is the union of the weakest-precondition of the two contracts. Thus, a further choice of the angel, further weakens the weakest-preconditions.
- The *demonic choice* is defined as the intersection of the weakest-preconditions of the two contracts. Thus, demonic choice means a strengthening of the weakest-preconditions.

For further details of the predicate transformer semantics, we refer to [BvW98].

2.4. Refinement and Abstraction

The notion of contracts includes specification statements as well as programming statements. More complicated specification statements as well as programming statements can be defined by the basic contract statements presented above. The refinement calculus provides a synthesis method for refining specification statements into programming statements that can be executed by the target system. The refinement rules of the calculus ensure by construction that a program is correct with respect to its specification.

Formally, refinement of a contract C by C' , written $C \sqsubseteq C'$, is defined by the pointwise extension of the subset ordering on predicates: For Γ being the after state space of the contracts, we have

$$C \sqsubseteq C' \triangleq \forall q \in (\Gamma \rightarrow \text{Bool}) . C.q \subseteq C'.q$$

This ordering relation defines a lattice of predicate transformers (contracts) with the lattice operators meet \sqcap and join \sqcup . The top element \top is $\text{magic}.q \triangleq \text{true}$, a statement that is not implementable since it can magically establish every postcondition. The bottom element \perp of the lattice is $\text{abort}.q \triangleq \text{false}$ defining the notion of abortion. The choice operators and negation of contracts are defined by pointwise extension of the corresponding operations on predicates. A large collection of refinement rules can be found in [BvW98, Mor94].

Abstraction is dual to refinement. If $C \sqsubseteq C'$, we can interchangeably say C is an abstraction of C' . In order to emphasize rather the search for abstractions than for refinements, we write $C \sqsupseteq C'$ to express that C' is an abstraction of C . Trivially, abstraction can be defined as:

$$C \sqsupseteq C' \triangleq C' \sqsubseteq C$$

Hence, abstraction is defined as the reverse of refinement.

3. Test-cases are Abstractions!

In the following, we will demonstrate that test-cases, common in software engineering, are in fact contracts — highly abstract contracts. To keep our discussion simple, we do not consider parameterized procedures, but only global state manipulations. In [BvW98] it is shown how procedures can be defined in the contract language. Consequently, our approach scales up to procedure calls.

3.1. Input-Output Tests

The simplest form of test-cases are pairs of input i and output o . We can define such an input-output test-case TC as a contract between the user and the unit under test:

$$\text{TC } i \ o \triangleq \{x = i\}; [y := y' | y' = o]$$

Intuitively, the contract states that if the user provides input i , the state will be updated such that it equals o . Here, x is the input variable and y the output variable.

In fact, such a TC is a formal pre-postcondition specification solely defined for a single input i . This demonstrates that a collection of n input-output test-cases TCs is indeed a pointwise defined formal specification:

$$\text{TCs} \triangleq \text{TC } i_1 \ o_1 \sqcup \dots \sqcup \text{TC } i_n \ o_n$$

Moreover, the following theorem shows that such test-cases are abstractions of general specifications, under the condition that the specification is deterministic for the input-value of the test-case.

Theorem 3.1. Let $p : \Sigma \rightarrow \text{Bool}$ be a predicate, $Q : \Sigma \rightarrow \Gamma \rightarrow \text{Bool}$ be a relation on states, and $TC \ i \ o$ represents a test-case with input i in variable x and expected output o in variable y . Then

$$\{p\}; [Q] \sqsupseteq \text{TC } i \ o \equiv ((x = i) \subseteq p \wedge (|x = i|; Q) \subseteq |y := o|)$$

where

- $|p|$ denotes the coercion of a predicate $p : \Sigma \rightarrow \text{Bool}$ to a relation (here $x = i$). The relation $|p| : \Sigma \rightarrow \Sigma \rightarrow \text{Bool}$ is defined as follows:

$$|p|. \sigma. \gamma \triangleq (\sigma = \gamma) \wedge p. \sigma$$

- $|f|$ denotes the coercion of a state transformer $f : \Sigma \rightarrow \Gamma$ to a relation (here $y := o$). The relation $|f| : \Sigma \rightarrow \Gamma \rightarrow \text{Bool}$ is defined as follows:

$$|f|. \sigma. \gamma \triangleq f. \sigma = \gamma$$

- the composition operator $;$ is overloaded for relations. The relation composition $P; Q$ is defined as follows:

$$(P; Q). \sigma. \delta \triangleq (\exists \gamma. P. \sigma. \gamma \wedge Q. \gamma. \delta)$$

Proof. See [Aic01b].

□

The intuitive justification of this abstraction relation between specifications and test-cases of Theorem 3.1 is that the precondition $\{p\}$ is strengthened to a single input state — and precondition strengthening is abstraction. Furthermore, Theorem 3.1 shows that only for deterministic specifications, simple input-output test-cases are sufficient, in general. The theorem becomes simpler if the whole input and output state is observable, which is shown in the following corollary.

Corollary 3.1. Let $p : \Sigma \rightarrow \text{Bool}$ be a predicate, $Q : \Sigma \rightarrow \Gamma \rightarrow \text{Bool}$ be a relation on states, and $TC \ i \ o$ represents a test-case, where the whole change of state is observable. Thus, the input $i : \Sigma$ and output $o : \Gamma$. Then

$$\{p\}; [Q] \sqsupseteq \text{TC } i \ o \equiv (p. i \wedge (Q. i = o))$$

Proof. The corollary follows from Theorem 1 and the assumption that $i : \Sigma$ and $o : \Gamma$.

