

# Contract-based Mutation Testing in the Refinement Calculus

Bernhard K. Aichernig<sup>1</sup>

*The United Nations University  
International Institute for Software Technology (UNU/IIST)  
P.O. Box 3058, Macao*

---

## 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.

*Key words:* formal methods, formal specifications, mutation testing, refinement, abstraction, test-case synthesis, test-coverage.

---

## 1 Introduction

The synergy of formal methods and testing have 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 efforts 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 [11]).

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 [2,1,3]. In this paper we focus on mutation testing and its relation to refinement.

---

<sup>1</sup> Email: [bka@iist.unu.edu](mailto:bka@iist.unu.edu)

### 1.1 Mutation Testing

Mutation testing is a fault-based testing technique introduced by Hamlet [9] and DeMillo et al [7]. It is a means of assessing test suites. When a program passes all tests in a suite, mutant programs are generated and 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. A hypothesis of this technique is that programmers only make small errors.

### 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 [2,1], we have shown that correct test-cases  $T$  can be defined as abstractions of the specifications  $S$ , 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 abstraction, and denoted as:

$$S \sqsupseteq T?$$

Hence, 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 [5] 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 Section 5 an example is used to discuss the proposed mutation technique in more detail. Finally, we draw our conclusions in Section 6.

## 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 [5] 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 is modeled by a global state space  $\Sigma$ . A single state  $x$  in this state space is denoted by  $x : \Sigma$ . 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  $\Sigma$  to the same or another state space  $\Gamma$ .

A relational update  $R : \Sigma \rightarrow \Gamma \rightarrow \mathbf{Bool}$  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 [5,4], 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  $\mu$  and the greatest fix-point operator  $\nu$ , 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 no inconsistencies into the theory of the refinement calculus are introduced by our new definitions.

## 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  $x'$ . Thus, the contract is breached unless  $x \geq 0 \wedge e > 0$  holds in the state initially. If this condition is true, then  $x$  is assigned some value  $x'$  for which  $-e \leq x - x'^2 \leq e$  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, so 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.

Iteration can be specified by recursive contracts  $(\mu X \cdot C)$ . 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  $C$ . 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  $C : (\Gamma \rightarrow \mathbf{Bool}) \rightarrow (\Sigma \rightarrow \mathbf{Bool})$  is a function mapping postcondition predicates to precondition predicates. The set of all predicate transformers from  $\Sigma$  to  $\Gamma$  is denoted by  $\Sigma \mapsto \Gamma \triangleq (\Gamma \rightarrow \mathbf{Bool}) \rightarrow (\Sigma \rightarrow \mathbf{Bool})$ .

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  $\sigma$  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

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

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

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**.
- The *angelic update* definition says that a final state  $\gamma$  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  $\gamma$  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 [5].

## 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  $\Gamma$  being the after state space of the contracts, we have

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

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 [5,14].

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  $C'$  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 [5] 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  $\text{TC}$  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 are indeed pointwise defined formal specifications:

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

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

**Theorem 3.1** *Let  $p : \Sigma \rightarrow \text{Bool}$  be a predicate,  $Q : \Sigma \rightarrow \Gamma \rightarrow \text{Bool}$  a relation on states, and TC  $i \ o$  a test-case with input  $i$  in variable  $x$  and 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 [2].

□

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

Thus, 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 3.3**

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

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, and 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  $\omega : \Sigma \rightarrow \mathbf{Bool}$  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 3.4** *Let  $p : \Sigma \rightarrow \mathbf{Bool}$  be a predicate,  $Q : \Sigma \rightarrow \Sigma \rightarrow \mathbf{Bool}$  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  $\omega$  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  $\omega$  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 the same. 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. [8,15,12,10]). This technique is based on rewriting according 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 3.5** *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 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  $\sqcup$  and  $\{R\}$  can be formulated in a pre-postcondition style (see Theorem 26.4 in [5]).

However, our result that test-cases are abstractions holds for general contract statements involving user interaction. We refer to [2,1] 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 criteria 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:

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

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  $m_1.Min = Min_1$  computes the following mutant:

$$Min_1 \triangleq \text{if } x \boxed{\geq} y \text{ then } [z := z' | z' = x] \text{ else } [z := z' | z' = y]$$

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.  $\square$

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 [7] 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*<sub>2</sub>:

$$Min_2 \triangleq \text{if } x \boxed{<} y \text{ then } [z := z' | z' = x] \text{ else } [z := z' | z' = y]$$

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*.  $\square$

**Definition 4.3 (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 4.4** 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*  $C_i \sqsubset C$ ,
- *unrelated mutants*  $C_i \not\sqsubseteq C$  and  $C_i \not\supseteq C$ .

Why abstraction is considered as a useful mutation for contracts becomes more clear when the criteria 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:

- (i) a set of mutation operators  $M$  explicitly represents the knowledge of typical errors that occur in a program in a specific application domain.
- (ii) usually, programmers create programs that are close to being correct: the small error assumption.
- (iii) the *coupling effect*: test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors.

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

**Definition 4.5 (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 4.6** 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

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

□

This example shows the central criteria 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 4.7** *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*

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

□

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 4.8** *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  $C \sqsupset C_i$ .

This can be proved by contradiction. Assume that no *useful* test-case  $TC$  exists if  $C \sqsupset C_i$ . 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  $C \sqsupset C_i$ .

