Journal of Universal Computer Science, vol. 7, no. 8 (2001), 710-735
submitted: 18/5/01, accepted: 21/8/01, appeared: 28/8/01 © Springer Pub. Co.

Test-Design through Abstraction

A Systematic Approach Based on the Refinement Calculus

Bernhard K. Aichernig
(Graz University of Technology, Austria
aichernig@ist.tugraz.at)

Abstract: This article discusses the calculation of test-cases for interactive systems. A novel approach is presented that treats the problem of test-case synthesis as a formal abstraction problem. It is shown that test-cases can be viewed as formal contracts and that such test-cases are in fact abstractions of requirements specifications. The refinement calculus of Back and von Wright is used to formulate abstraction rules for calculating correct test-cases from a formal specification. The advantage of this abstraction approach is that simple input-output test-cases, as well as testing scenarios can be handled. Furthermore, different testing strategies like partition testing and mutation testing can be formulated in one theory.
Key Words: testing, test-case generation, formal methods, refinement calculus.
Category: D.2.1, D.2.5

1 Introduction

Software testing is a challenging task. It should evaluate the software to demonstrate that it meets the requirements. This is difficult, both in theory and in practice.

From theory it is known that testing cannot guarantee the absence of failures in complex computer-based systems. The reason is the non-continuous nature of software. For example, if two test-points in the input-space of a piece of software have been tested successfully, nothing is known about the behavior of the software for other inputs situated between these test-points. This is fundamentally different to other engineering disciplines, mainly dealing with continuous physical domains.

Furthermore, the testing process is strongly dependent on the quality of the requirements documentation available to a tester. User requirements that have not been explicitly documented cannot be systematically tested. It is known from practice that approximately 60 percent of all defects introduced in the requirements phase are due to missing requirements (see page 6 of [20]).

However, software is increasingly being used in our daily life as well as in safety-critical systems where failure could be life threatening. This enforces the need for an improved systematic discipline of testing which must be scientifically justifiable. Since, requirement documents written in a natural language tend to be incomplete, ambiguous and unsound, such a discipline of testing has to be based on a specification language with a formal semantics.

The remainder of this section briefly summarizes the experiences and motivations that cumulated into our formal approach to testing. In Section 2 the main idea of this paper is presented informally. The formal system of the refinement calculus that is used to define our testing theory is explained in Section 3. Section 4 demonstrates that test-cases can be viewed as abstractions of requirement specifications. Section 5 shows that this holds for test-case scenarios, too. Based on this observation, several testing strategies can be formulated (Section 6 and Section 7). Finally, the conclusions are drawn in Section 8.

1.1 Experience and Motivation

The work pesented in the following sections is mainly motivated by two projects that have been carried out in the highly safety-critical domain of voice-communication for air-traffic control. Here, executable VDM specifications have been used for executing previously designed system-level test-cases.

Project I detected 64 problems in the requirement documentation of the existing voice communication system. Furthermore, an unacceptable low coverage rate of 80 % has been analyzed by testing the abstract prototype with the existing system-level test-cases. As a consequence, additional test-cases have been derived from the formal specification. The work has been done in less than 13 man-weeks [15, 16, 17, 1].

Project II detected 33 problems in the requirements documentation of the new voice communication network under development. This time the existing test-cases covered 100 % of the abstract prototype, but the 65 test-cases with 200 test-steps contained 16 faults. Since almost all detected faults occurred in different test-cases, this means that approximately 25 % of the test-cases have been erroneous. The effort took 14 man-weeks [3, 1].

These results indicate the efficient applicability of executable specifications in practice. The high number of faults in the requirements reflect the fact that requirements engineering is a difficult task. It has been shown that formal specifications may help to raise the quality of both, the requirements and the system-level test cases. That supervised students could do this work underpins our opinion that such methods can be adopted by practicing engineers. Furthermore, the author thinks that the efforts are justifiable, especially for such safety-critical applications.

However, the experiments also showed that more can be achieved. We learned that executable specifications have the disadvantage that non-experienced users tend to think too operationally. Consequently, the prototypes lack abstraction and incorporate too many design details. However, the engineers liked the test-cases that have been run on the specification to validate our formal models.

In order to get the advantage of such test-cases without the need of executability, we propose a test-case generation method that derives the complex test-cases from a general non-executable formal specification. These test-cases have to be correct with respect to the formal specification. Here, by the correctness of a test-case it is meant that the functionality described by a test-case (for a given input) must be consistent with the functionality described by the formal specification. The reason is that the test-cases should serve as a tool for validating an abstract possibly non-executable formal requirement specification. Test-cases can be easily understood by a customer or engineer who is not familiar with a formal specification language. It will be shown that the general correctness relation for all kinds of test-cases is abstraction.