□

Furthermore, the selection of certain test-cases out of a collection of test-cases can be considered as abstraction:

Corollary 3.2.

$$\text{TC } i_1 \ o_1 \sqcup \dots \sqcup \text{TC } i_n \ o_n \sqsupseteq \text{TC } i_k \ o_k$$

for all k , $1 \leq k \leq n$.

Proof. The theorem is valid by definition of the join operator $a \sqcup b \sqsupseteq a$ or $a \sqcup b \sqsupseteq b$, respectively.

□

The fact that test-cases are indeed formal specifications and, as Theorem 1 shows, abstractions of more general contracts explains why test-cases are so popular: First, they are abstractions in the sense that many details are left away, and are thus easy to understand. Second, they are formal and thus unambiguous.

3.2. Non-Deterministic Test-Cases

In general, a contract can permit more than one result for a given input data. In this case, testing the requirements with simple input-output values is insufficient. An output predicate $\omega : \Sigma \rightarrow \text{Bool}$ can be used for describing the set of possible outputs. We define such a test-case as follows:

$$\text{TCp } i \ \omega \triangleq \{x = i\}; [y := y' | \omega]$$

For being a correct test-case with respect to a contract this type of test-case should be an abstraction of the contract.

Theorem 3.2. Let $p : \Sigma \rightarrow \text{Bool}$ be a predicate, $Q : \Sigma \rightarrow \Sigma \rightarrow \text{Bool}$ a relation on states, and $\text{TC2 } i \ o$ a test-case with input i in variable x and output in variable y such that the output predicate ω holds. Then we have:

$$\{p\}; [Q] \sqsupseteq \text{TCp } i \ \omega \equiv (x = i) \subseteq p \wedge (|x = i; Q| \subseteq |\omega|)$$

□

The theorem shows that a test-case for non-deterministic results can be calculated by strengthening the precondition to a single input value and weakening the postcondition to the outputs of interest. The fact that the output predicate ω might be weaker than Q represents the case that not all properties of an output might be observed. This can be useful if not all variables, or only selected properties of the output, should be checked.

3.3. Partition Tests

Partition analysis of a system is a powerful testing technique for reducing the possible test-cases: Here, a contract is analyzed and the input domains are split into partitions. A partition is an equivalence class of test-inputs for which the tester assumes that the system will behave in the same way. These assumptions can be based on a case analysis of a contract, or on the experience that certain input values are fault-prone.

In case of formal specifications, the transformation into a disjunctive normal form (DNF) is a popular partition technique (see e.g. [DF93, Sto93, HP94, HNS97]). This technique is based on rewriting according to the rule $A \vee B \equiv (A \wedge B) \vee (\neg A \wedge B) \vee (A \wedge \neg B)$.

A *partitioning* of a contract statement $\{p\}; [R]$ is a collection of n disjoint partitions $\{p_i\}; [R_i]$, such that

$$\{p\}; [R] = \{p_1\}; [R_1] \sqcup \dots \sqcup \{p_n\}; [R_n]$$

and

$$\forall i, j \in \{1, \dots, n\} \cdot i \neq j \Rightarrow p_i \cap p_j = \emptyset$$

These partitions describe classes of test-cases, here called partition test-cases. Often in the literature, if the context is clear, a partition test-case is simply called a test-case.

Partition test-cases are abstractions of specifications, too:

Theorem 3.3. Let $\{p_i\}; [R_i]$ be a partition of a specification $\{p\}; [R]$. Then

$$\{p\}; [R] \sqsupseteq \{p_i\}; [R_i]$$

Proof. The result follows directly from the definition of partitioning above, and from the definition of \sqsupseteq .

□

Up to now, only the commonly used pre-postcondition contracts have been considered. They are a normal form for all contracts not involving angelic actions. This means that arbitrary contracts excluding \sqsupseteq and $\{R\}$ can be formulated in a pre-postcondition style (see Theorem 26.4 in [BvW98]).

However, our result that test-cases are abstractions holds for general contract statements involving user interaction. We refer to [Aic01b, Aic01a] where we have demonstrated that even sequences of interactive test-cases, so called scenarios, are abstractions of interactive system specifications. In the same work we have shown how test-synthesis rules can be formulated for deriving test-partitions or scenarios. In the next section we concentrate on test-synthesis based on contract mutation.

4. Contract-based Mutation Testing

This section shows how our abstraction view of test-cases relates to mutation testing and how it can be generalized to contracts. First, it is analyzed what kind of contract mutations are useful. Then, the criterion for selecting adequate test-cases is formulated. Finally, a precise definition of test-coverage for mutation testing is presented.

4.1. Contract Mutation

In analogy to program mutation in the traditional approaches of mutation testing, we produce a mutant contract by introducing small changes to the formal contract definition. Then, we select test-cases that are able to detect the introduced mutations. In the previous section it has been shown that such a test-case must be an abstraction of the original contract.

An example shows how mutation testing can be applied to contracts.