□

### 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. Consequently, the coverage criteria of a collection of test-cases can be defined formally:

**Definition 4.9 (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

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

□

This represents a new approach to the quality criteria 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 [5].

**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 nonempty).

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

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

Then the contract

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

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

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

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

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:

$$T_1 \triangleq TC \ [2, 1, 3] \ 2$$

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:

$$minat_1.i.A \triangleq 1 \leq i \leq len.A \wedge (\forall j \mid 1 \leq j \leq len.A \bullet A[i] \boxed{\geq} A[j])$$

$$MIN_1 \triangleq \{1 \leq len.A\}; [i := i' \mid minat_1.i'.A]$$

$$minat_2.i.A \triangleq 1 \leq i \leq len.A \wedge (\forall j \mid 1 \leq j \leq len.A \bullet A[i] \boxed{<} A[j])$$

$$MIN_2 \triangleq \{1 \leq len.A\}; [i := i' \mid minat_2.i'.A]$$

The mutant  $MIN_2$  is subtle since  $minat_2$  is equivalent to **false**. Hence, the weakest-precondition  $wp.MIN_2 = \mathbf{false}$ , which 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  $T_1 \neq \mathbf{abort}$ , 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 criteria 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  $minat$ :

$$minat_3.i.A \triangleq 1 \leq i \leq len.A \wedge (\forall j \mid 1 \leq j \boxed{<} len.A \cdot A[i] \leq A[j])$$

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  $i$ :

$$minat_4.i.A \triangleq \boxed{2} \leq i \leq len.A \wedge (\forall j \mid 1 \leq j \leq len.A \cdot A[i] \leq A[j])$$

$$minat_5.i.A \triangleq 1 \leq i \leq \boxed{(len.A - 1)} \wedge (\forall j \mid 1 \leq j \leq len.A \cdot A[i] \leq A[j])$$

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:

$$MIN_6 \triangleq \{1 \leq len.A\}; [i := \boxed{A[i']} \mid minat.i'.A]$$

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 [6]).

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:

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

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

$$\begin{aligned} MINI_1 \triangleq & \text{ begin var } k := \boxed{1}; i := 1; \\ & \text{ do } k \leq n \wedge A[k] < A[i] \rightarrow i, k := k, k + 1 \\ & \quad [] k \leq n \wedge A[k] \geq A[i] \rightarrow k := k + 1 \\ & \text{ od} \\ & \text{ end} \end{aligned}$$

$$\begin{aligned} MINI_2 \triangleq & \text{ begin var } k := 2; i := 1; \\ & \text{ do } k \leq n \wedge A[k] \boxed{\leq} A[i] \rightarrow i, k := k, k + 1 \\ & \quad [] k \leq n \wedge A[k] \geq A[i] \rightarrow k := k + 1 \\ & \text{ od} \\ & \text{ end} \end{aligned}$$

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.

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

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  $MIN \sqsubseteq FirstMIN$  is presented below:

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

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

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

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

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 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 criteria when a given test-case is able to detect an error, and (3) what full coverage with respect to mutation testing precisely means. The refinement calculus has provided the formal framework in order to extend mutation testing to the more general contract language. 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. Surprisingly, the analysis of the implementation did not add any new test-cases, that could not have been found on the specification level.

### Related Work.

To our current knowledge only Stocks has applied mutation testing to formal specifications [15]. 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. However, 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 [7]). A summary of work on formal methods and testing can be found in [1].

### Discussion.

The main advantage of contract mutation over pure program mutation is that it can be applied very early in the design process. 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 [13] that also focus on faulty behavior. In our opinion, the fact that no additional test-case has been needed for the implementation can be related to the coupling effect (see Section 4.2). However, more abstract specifications and more complex implementations have to be considered if this holds in general.

### Future Work.

We hope that our newly gained insight into the principles of mutation testing, will lead to tools that are able to partly automate the test-selection process. This will be one of our next research topics. 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 techniques.

## References

- [1] 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.
- [2] Bernhard K. Aichernig. Test-case calculation through abstraction. In *Proceedings of Formal Methods Europe 2001, FME 2001, March 12–16 2001, Berlin, Germany*, Lecture Notes in Computer Science. Springer Verlag, 2001.
- [3] 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.
- [4] 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*. Springer Verlag, 1999.

- [5] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus, a Systematic Introduction*. Graduate Texts in Computer Science. Springer Verlag, 1998.
- [6] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 2nd edition, 1990.
- [7] 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.
- [8] 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, *FME'93: Industrial-Strength Formal Methods*. Springer Verlag, April 1993.
- [9] Richard G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.
- [10] Steffen Helke, Thomas Neustupny, and Thomas Santen. Automating test case generation from Z specifications with Isabelle. In *ZUM'97*, 1997.
- [11] 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, 2002. To be published.
- [12] Hans-Martin Hörcher and Jan Peleska. The role of formal specifications in software testing. In *Tutorial Notes for the FME'94 Symposium*. Formal Methods Europe, October 1994.
- [13] Nancy G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.
- [14] Carrol C. Morgan. *Programming from Specifications*. Series in Computer Science. Prentice-Hall International, 1990.
- [15] Philip Alan Stocks. *Applying formal methods to software testing*. PhD thesis, The Department of computer science, The University of Queensland, 1993.