A further motivation for this work is the need of a general formal framework for certifying test-case generation tools. Simon Burton has a similar motivation in his work on Z that has been recently published [8]. However, our framework is more general as will be seen in the following.

2 The New Idea: 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 goal 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 this section, it is shown that correct test-cases T are 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 is the most important idea in this paper. Its main contribution is to use the theory of Back and von Wright's refinement calculus [5] for formulating abstraction rules for deriving correct test-cases.

The refinement calculus is a theory for reasoning about the correctness and the refinement of programs. In the following it is shown how its program synthesis techniques can be applied to test-case synthesis.

3 The Refinement Calculus

3.1 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 Σ. 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 $f : \Sigma \to \Gamma$ mapping a state space Σ to the same or another state space Γ.

A relational update $R : \Sigma \to \Gamma \to \mathsf{Bool}$ 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 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 no inconsistencies into the theory of the refinement calculus are introduced by our new definitions.

3.2 Example Contracts

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

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

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:

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

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:

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

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:

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

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:

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

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

3.3 Semantics

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

The different roles of the angel and the demon are reflected in the following weakest-precondition semantics. Here q denotes a postcondition predicate and σ a particular state, p is an arbitrary predicate, and R a relation. Following the convention, 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 $f(x)$.

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

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.

– 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 [5].

3.4 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

$$C \sqsubseteq C' \;\triangleq\; \forall q \in (\Gamma \to \mathsf{Bool}) \cdot C.q \subseteq C'.q$$

This ordering relation defines a lattice of predicate transformers (contracts) with the lattice operators meet $\sqcap$ and join $\sqcup$. The top element $\top$ is $\mathsf{magic}.q \triangleq \mathsf{true}$, a statement that is not implementable since it can magically establish every postcondition. The bottom element $\bot$ of the lattice is $\mathsf{abort}.q \triangleq \mathsf{false}$ 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, 19].

Abstraction is dual to refinement. If $C \sqsubseteq C'$, we can interchangeable 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.

4 Test-Cases as 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.

4.1 Input-Output Tests

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

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

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:

$$\mathrm{TCs} \ \triangleq\ \mathsf{TC}\ i_1\ o_1 \sqcup \ldots \sqcup \mathsf{TC}\ i_n\ o_n$$

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 1. *Let $p : \Sigma \to \mathsf{Bool}$ be a predicate, $Q : \Sigma \to \Gamma \to \mathsf{Bool}$ a relation on states, and TC i o a test-case with input i in variable x and output o in variable y. Then*

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

, where

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

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

- *$|f|$ denotes the coercion of a state transformer $f : \Sigma \to \Gamma$ to a relation (here $y := o$). The relation $|f| : \Sigma \to \Gamma \to \mathsf{Bool}$ 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:*

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

Proof. See Appendix.
□

Theorem 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 is observable, which is shown in the following corollary.

Corollary 2. *Let $p : \Sigma \to \mathsf{Bool}$ be a predicate, $Q : \Sigma \to \Gamma \to \mathsf{Bool}$ 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*

$$\{p\};[Q] \sqsupseteq \mathsf{TC}\ i\ o \ \equiv \ p.i \wedge Q.i.o$$

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

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 abstract, and thus easy to understand. Second, they are formal and thus unambiguous.

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

Corollary 3.

$$\mathsf{TC}\ i_1\ o_1 \sqcup \ldots \sqcup \mathsf{TC}\ i_n\ o_n \sqsupseteq TC\ i_k\ o_k$$

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.
□

4.2 Non-Deterministic Test-Cases

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

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

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

Theorem 4. *Let $p : \Sigma \to \mathsf{Bool}$ be a predicate, $Q : \Sigma \to \Sigma \to \mathsf{Bool}$ a relation on states, and $\mathsf{TC2}\ i\ o$ a test-case with input i in variable x and output in variable y such that the output predicate ω holds . Then we have:*

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

□

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.

4.3 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. [10, 21, 14, 13]). 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

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

and

$$\forall i,j \in \{1,\ldots,n\} \mathbin{\bullet} i \neq j \Rightarrow p_i \cap p_j = \emptyset$$

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 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 $\sqcup$.
□

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 inter-action. In order to justify this, user-interaction has to be discussed with respect to testing. The next section will introduce the necessary concepts.

5 Testing Interactive Systems

The synthesis of black-box tests for an interactive system has to consider the possible user actions. Furthermore, simple input-output test-cases are insufficient for practical systems. Moreover, sequences of interactions, called scenarios, are necessary for setting the system under test into the interesting states. Consequently, scenarios of the system's use have to be developed for testing.