Example 4.1. Consider the following example of a contract *Min* for computing the minimum of two numbers:

$$\text{Min} \triangleq \text{if } x \leq y \text{ then } [z := z' | z' = x] \text{ else } [z := z' | z' = y]$$

In mutation testing, the assumption is made that programmers produce small errors. A common fault made by programmers is that operators are mixed. In this example a possible fault would be to implement the \geq operator instead of the \leq . Hence, a mutation operator

$$m_1(\dots \leq \dots) = \dots \geq \dots$$

is defined that changes all occurrences of \leq in a contract to \geq . Applying this mutant operator $m_1.\text{Min} = \text{Min}_1$ computes the following mutant:

$$\text{Min}_1 \triangleq \text{if } x \geq y \text{ then } [z := z' | z' = x] \text{ else } [z := z' | z' = y]$$

Now we can find a test-case $\{x < y\}; [z := z' | z' = x]$ (here a partition test-case) that is able to detect this error due to a mutated requirement in an implementation of the *Min* specification. □

However, not all changes in a contract actually produce mutants that reflect errors in the original contract. This effect in program mutations has been reported by DeMillo et al [DLS78] and it holds for general contracts, too.

Consider again our previous contract:

Example 4.2. A mutation m_2 might change the \leq operator in *Min* to $<$ leading to another mutant *Min*₂:

$$\text{Min}_2 \triangleq \text{if } x < y \text{ then } [z := z' | z' = x] \text{ else } [z := z' | z' = y]$$

The mutant *Min*₂ is equivalent to *Min* from an extensional view of a tester. Consequently, no test-case will be able to detect this mutation. We define such mutants as not *useful*. □

Definition 4.1. (useful mutant) A *useful mutant* of a contract C is a mutated contract C_i such that there exists a test-case $TC_i \sqsubseteq C$ that is able to distinguish an implementation of C_i from an implementation of C . \square

The formal interpretation of the term implementation in the definition above leads to the next insight: Mutations that are refinements of the original contract are not useful.

Theorem 4.1. A mutant C_i of a contract C is a *useful mutant* iff $C \not\sqsubseteq C_i$. \square

As a consequence of this theorem we are able to distinguish the following two kinds of useful mutants of a contract C :

- *abstract mutants* $C_i \sqsubseteq C$,
- *unrelated mutants* $C_i \not\sqsubseteq C$ and $C_i \not\supseteq C$.

Why abstraction is considered as a useful mutation for contracts becomes clearer when the criterion for selecting test-cases is explained in the following.

4.2. Selecting Test-Cases

A strategy for selecting test-cases is always based on a hypothesis about faults in a program. The hypothesis of mutation testing can be divided into three parts:

1. a set of mutation operators M explicitly represents the knowledge of typical errors that occur in a program in a specific application domain.
2. usually, programmers create programs that are close to being correct: the small error assumption.
3. the *coupling effect*: test data that distinguish all non-correct programs differing from a correct one by only simple errors are so sensitive that they also implicitly distinguish more complex errors.

Based on this hypothesis, *adequate test-cases* are designed such that they are able to distinguish the *useful mutants*.

Definition 4.2. (adequate test-case) A test-case for a contract C is called *adequate* with respect to a set of mutation operators M if it is able to distinguish at least one useful mutant M_i , with $m.C = M_i$ and $m \in M$. \square

Applied to executable contracts, usually called programs, adequacy is checked by test-execution. But what does *adequacy* mean with respect to our general framework of possibly non-executable contracts? The answer can be given using the notion of abstraction.

Example 4.3. Consider again the *Min* example and its useful mutant Min_1 . Is the proposed partition test-case adequate? And if yes, why? In order to answer this questions, first, it must be shown that $\{x < y\}; [z := z' | z' = x]$ is indeed a proper test-case of *Min*. From Section 3 we know that this corresponds to checking the abstraction relation between *Min* and the (partition) test-case. And in fact

$$Min \sqsupseteq \{x < y\}; [z := z' | z' = x]$$

holds. A closer look on the mutant Min_1 reveals the fact that this abstraction property does not hold for this test-case. We have

$$Min_1 \not\sqsupseteq \{x < y\}; [z := z' | z' = x]$$

\square

This example shows the central criterion for an adequate test-case in mutation testing: a test-case must be an abstraction of the original contract, but must not be an abstraction of one useful mutant. We denote the fact that a test-case TC is adequate, with respect to a set of mutation operators M and a contract C , by $C \sqsupseteq_M TC$.

We are now able to formulate an abstraction rule for deriving mutation test-cases for a given contract and its mutants.

Theorem 4.2. Given a contract C and a mutation C_i , generated by applying a mutation operator m such that $m.C = C_i$ holds, then, a test-case TC is adequate iff it satisfies the synthesis rule

$$\frac{m \in M, m.C = C_i, C \not\sqsubseteq C_i, \quad C \sqsupseteq TC, C_i \not\sqsupseteq TC}{C \sqsupseteq_M TC}$$

□

In fact, the third premise is redundant since the two abstraction properties $C \sqsupseteq TC$ and $C_i \not\sqsupseteq TC$ guarantee by transitivity of \sqsupseteq that C_i is not a refinement of C . However, since it is our intention to stress the necessity of useful mutants, we keep the additional premise of usefulness in the prove rule above. Actually, this redundancy demonstrates the absence of adequate test-cases for mutants that are not *useful*.

This answers also the previous question, why mutants C_i that are abstractions of the original contract C , thus $C \sqsupseteq C_i$, are considered as *useful*: because an adequate test-case TC can be found such that $C \sqsupseteq TC$ but $C_i \not\sqsupseteq TC$ holds.

Corollary 4.1. A mutant C_i of C is *useful* if $C \sqsupseteq C_i$ holds.

Proof.

A mutant C_i is useful iff an adequate test-case TC exists. Thus we have to show the existence of a TC with $C \sqsupseteq TC$ and $C_i \not\sqsupseteq TC$ under the assumption that $C \sqsupseteq C_i$.

This can be proved by contradiction. Assume that no *useful* test-case TC exists if $C \sqsupseteq C_i$. That means that every test-case TC_k in the set of possible test-cases $\{TC_k | C \sqsupseteq TC_k\}$ is also an abstraction of the mutant C_i , thus $C_i \sqsupseteq TC_k$ holds. The exhaustive test-suite $\{TC_k | C \sqsupseteq TC_k\}$ is in fact equivalent to the contract C .

Since the input domain of C_i must be equal or less than C , the test-cases $\{TC_k | C \sqsupseteq TC_k\}$ also represent an exhaustive test of C_i , covering the whole domain of C_i .

Consequently the equivalence of the two contracts follows: $C = C_i$, which contradicts the assumption $C \sqsupseteq C_i$.

□

4.3. Coverage

The synthesis rule above shows that a test-case for finding errors in a mutant, has to be, first, a correct test-case of the original contract, second, it must not be an abstraction of the mutated contract. Consequently, if only *useful mutants* are considered, then the test-cases that are abstractions of the mutations do not cover the mutated parts of a contract. Therefore, the coverage criterion of a collection of test-cases can be defined formally:

Definition 4.3. (mutation coverage) Given a set of test-cases T for a contract C and a set of mutation operators M , then the full coverage with respect to the mutation operators M , called *mutation coverage*, is reached iff

$$\forall m \in M. (C \not\sqsubseteq m.C \Rightarrow \exists tc \in T. (C \sqsupseteq tc \wedge m.C \not\sqsupseteq tc))$$

□

This represents a new approach to the quality criterion for test-cases based on mutations. It applies to general contracts as specifications and programs. In the following, a more interesting example involving arrays serves to demonstrate its application.

5. Example involving Arrays

5.1. Specification Contract

The following example contract for finding the minimum number in a given array A is a simplified version taken from [BvW98].

Example 5.1. Assume that we want to find an index i pointing to the smallest element in an array $A[1..n]$, where $n = \text{len}.A$ is the length of the array and $1 \leq n$ (so the array is nonempty).

We define the predicate $\text{minat}.i.A$ to hold when the minimum value in A is found at $A[i]$.

$$\text{minat}.i.A \triangleq 1 \leq i \leq \text{len}.A \wedge (\forall j \mid 1 \leq j \leq \text{len}.A. A[i] \leq A[j])$$

Then the contract

$$MIN \triangleq \{1 \leq \text{len}.A\}; [i := i' \mid \text{minat}.i'.A]$$

constitutes the problem of finding an index i with the minimum $A[i]$ in A .

In this example only input-output test-cases are considered. We define

$$\text{TC } i \ o \triangleq \{A = i\}; [i := i' \mid i' = o]$$

$$\text{TC } i \ \omega \triangleq \{A = i\}; [i := i' \mid \omega]$$

to be the corresponding deterministic and non-deterministic test-case contracts with the input array A and the output index i containing the minimum of the array.

A first test-case for MIN might be chosen as follows:

$$T_1 \triangleq \text{TC } [2, 1, 3] \ 2$$

How sensitive is this test-case? By inspection, we notice that if an error had occurred in the implementation of the relational operation $A[i] \leq A[j]$ the test-case T_1 would have distinguished those errors. That means that for example, implementations of the following two mutants can be distinguished:

$$\text{minat}_1.i.A \triangleq 1 \leq i \leq \text{len}.A \wedge (\forall j \mid 1 \leq j \leq \text{len}.A. A[i] \boxed{\geq} A[j])$$

$$MIN_1 \triangleq \{1 \leq \text{len}.A\}; [i := i' \mid \text{minat}_1.i'.A]$$

$$\text{minat}_2.i.A \triangleq 1 \leq i \leq \text{len}.A \wedge (\forall j \mid 1 \leq j \leq \text{len}.A. A[i] \boxed{<} A[j])$$

$$MIN_2 \triangleq \{1 \leq \text{len}.A\}; [i := i' \mid \text{minat}_2.i'.A]$$

The mutant MIN_2 is subtle since minat_2 is equivalent to **false**. This means that its weakest-precondition $wp.MIN_2.q = \text{false}$, that denotes the most abstract contracts in the hierarchy of contracts equivalent to **abort**. Since, like every contract, T_1 is a refinement of **abort** and $T_1 \neq \text{abort}$, it cannot be an abstraction of MIN_2 . Which proves that the test-case is able to detect this kind of error. This shows that our criterion based on abstraction also holds for the trivial case of non-terminating mutants.

Similarly, the test-case detects a swapping of the index variables resulting in a mutated expression $A[j] \leq A[i]$.