Scenarios are gaining more and more popularity in software engineering. The reasons are the same as for other test-cases: Scenarios are abstractions of interactive systems. For a comprehensive introduction into the different roles of scenarios in software engineering see [18]. In this work, the focus is on validation and verification.

5.1 User Interaction

Testing interactive systems, typically involves the selection of a series of parameters. Some of these parameters can be entered directly, some have to be set up, by initiating a sequence of preceding actions. Adequate test-cases should distinguish between these two possibilities of parameter setup. Therefore, simple pre-postcondition contracts are not sufficient to specify test-cases. Moreover, the tester's interaction with the system has to be modeled.

We define an atomic *interaction* IA of a tester, as a composition of the testers system update T and the following system's response Q.

$$\mathrm{IA} \triangleq \{T\};[Q]$$

The fact that we define an atomic interaction by means of angelic and demonic updates does not exclude other contract statements for modeling interaction. Theorem 13.10 in [5] states that {T};[Q] is a normal form, thus arbitrary contract statements can be defined by means of interactions.

In this context a simple input-output test-case $\mathsf{TCI}\ i\ o$ includes the actual setting of the input variable to i.

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

Again the abstraction relation holds for this kind of test-cases.

Theorem 6. *Let $T : \Sigma \rightarrow \Gamma \rightarrow$ Bool and $Q : \Gamma \rightarrow \Theta \rightarrow$ Bool relations on states, and $\mathsf{TCI}\ i\ o$ a test-case with input i in variable x and output o in variable y. Then*

$$\{T\}; [Q] \sqsupseteq \mathsf{TCI}\ i\ o\ \Leftarrow\ |x := i| \subseteq T \wedge Q \subseteq |y := o|$$

Proof.

The theorem holds by homomorphism and monotonicity properties. For abstracting an interaction, demonic updates may be weakened and angelic updates strengthened. The formal proof is similar to that of Theorem 1.
□

5.2 Iterative Choice

The application of an *iterative choice* statement for specifying and refining interactive systems have been extensively discussed in [4]. This statement, introduced in [5], is defined as a recursive selection of possible interactions S.

$$\mathsf{do}\ \lozenge_i^n\ g_i\ ::\ S_i\ \mathsf{od}\ \triangleq\ (\mu X \cdot \{g_1\}; S_1; X \sqcup \ldots \sqcup \{g_n\}; S_n; X \sqcup \mathsf{skip})$$

The skip statement, models the user's choice of stopping the dialog with the system. μ denotes the least fix-point operator. In general, a recursive contract $\mu X \cdot S$ is interpreted as the contract statement S, but with each occurrence of statement variable X in S treated as a recursive invocation of the whole contract. Note that sequential composition binds stronger than the two choice operators.

The iterative choice statement follows a common iteration pattern, called angelic iteration. This iteration construct over S is defined as the following fix-point:

$$S^{\Phi}\ \triangleq\ (\mu X \cdot S; X \sqcup skip)$$

Therefore, we have

$$\mathsf{do}\ \lozenge_i^n\ g_i\ ::\ S_i\ \mathsf{od}\ = (\{g_1\}; S_1 \sqcup \ldots \sqcup \{g_n\}; S_n)^{\Phi}$$

Iterative choice should not be mixed with guarded command iterations used by Dijkstra [11]. Guarded command iterations are strong iterations defined by $S^{\omega} \triangleq (\mu X \cdot S; X \sqcap skip)$ with, in contrast to angelic iteration, the termination out of a user's control.

In [4] refinement rules for iterative choice are given. However, for testing we need abstraction rules for the synthesis of test-cases — scenarios are our goal.

5.3 Scenarios

An arbitrary scenario SC of an interactive system with n possible interactions S_i and of length l is a sequence of l sequential user interactions S_i. We write a sequence comprehension expression

$$\langle S_i(k) \mid (1 \leq i \leq n) \wedge (1 \leq k \leq l) \rangle$$

to denote such arbitrary sequences, where k is the position in the sequence[1].

Scenarios are abstractions of interactive systems, modeled by iterative choice, as the following theorem shows.

Theorem 7.

$$\mathsf{do}\ \Diamond_i^n\ g_i\ ::\ S_i\ \mathsf{od}\ \sqsupseteq \langle(\{g_i\}; S_i)(k) \mid (1 \leq i \leq n) \wedge (1 \leq k \leq l)\rangle$$

Proof. The theorem is valid by definition of the angelic iteration statement and thus by definition of iterative choice:

$$\begin{aligned} &\mathsf{do}\ \Diamond_i^n\ g_i\ ::\ S_i\ \mathsf{od} \\ &\quad \equiv\ \mathit{skip} \sqcup \{g_1\}; S_1 \sqcup \{g_2\}; S_2 \sqcup \{g_1\}; S_1; \{g_1\}; S_1 \sqcup \{g_1\}S_1; \{g_2\}S_2 \sqcup \ldots \end{aligned}$$