However, is T_1 able to detect a mutation in the range definition of j ? Consider a corresponding mutation of minat :

$$\text{minat}_3.i.A \triangleq 1 \leq i \leq \text{len}.A \wedge (\forall j \mid 1 \leq j \boxed{<} \text{len}.A. A[i] \leq A[j])$$

This mutation corresponds to a common implementation error in loops, where not the whole array is searched through due to an error in the termination condition. In fact, T_1 is not able to distinguish this kind of error, since the mutant MIN_3 is a correct refinement of T_1 (or T_1 an abstraction of MIN_3). In order to violate the abstraction relation, the minimum must be at the last position in the array. Thus, a second test-case is needed in order to distinguish implementations of MIN_3 :

$$T_2 \triangleq \text{TC } [2, 3, 1] \ 3$$

This test-case is an example that would not have been found by simple path analysis. It indicates that an error-based strategy is more efficient in detecting errors in commonly used data-centered applications than path-based strategies.

Similarly, an additional test-case for detecting an error in the lower bound of the array range must be designed.

$$T_3 \triangleq \text{TC } [1, 2, 3] \ 1$$

The test-cases T_2, T_3 also cover the typical one-off mutations effecting the range of i :

$$\text{minat}_{4,i}.A \triangleq \boxed{2} \leq i \leq \text{len}.A \wedge (\forall j \mid 1 \leq j \leq \text{len}.A. A[i] \leq A[j])$$

$$\text{minat}_{5,i}.A \triangleq 1 \leq i \leq \boxed{(\text{len}.A - 1)} \wedge (\forall j \mid 1 \leq j \leq \text{len}.A. A[i] \leq A[j])$$

One might think that test-case T_1 becomes redundant since T_2, T_3 can distinguish all mutants so far. However, we did not yet take a mutation of the assignment variable itself into account. Consider the mutation:

$$\text{MIN}_6 \triangleq \{1 \leq \text{len}.A\}; [i := \boxed{A[i']} \mid \text{minat}.i'.A]$$

This contract represents the error that not the index, but the minimum itself is assigned to variable i . Only, test-case T_1 is able to detect this error, since its minimum does not equal the corresponding index. Of course, more clever values for $A[1]$ to $A[3]$ might be chosen such that T_1 can be excluded from the test-suite. However, having one typical case in addition to extreme values is considered to be good practice for demonstrating the basic functionality (see also [Bei90]).

In the following, it is analyzed how refinement introduces new mutation test-cases.

5.2. Implementation Contract

Consider the following implementation (refinement) of MIN using a guarded iteration statement:

$$\begin{aligned} \text{MIN} \sqsubseteq \text{MINI} \triangleq & \text{begin var } k := 2; i := 1; \\ & \text{do } k \leq n \wedge A[k] < A[i] \rightarrow i, k := k, k + 1 \\ & \quad [] k \leq n \wedge A[k] \geq A[i] \rightarrow k := k + 1 \\ & \text{od} \\ & \text{end} \end{aligned}$$

It turns out that the test-suite T_1, T_2, T_3 derived from the specification contract is able to distinguish all mutants of $MINI$, except

$$\begin{aligned} \text{MINI}_1 \triangleq & \text{begin var } k := \boxed{1}; i := 1; \\ & \text{do } k \leq n \wedge A[k] < A[i] \rightarrow i, k := k, k + 1 \\ & \quad [] k \leq n \wedge A[k] \geq A[i] \rightarrow k := k + 1 \\ & \text{od} \\ & \text{end} \end{aligned}$$

$$\begin{aligned} \text{MINI}_2 \triangleq & \text{begin var } k := 2; i := 1; \\ & \text{do } k \leq n \wedge A[k] \boxed{\leq} A[i] \rightarrow i, k := k, k + 1 \\ & \quad [] k \leq n \wedge A[k] \geq A[i] \rightarrow k := k + 1 \\ & \text{od} \\ & \text{end} \end{aligned}$$

Inspection shows that $MINI_1$ is in fact functionally equivalent to $MINI$ and thus not a *useful mutant*. It is just less efficient.

The implementation $MINI_2$ is a second refinement of MIN and thus the test-cases derived from MIN are not able to distinguish it from MIN . A new test-case might be derived that is designed to detect the difference.

$$T_4 \triangleq \text{TC } [1, 1, 3] \ 1$$

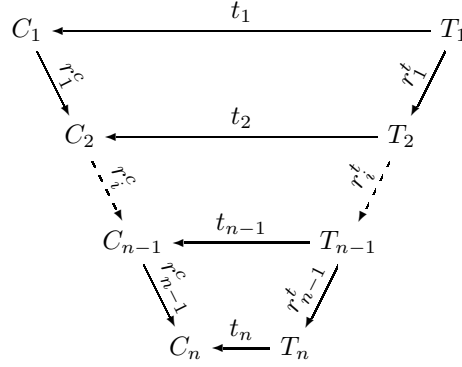


Fig. 2. The V-Diagram.

Note, that this test-case is not an abstraction (and thus not a correct test-case) of MIN . The specification MIN does not define which minimum should be returned, and would need a non-deterministic test-case $\text{TCp } [1, 1, 3]$ ($i = 1 \vee i = 2$).

At this point in the development, the tester has to decide, if he wishes to refine the specification MIN such that always the first occurrence of the minimum should be returned. Such a refined version $MIN \sqsubseteq \text{FirstMIN}$ is presented below:

$$\text{first}.i.A \triangleq \forall j \mid \text{minat}.j.A . i \leq j$$

$$\text{FirstMIN} \triangleq \{1 \leq \text{len}.A\}; [i := i' \mid \text{minat}.i'.A \wedge \text{first}.i'.A]$$

It is interesting to note that even test-case T_4 would have been found by mutating the specification FirstMIN . The mutation

$$\text{first}.i.A_1 \triangleq \forall j \mid \text{minat}.j.A . i \boxed{\geq} j$$

needs test-case T_4 in order to detect the mutation FirstMIN_1 .