Hence, by definition of $\sqcup$ any choice of sequences of $\{g_i\}S_i$ is an abstraction. □

However, for test-case generation, we are only interested in valid scenarios. A scenario is considered a test-scenario if it terminates for every possible initial state. Thus its weakest precondition should be true for an unspecified final state:

$$\langle S_i(k) \mid (1 \leq i \leq n) \wedge (1 \leq k \leq l\rangle.\mathsf{true} = \mathsf{true}$$

Consequently, the abstraction should not equal the abort statement. Since abort is the bottom element $\perp$ of the predicate transformer lattice, it is the trivial abstraction of every statement. Therefore, we define a notion of testing abstraction $\sqsupseteq_T$

$$S \sqsupseteq_T T\ \triangleq\ S \sqsupseteq T \sqsupset \mathsf{abort}$$

and get the abstraction rule for testing scenarios:

Theorem 8. *Let $g(k)$ denote the guard at the kth position in a scenario and assume that the system specification is consistent. Hence we assume that for all*

[1] It should be mentioned that this sequence comprehension expression is not a valid predicate transformer, but rather serves as a scheme for sequences of predicate transformers. We use sequence comprehensions as a convenient notation, but they cannot be defined in higher-order logic due to its strong type system.

interactions $S_i.\mathsf{true} \subseteq g_i$. *Furthermore,* $g(l+1) \neq \mathsf{false}$ *should be an arbitrary predicate called the* ***goal****. Then*

$$\mathsf{do}\ \lozenge_i^n\ g_i\ ::\ S_i\ \mathsf{od}\ \sqsupseteq_T$$
$$\langle(\{g_i\};S_i)(k) \mid (1 \leq i \leq n) \wedge (1 \leq k \leq l) \wedge g_i(k) \subseteq S_i(k).g(k+1) \wedge g(1) = \mathsf{true}\rangle$$

Proof. Abstraction follows from Theorem 7. Termination is valid by induction: The weakest precondition of the first interaction is true, due to the assumption that for all interactions $S_i.\mathsf{true} \subseteq g_i$ and $g(1)$ chosen to be true. Consequently $S_i(1)$ terminates. An interaction $S_i(k+1)$ terminates due to the fact that its pre-condition $g(k+1)$ can be reached by definition.
□

This abstraction rule defines the calculation of valid test scenarios. The **goal** predicate is a condition that defines the states that should be reached by a sequence of interactions. Trivially, it can be chosen to be true. For developing a scenario for setting a system to a certain state, this goal predicate represents the corresponding state description.

The theorem above shows that the question if a scenario terminates, can be reduced to the question if two following interactions are composeable. From this observation a new testing strategy will be derived in the next section.

6 Test Strategies

A strategy for selecting test-cases is based on a hypothesis about faults in a program. This section shows that such strategies can be formulated as formal synthesis rules for deriving test-cases. This is a consequence of our observation that correct test-cases are abstractions of contracts, as has been explained in the previous section. The abstraction rules of the prominent strategies of partition testing and mutation testing are presented. In addition, it is shown that structural testing strategies are covered by this approach.

Furthermore, a new technique for calculating sequences of test-cases, here called scenarios, is presented. In contrast to previous work on test sequencing, our approach does not need to compute a finite state machine. This is in contrast to previous work on testing from state-based specifications.

6.1 Partition Testing

Partition testing techniques are based on a uniformity hypothesis. This strategy assumes that a system shows the same behavior for a certain partition of input values. Therefore, once the equivalence partitions are selected, it is sufficient to test one case for every partition.

For partitioning model-based specifications, [10] proposed the rewriting of specifications into their disjunctive normal form (DNF). This popular technique is based on rewriting disjunctions with:

$$a \vee b \equiv (a \wedge \neg b) \vee (\neg a \wedge b) \vee (a \wedge b)$$

The proof of the following abstraction rule for this well-known strategy highlighted a problem with this technique. In general, DNF-rewriting results in disjoint partitions. Applied to a relational specification it gives disjoint input-output relations, but the input partitions (the domain of the relation) may overlap.

In such pathological cases, selecting a test-case for each partition is not correct. If test-cases are derived for overlapping domains and the specification is non-deterministic then two test-cases with the same input may define two different deterministic outputs.

Therefore, the disjointness of the resulting input partitions $\mathsf{dom}.Q_i$ is assumed in the following synthesis rule.