We can conclude that in this example, specification-based mutation testing was as powerful, with respect to coverage, as the program-based mutation technique. The difference is that the specification-based technique can be applied much earlier in the development cycle than the program-based one.

6. Testing under Refinement

In the previous Example 5.1 it could be seen that mutating the specification was sufficient to derive the adequate test-cases in order to test if an implementation meets this specifications. However, during the implementation process one is not just interested in the original specification, but one also wants to test if the design decisions have been correctly implemented during the refinement process. That means that new test-cases have to be designed for the refined version of a contract. To be more precise, new test-cases have to be added to the existing ones. This section shows that our notion of mutation test-cases defines the properties of these new test-cases to be added under refinement.

The relations between contracts and their test-cases in a refinement process are shown in the diagram in Figure 2. Note the close similarity to the V-process model (V-model), that is a derivative of the waterfall model where specification, design, and development activities are explicitly linked to testing [Tuc96]. The left-hand side of the V in Figure 2 represents the step-wise development of a specification contract C_1 into an implementation C_n . On the right-hand side of the V, the corresponding test-cases are shown. All the arrows in the diagram denote refinement pointing from the abstract to the more concrete. The arrows labeled with r represent the refinement of contracts (r_i^c) and the refinement of the associated test-cases (r_i^t). The t -labeled arrows show the refinement relation between test-cases and their associated contracts. The diagram explicitly illustrates the fact that each contract C_i must be a refinement of their test-cases T_i . This represents the fact that test-cases are a special form of specification.

Mathematically, this *V-diagram* in Figure 2 can be interpreted as a commuting diagram in the category of contracts, where objects are contracts (here specifications, implementations and test-cases) and arrows represent the refinement relation. Then, the commutations in the V-diagram represent $n - 1$ equations for $1 \leq i < n$:

$$r_i^c \circ t_i = t_{i+1} \circ r_i^t$$

This equation states that, given two sets of test-cases, T_i for a contract C_i and T_{i+1} for a refined contract C_{i+1} such that $C_i \sqsubseteq C_{i+1}$, then the test-cases T_{i+1} must be a refinement of T_i or $T_i \sqsubseteq T_{i+1}$. This means that under refinement the test-cases have to be refined as well.

However, the refinement relation \sqsubseteq alone is a too weak criterion in order to design new test-cases. For example, the relation \sqsubseteq includes the equality of contracts $=$. This means that the same test-cases could be used for the refined contract.

The problem of finding new test-cases can be formulated as follows:

Theorem 6.1. Assume that a contract C and its refinement C' is given. Furthermore, it is assumed that test-cases T for C have been correctly designed. Then the refined test-cases T' for testing C' have the general form

$$T' = (T \sqcup T'_{new})$$

for arbitrary new test-cases T'_{new} such that $C' \sqsupseteq T'_{new}$.

Proof. It must be shown that $(T \sqcup T'_{new})$ is a refinement of T and that they are test-cases of C' , thus $T \sqsubseteq (T \sqcup T'_{new})$ and $C' \sqsupseteq (T \sqcup T'_{new})$ must hold.

The first property $T \sqsubseteq (T \sqcup T'_{new})$ holds for any T'_{new} by definition of the join operator \sqcup in the lattice of contracts.

Similarly, by the definition of \sqcup , the second property $C' \sqsupseteq (T \sqcup T'_{new})$ holds iff $C' \sqsupseteq T$ and $C' \sqsupseteq T'_{new}$. Trivially, $C' \sqsupseteq T'_{new}$ is a premise. $C' \sqsupseteq T$ follows from the facts that C' is a refinement of C , thus $C' \sqsupseteq C$, and that T are correct test-cases designed for C , thus $C \sqsupseteq T$ holds.

This proves that the diagram commutes for refined test-cases $(T \sqcup T'_{new})$.

□

Theorem 6.1 shows that the problem of finding test-cases for a refined specification can be reduced to the question of finding additional new test-cases. However, not all new test-cases are useful. In order to be economical, they should cover the newly added parts or properties of the refined contract C' . It is interesting that the problem of finding useful new test-cases T'_{new} can be mapped to the mutation testing approach above.

Here, in our quest for useful new test-cases T'_{new} for a contract C' and $C \sqsubseteq C'$, we consider C as a useful mutant. Then T'_{new} should be adequate test-cases that are able to distinguish the abstract mutant from its refinement.

Thus the criterion for useful new test-cases is an adaptation of the abstraction rule for deriving mutation test-cases:

Theorem 6.2. Given a contract C and its refinement C' . Furthermore, test-cases T for C have been correctly designed. Then the refined test-cases T' for testing C' have the general form

$$T' = (T \sqcup T'_{new})$$

and

$$C' \sqsupseteq T'_{new} \wedge C \not\sqsupseteq T'_{new}$$

Alternatively an abstraction rule for producing useful new test-cases can be formulated as follows:

$$\frac{C \sqsubseteq C', C \sqsupseteq T \quad C' \sqsupseteq T'_{new}, C \not\sqsupseteq T'_{new}}{C' \sqsupseteq_t T \sqcup T'_{new}}$$

□

It can be seen that $C \not\sqsupseteq T'_{new}$ is the central property for adding new test-cases that are able to distinguish between contract C and its refinement C' or between their implementations, respectively. An example serves to illustrate this fact.

Example 6.1. In Example 5.1 three test-cases for the specification MIN have been designed by our mutation strategy:

$$T \triangleq T_1 \sqcup T_2 \sqcup T_3$$

In addition, we have discussed that an additional non-deterministic test-case $\text{TCp } [1, 1, 3]$ ($i = 1 \vee i = 2$) could be included in the test-set T .

Which test-cases should be added in order to test the refinement $MINI$? The rule in Theorem 6.2 points us to the answer, since a correct test-case of $MINI$ should be found that is not an abstraction of MIN . It is straightforward to reconsider the non-deterministic test-case, since the implementation $MINI$ is deterministic. Hence, new test-cases should be able to distinguish the deterministic from the non-deterministic version (mutant). Consequently,

$$T'_{new} \triangleq \text{TC } [1, 1, 3] \ 1$$

is such a useful new test-case. Our framework actually allows us to prove this fact by showing that $MINI \sqsubseteq T'_{new}$ and $MIN \not\sqsubseteq T'_{new}$.
□

7. Concluding Remarks

Summary. In this article we have presented a novel approach to both the understanding and the application of mutation testing. The formal development concepts, like refinement and abstraction, have been the tools to (1) define when a mutant can be successfully distinguished by any test-case, (2) give a criterion when a given test-case is able to detect an error, and (3) what full coverage with respect to mutation testing precisely means. In addition we could define a criterion for adding new test-cases under refinement. The refinement calculus has provided the formal framework in order to extend mutation testing to a more general language of contracts. An example has served to illustrate our view on mutation testing. This example has shown how an abstract contract may be effectively used to derive test-cases by analyzing the possible mutations.

Related Work. Testing from formal specifications has become a popular area of research. Nowadays, nearly every formal specification language, like for example VDM [DF93], Z, B [LPU02], or Lotos [GJ98], comes with a test-case generation approach. A general summary of work on formal methods and testing can be found in [Aic01a]. In contrast to previous work, we try to formalize the problem of selecting adequate test-cases as an abstraction problem.

It should be stressed that the relation of testing and refinement is well known. 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 labeled transition systems [Tre92, Tre99]. However, these theories do not focus on the use of abstraction in order to select a subset of test-cases.

To our best knowledge, 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 author's doctoral thesis [Aic01a]. It was in this thesis, where the idea of mutation testing has been presented the first time. Although others worked on specification-based mutation testing before, the use of a refinement relation in order to define adequate mutation test-cases and coverage is new.

Very early, Stocks applied mutation testing to formal 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 for his specification mutations is the exchange of the join operator \cup of sets with intersection \cap . From these mutants, test-cases are generated for demonstrating that the implementation does not implement one of the specification mutations, but Stocks does not address issues like useful mutations, coverage or refinement. Nor does he discuss why some test-cases are more successful in finding errors than others. Furthermore, in contrast to our approach, he does not use the mutants for finding adequate test-cases, but use the mutations to assess test-cases generated through path-analysis techniques. However, finding a small and yet powerful set of test-cases is one of the main ideas of mutation testing (see [DLS78]).

Recently, mutation testing has been used to generate test-cases for security-critical systems by Wimmel and Jürjens [WJ02]. Their aim is to find those test sequences that are most likely to find vulnerabilities. Here, mutants of an AUTOFOCUS model Ψ are generated. Then, a constraint solver is used to search for a test sequence that satisfies a mutated system model Ψ' (a predicate over traces) and that does not satisfy a security requirement ϕ . If a test-case for $\Psi' \wedge \neg\phi$ can be found, then the mutation Ψ' introduces a vulnerability and the test-case shows how it can be exploited. Actually, this approach is a special instantiation of our more general refinement technique. We can easily translate their approach by observing that $\Psi' \wedge \neg\phi = \neg(\Psi' \Rightarrow \phi)$. Since in their predicative framework implication represents refinement, this is equal to $\Psi' \not\sqsubseteq \phi$. Since ϕ can be considered as a set of test-cases, their constraint solver generates a test that is not an abstraction of the mutant Ψ' . This is exactly our criterion for generating test-cases. However, this example shows that it is more general to use refinement as the general notion to define test-selection criteria, because such a testing framework can be applied to several logical frameworks with different definitions of refinement.

Discussion. The main advantage of contract mutation over pure program mutation is that it can be applied very early in the design process. In this early phases test-designers could apply their experience of what kind of requirements are difficult to grasp and, therefore, could be wrongly implemented. It allows a tester to design test-cases in advance or in parallel to the implementation process. The array-example should illustrate that even in such an early phase it is possible to design test-cases that focus on common implementation errors rather than on covering the cases (partitions) in a formal specification. We believe that this technique could be related to safety analysis techniques [Lev95] that also focus on faulty behavior.