Theorem 9. *Let a and b be arbitrary Boolean expressions and Q_1, Q_2, Q_3 be relations. The following rule for deriving partition test-cases from pre-post-condition contracts is then generally valid:*

$$\frac{\begin{array}{c}[x := x' | a \wedge \neg b] \sqsupseteq [Q_1], \\ [x := x' | \neg a \wedge b] \sqsupseteq [Q_2], \\ [x := x' | a \wedge b] \quad \sqsupseteq [Q_3], \\ \forall i, j \in \{1,2,3\} \,\textbf{.}\, i \neq j \Rightarrow \mathsf{dom}.Q_i \cap \mathsf{dom}.Q_j = \emptyset\end{array}}{\begin{array}{l}[x := x' | a \vee b] \sqsupseteq \{\mathsf{dom}.Q_1\}; [Q_1] \sqcup \\ \qquad\qquad\qquad\quad \{\mathsf{dom}.Q_2\}; [Q_2] \sqcup \\ \qquad\qquad\qquad\quad \{\mathsf{dom}.Q_3\}; [Q_3]\end{array}}$$

Proof. See Appendix.

The derivation rule of Theorem 9 yields three disjoint partitions, although a derived partition might be empty, thus $\mathsf{dom}.Q_i = \mathsf{false}$. It follows from Theorem 5 that each of these partitions is an abstraction of the original contract. It is obvious that the rule has to be applied recursively to the resulting partitions if further sub-partitions are needed.

The proof of Theorem 9 shows that the derivation rule yields partition test-cases that cover the full input domain. Furthermore, it follows from the assumptions that for one partition more non-determinism can be introduced due to abstraction. This reflects the possibility of weakening certain tests.

The rule shows that test-cases can be further abstracted by weakening a demonic update **(weakening of tests)**. In the rule above, for example the premise $[x := x'|a \wedge \neg b] \sqsupseteq [Q_1]$ indicates such a possible weakening of a test-case. This form of weakening a test-case is a further strategy for test-case design. The reason for this further abstraction of a partition might be that

– a tester does not wish to observe the (demonic) changes of the whole state space, or

– not the whole state and thus its updates are observable.

Therefore, parts of the demonic updates of the system are skipped in the test-case specifications. Formally this is an abstraction process carried out by weakening a post-condition.

Example 1. The rule is illustrated by deriving the test-cases from a specification of the computation of the minimum of two numbers.

$$[z := z'|(z' = x \wedge x \leq y) \vee (z' = y \wedge x \geq y)]$$

$=$ by the partitioning rule

$$\begin{array}{l}\{x < y\}; [z := z'|(z' = x \wedge x \leq y) \wedge (z' \neq y \vee x < y)] \sqcup \\ \{x > y\}; [z := z'|(z' \neq x \vee x > y) \wedge (z' = y \wedge x \geq y)] \sqcup \\ \{x = y\}; [z := z'|(z' = x \wedge x \leq y) \wedge (z' = y \wedge x \geq y)]\end{array}$$

$=$ by simplification

$$\begin{array}{l}\{x < y\}; [z := z'|z' = x] \sqcup \\ \{x > y\}; [z := z'|z' = y] \sqcup \\ \{x = y\}; [z := z'|z' = x \wedge z' = y]\end{array}$$

In the third test partition $\{x = y\}$ a further abstraction by weakening the relational update might be applied. For testing the correct computation for input $x = y$, a tester might-choose to compare the new value z' only with one input variable. If x is compared to z, then the following three test cases are obtained:

$\sqsupseteq$ by weakening the post-condition in the third test-case

$$\begin{array}{l}\{x < y\}; [z := z'|z' = x] \sqcup \\ \{x > y\}; [z := z'|z' = y] \sqcup \\ \{x = y\}; [z := z'|z' = x]\end{array}$$

□

Note that it has been the search for a valid abstraction that highlighted the necessary condition of disjointness. The last step in the proof shows that only if the demonic choice $\sqcap$ is in fact deterministic it can be transformed into an angelic choice $\sqcup$ of partitions. For a general demonic choice of partitions, the abstraction relation would not hold. The consequence could be the derivation of incorrect test-cases. The second example shows such a pathological case, where the rule must not be applied.

Example 2. The example demonstrates the consequence, if the rule is applied to non-disjoint partitions.

$$[z := z' | z' = 0 \lor z' = 1]$$

$\sqsupseteq$ by a wrong application of the partitioning rule

$$\begin{array}{l}\{\mathsf{true}\}; [z := z' | z' = 0 \land z' \neq 1] \sqcup \\ \{\mathsf{true}\}; [z := z' | z' \neq 0 \land z' = 1] \sqcup \\ \{\mathsf{false}\}; [z := z' | z' = 0 \land z' = 1]\end{array}$$

$=$ by simplification

$$\{\mathsf{true}\}; [z := z' | z' = 0] \sqcup \{\mathsf{true}\}; [z := z' | z' = 1]$$