The array example also indicates that this approach needs tool support in order to scale up to larger specifications. The necessary refinement and non-refinement checks can be automated by verification tools like theorem provers, BDD-checkers or constraint solvers. For the simple form of test-cases used in this article full automation should be possible. A question of pragmatic concern is how much the mutation process should be automated. If faults are introduced fully automatically, the approach loses its advantage of including a tester's insight which parts of a specification might cause problems. However, a semiautomatic approach with a graphical user-interface that supports the localization and input of mutations would be worthwhile. Tool support is part of our future work.

Our approach to mutation testing is very general and not limited to the contract language used in this paper. Any formal framework with a refinement concept should be suitable for selecting test-cases following our technique. Hence, other frameworks, like the Unifying Theories of Programming [HH98] could be considered. Automation is an obvious direction for future research. Another area of future work, is the extension to interactive systems and thus to scenario generation. Stimulating feedback from the program synthesis community might lead to further advances in the automation of contract-based mutation testing techniques.

References

- [Aic01a] Bernhard K. Aichernig. *Systematic Black-Box Testing of Computer-Based Systems through Formal Abstraction Techniques*. PhD thesis, Institute for Software Technology, TU Graz, Austria, January 2001. Supervisor: Peter Lucas.
- [Aic01b] Bernhard K. Aichernig. Test-case calculation through abstraction. In J. N. Oliveira and Pamela Zave, editors, *Proceedings of FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, March 2001, Berlin, Germany*, volume 2021 of *Lecture Notes in Computer Science*, pages 571–589. Springer-Verlag, 2001.
- [Aic01c] Bernhard K. Aichernig. Test-Design through Abstraction: a Systematic Approach Based on the Refinement Calculus. *Journal of Universal Computer Science*, 7(8):710 – 735, August 2001.
- [Bei90] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 2nd edition, 1990.
- [BMvW99] Ralph Back, Anna Mikhajlova, and Joakim von Wright. Reasoning about interactive systems. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 — Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999, Proceedings, Volume II*, volume 1709 of *Lecture Notes in Computer Science*, pages 1460–1476. Springer-Verlag, 1999.
- [BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus, a Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.
- [DF93] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In J.C.P. Woodcock and P.G. Larsen, editors, *Proceedings of FME'93: Industrial-Strength Formal Methods, International Symposium of Formal Methods Europe, April 1993, Odense, Denmark*, pages 268–284. Springer-Verlag, April 1993.

- [DLS78] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.
- [DNH84] R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.
- [GJ98] Marie-Claude Gaudel and Perry R. James. Testing algebraic data types and processes: a unifying theory. *Formal Aspects of Computing*, 10(5 & 6):436–451, 1998.
- [Ham77] Richard G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.
- [HH98] C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall International, 1998.
- [HNS97] Steffen Helke, Thomas Neustupny, and Thomas Santen. Automating test case generation from Z specifications with Isabelle. In J.P. Bowen, M.G. Hinchey, and D. Till, editors, *Proceedings of ZUM'97, the 10th International Conference of Z Users, April 1997, Reading, UK*, volume 1212 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1997.
- [Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall, 1985.
- [Hoa03] Tony Hoare. Towards the Verifying Compiler. In *The United Nations University/ International Institute for Software Technology 10th Anniversary Colloquium: Formal Methods at the Crossroads, from Panacea to Foundational Support, Lisbon, March 18–21, 2002*. Springer-Verlag, 2003. To be published.
- [HP94] Hans-Martin Hörcher and Jan Peleska. The role of formal specifications in software testing. In *Tutorial Notes of FME'94: Industrial Benefit of Formal Methods, Second International Symposium of Formal Methods Europe, October 1994, Barcelona, Spain*. Formal Methods Europe, October 1994.
- [Lev95] Nancy G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.
- [LPU02] Bruno Legeard, Fabien Peureux, and Mark Utting. Automated boundary testing from Z and B. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *Proceedings of FME 2002: Formal Methods - Getting IT Right, International Symposium of Formal Methods Europe, July 2002, Copenhagen, Denmark*, volume 2391 of *Lecture Notes in Computer Science*, pages 21–40. Springer-Verlag, 2002.
- [Mor94] Carroll Morgan. *Programming from Specifications*. Series in Computer Science. Prentice-Hall International, 2nd edition, 1994.
- [Ste95] Susan Stepney. Testing as abstraction. In J. P. Bowen and M. G. Hinchey, editors, *Proceedings of ZUM '95, the 9th International Conference of Z Users, September 1997, Limerick, Ireland*, volume 967 of *Lecture Notes in Computer Science*, pages 137–151. Springer-Verlag, 1995.
- [Sto93] Philip Alan Stocks. *Applying formal methods to software testing*. PhD thesis, The Department of computer science, The University of Queensland, 1993.
- [Tre92] Jan Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, Universiteit Twente, 1992.
- [Tre99] Jan Tretmans. Testing concurrent systems: A formal approach. In Jos C.M. Baeten and Sjouke Mauw, editors, *CONCUR'99*, volume 1664 of *Lecture Notes in Computer Science*, pages 46–65. Springer-Verlag, 1999.
- [Tuc96] B. Allen Tucker, Jr., editor. *The Computer Science and Engineering Handbook*. CRC press, 1996.
- [WJ02] Guido Wimmel and Jan Jürjens. Specification-based test generation for security-critical systems using mutations. In Chris George and Miao Huaikou, editors, *Proceedings of ICFEM'02, the International Conference of Formal Engineering Methods, October 21–25, 2002, Shanghai, China*, Lecture Notes in Computer Science. Springer-Verlag, 2002.