$\sqsupseteq$ by selecting input-output test-cases

$$(\mathsf{TC}\ 1\ 0) \sqcup (\mathsf{TC}\ 1\ 1)$$

The result of this wrong application of the rule is the derivation of two deterministic test-cases for a non-deterministic result. If applied in testing an implementation of this specification, these test-cases indicate an error in the implementation that does not exist. The correct solution, for non-disjoint domains is to merge the partitions. Then, only non-deterministic test-cases would have been possible, like $\mathsf{TCp}\ 1\ (z' = 0 \lor z' = 1)$.
□

6.2 Structural Partitioning

Specification and implementation oriented testing strategies can be combined as has been shown in [6]. The same can be done in our refinement calculus approach. Here, the syntax of the specification language is considered for partitioning. This structural technique provides a better control of the partitioning process, then pure DNF-partitioning.

Our abstraction approach can be extended for defining test-case synthesis rules for different kinds of specification statements. For example, the rule for partitioning a conditional specification can be defined as follows:

Theorem 10. *Given a conditional contract statement as defined in Section 3.2. C_1 and C_2 be arbitrary contracts and P a predicate. Then, a conditional contract can be partitioned by*

$$\frac{C_1 \sqsupseteq C_1',\quad C_2 \sqsupseteq C_2'}{\mathsf{if}\ P\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi} \sqsupseteq \{P\}; C_1' \ \sqcup\ \{\neg P\}; C_2'}$$

Proof.
The rule is valid by the definition of the if-statement and the monotonicity of the $\sqsupseteq$ relation.
□

Synthesis rules, like the one for if-statements, can be given for arbitrary specification statements. Each such syntax oriented synthesis rule reflects a test-selection strategy, known from white-box testing. For example, the rule above represents branch testing, a strategy, where every branch in the control-flow should be tested once.

The following example demonstrates that a structural strategy might help to reduce the set of test-cases.

Example 3. The rule is illustrated by deriving the test-cases from an alternative specification of the computation of the minimum of two numbers.

$$\mathsf{if}\ x \le y\ \mathsf{then}\ [z := z' | z' = x]\ \mathsf{else}\ [z := z' | z' = y]$$

$=$ by the partitioning rule for if-statements

$$\{x \le y\}; [z := z' | z' = x] \ \sqcup\ \{x > y\}; [z := z' | z' = y]$$

In contrast to the three test-cases of Example 1, here only two test-cases are derived for the same problem.
□

6.3 Mutation Testing

Mutation testing is a fault-based testing technique introduced by Hamlet [12] and DeMillo et al [9]. 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. The hypothesis is that programmers only make small errors. Stocks extends this technique in [21] to model-based specification languages by defining a collection of mutation operators for Z's specification language. An example for specification mutation 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.

We can formulate abstraction rules for deriving mutation test-cases for a given contract and its mutants.

Theorem 11. *Given a contract C and a mutation C^m generated by applying a mutation operator m such that $m(C) = C^m$ holds. Then a test-case $\mathsf{TC}\ i\ o$ that covers the mutation must be derived by the rule*

$$\frac{\begin{array}{l} C \sqsupseteq \mathsf{TC}\ i\ o, \\ C^m \not\sqsupseteq \mathsf{TC}\ i\ o \end{array}}{C \sqsupseteq \mathsf{TC}\ i\ o}$$

The rule 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 a an abstraction of the mutated contract. 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 T for a contract C in mutation testing with a collection of mutation operators M can be given by:

$$\forall\, m \in M \centerdot \exists\, tc \in \mathsf{T} \centerdot (C \sqsupseteq tc \wedge m(C) \not\sqsupseteq tc)$$

This represents a new approach to the quality criteria for test-cases based on mutations. Again, the minimum example serves to illustrate the synthesis rule.

Example 4. This example shows how mutation testing techniques might be applied to specification based testing. Consider this specification C of a minimum computation:

$$\mathsf{if}\ x \le y\ \mathsf{then}\ [z := z' | z' = x]\ \mathsf{else}\ [z := z' | z' = y]$$

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

$$m(\ldots \le \ldots) = \ldots \ge \ldots$$

is defined that changes all occurrences of $\leq$ in a contract to $\geq$. Applying this mutant operator $m(C) = C^m$ computes the following mutant:

$$\mathsf{if}\ x \ge y\ \mathsf{then}\ [z := z' | z' = x]\ \mathsf{else}\ [z := z' | z' = y]$$

The synthesis rule says that a successful test-case for this mutation must not be an abstraction. Since

$$\mathsf{if}\ x \geq y\ \mathsf{then}\ [z := z' | z' = x]\ \mathsf{else}\ [z := z' | z' = y]\ \not\sqsupseteq\ \{x < y\}; [z := z' | z' = x]$$

but

$$\mathsf{if}\ x \leq y\ \mathsf{then}\ [z := z' | z' = x]\ \mathsf{else}\ [z := z' | z' = y]\ \sqsupseteq\ \{x < y\}; [z := z' | z' = x]$$

a test-case $\{x < y\}; [z := z' | z' = x]$ is a valid test-case for detecting a specification mutation of this kind in our implementation. On the other hand, the test-case $\{x = y\}; [z := z' | z' = x]$ would not detect the mutation since it is an abstraction of both, the original specification and the mutated one.
□

7 Calculating Scenarios for Testing

In this section an alternative strategy for the sequencing of test-cases into test-scenarios is proposed.

7.1 Critical Remarks on FSM approaches

In the literature on related work on test-sequencing for model-oriented specifications, authors have been concentrating solely on the approach proposed by Dick and Faivre in [10]. This strategy first calculates partitions of the available operations and states. Then a finite state machine (FSM) is constructed by calculating possible transitions between the states. The result is a graph with nodes that are state partitions and transitions that are operation partitions. To derive test-sequences (scenarios) the tester follows the paths in the graph. See the related work summarized in [1] for examples of this approach.

One disadvantage of this technique is that the whole FSM has to be calculated in advance, even if full coverage is out of the tester's scope due to resource limitations. This situation is even worse: Due to the focus on state partitions, the number of states increases exponentially with the number of partitioned state variables. Hence, rather large FSMs have to be calculated in advance. The second disadvantage is that a state based testing strategy is enforced, although the contract does not emphasize states but, like for interactive systems, possible interactions are the central paradigm of description.

In the following, a scenario oriented testing strategy is proposed. We call it a lazy technique, since the test-cases are calculated by need. It does not calculate a FSM, since it is not based on states. It is based on atomic scenarios, called compositions.

7.2 Compositions

We define a composition of an interactive system as a terminating sequential composition of two interactions:

$$Composition = C_1; C_2$$

The following corollaries follows directly from Theorem 8 and defines a rule for calculating such *compositions*.

Corollary 12. *For a consistent specification of interactions we have that*

$$(p \cap g_a) \subseteq S_a.g_b \Rightarrow \\ \mathsf{do}\ \Diamond_i^n\ g_i\ ::\ S_i\ \mathsf{od}\ \sqsupseteq_T \{p \cap g_a\}; S_a; \{g_b\}; S_b$$

where $1 \leq a, b \leq n$ *holds and* p *is an arbitrary predicate such that* $p \neq \mathsf{false}$.

In practice, we will not calculate the compositions from the original specification, but will previously perform a partition analysis on the interactions, leading to more (partition) interactions. However, the approach keeps the same. These compositions should be calculated for all interaction partitions of interest. Next, these compositions are combined into scenarios.

7.3 Scenario Synthesis

The following rule defines the general calculation of scenarios by combining two compositions of interest.

Corollary 13. *Let the interactions with indices* $1 \leq i, j, k \leq n$ *be interactions of an interactive system with* n *interaction partitions, and*

- *given two compositions*

$$\mathsf{do}\ \Diamond_i^n\ g_i\ ::\ S_i\ \mathsf{od}\ \sqsupseteq_T \{p_1 \cap g_i\}; S_i; \{g_j\}; S_j \sqcap \{p_2 \cap g_j\}; S_j; \{g_k\}; S_k$$

- *such that* $p \cap p_1 \cap g_i \subseteq S_i.(p_2)$
- *then the compositions can be combined to a new scenario*

$$\mathsf{do}\ \Diamond_i^n\ g_i\ ::\ S_i\ \mathsf{od}\ \sqsupseteq_T \{p \cap p_1 \cap g_i\}; S_i; \{g_j\}; S_j; \{g_k\}; S_k$$

In order to generate valid scenarios, a tester can e.g. start by an initial interaction with a guard equal to *true* and then he further searches for compositions leading to his test-goal. Which scenarios and how many scenarios are tested, depends on the testing strategy.

7.4 Scenario Based Testing Strategies

The new test approach can be divided into three phases:

1. calculation of interesting partitions for each interaction.
2. calculation of compositions.
3. combination of compositions to validate or to generate test-scenarios.

Different test-coverage strategies can be derived, determined by the strategy for combining the compositions. Interesting scenario analysis strategies are:

Derive scenarios that include for each partition

- one composition consisting of the partition: for each partition one scenario.
- all possible compositions consisting of the partition: for each partition, one scenario for each interaction reaching the partition.
- all possible combinations of compositions between two interactions of interest: all scenarios leading from one interaction of interest to another.
- all possible combinations of compositions: all possible scenarios.

The strategies are similar to the testing strategies used in data-flow testing [7]. The difference is that here atomic scenarios, called compositions, are considered, and in data-flow testing data-objects. For examples of scenario derivations we refer to [2] and [1].

8 Concluding Remarks

What we have presented, is to our knowledge, the first application of the refinement calculus for generating test-cases. In this paper several formal abstraction rules for calculating correct test-cases have been presented. These rules represent different strategies for test-case selection, known as DNF-partition testing, structural testing and mutation testing.

It is the author's opinion that an abstraction calculus for testing, provides deeper insight into the methods of test-case derivation. Especially, the understanding of new black-box testing strategies is supported by reasoning about the synthesis process. The examined testing strategies showed that even existing techniques reveal interesting properties. New test-case generation techniques may be the result.

For example, the presented synthesis rules for scenario calculation define an alternative method for finding sequences of interactions. In contrast to finite state machine (FSM) based approaches, the focus is on finding possible compositions of interactions. Which compositions are possible is determined by the abstraction rules.

We hope that the presented work stimulates further research on test-synthesis based on other program-synthesis approaches. Especially, the application of program synthesis and transformation tools for testing could be a promising topic of future research.

Acknowledgments

Since this work is part of my doctoral thesis [1], I wish to thank my *Doktorvater* Prof. Peter Lucas, for his help and patience during my research on this topic. It has been presented at the *Formal Aspects of Software Engineering* colloquium to mark his retirement from the chair in Software Technology at the Graz University of Technology, held on the 18th and 19th of May 2001 in Graz, Austria. As an organizer of this meeting, I wish to express my gratitude to all the participants who honored Peter through their presence. Thank you, all.

References

1. Bernhard Aichernig. *Systematic Black-Box Testing of Computer-Based Systems through Formal Abstraction Techniques.* PhD thesis, Institute for Software Technology, TU Graz, Austria, Jannuary 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, Andreas Gerstinger, and Robert Aster. Formal specification techniques as a catalyst in validation. In *Proceedings of the 5th Conference on High-Assurance Software Engineering, 15th–17th November 2000, Albuquerque, New Mexico, USA*. IEEE, 2000.
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, 1999.
5. Ralph-Johan Back and Joakim von Wright. *Refinement Calculus, a Systematic Introduction.* Graduate Texts in Computer Science. Springer, 1998.
6. Salimeh Behnia and Hélène Waeselynck. Test criteria definition for B models. 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 I*, volume 1709 of *Lecture Notes in Computer Science*, pages 509–529. Springer, 1999.
7. Boris Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New York, 2nd edition, 1990.
8. Simon Burton. Automated testing from Z specifications. Technical report, Department of Computer Science, University of York, November 2000.
9. 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.
10. 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.

11. E.W. Dijkstra. *A Discipline of Programming.* Series in Automatic Computation. Prentice-Hall International, 1976.
12. Richard G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.
13. Steffen Helke, Thomas Neustupny, and Thomas Santen. Automating test case generation from Z specifications with Isabelle. In *ZUM'97*, 1997.
14. 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.
15. Johann Hörl and Bernhard K. Aichernig. Formal specification of a voice communication system used in air traffic control, an industrial application of light-weight formal methods using VDM++ (abstract). In J.M. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of FM'99 – Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999*, volume 1709 of *Lecture Notes in Computer Science*, page 1868. Springer, 1999.
16. Johann Hörl and Bernhard K. Aichernig. Requirements validation of a voice communication system used in air traffic control, an industrial application of light-weight formal methods (abstract). In *Proceedings of the Fourth International Conference on Requirements Engineering (ICRE2000), June 19–23, 2000, Schaumburg, Illinois*, page 190. IEEE, 2000. Selected as one of three **best paper**s.
17. Johann Hörl and Bernhard K. Aichernig. Validating voice communication requirements using lightweight formal methods. *IEEE Software*, pages 21–27, May/June 2000.
18. Mathias Jarke and Reino Kurki-Suoni. Special issue on scenario management. *IEEE Transactions on Software Engineering*, 24(12), 1998.
19. Carrol C. Morgan. *Programming from Specifications.* Series in Computer Science. Prentice-Hall International, 1990.
20. William Perry. *Effective methods for software testing.* John Wiley & Sons, 1995.
21. Philip Alan Stocks. *Applying formal methods to software testing.* PhD thesis, The Department of computer science, The University of Queensland, 1993.

A Formal Proofs

Proof of Theorem 1

$\{p\}; [Q] \sqsupseteq \mathsf{TC}\ i\ o$
$\equiv$ by definitions
$\forall\ \sigma\ r \cdot p.\sigma \wedge Q.\sigma \subseteq r \Leftarrow (x = i).\sigma \wedge [y := y' | y' = o].r$
$\equiv$ by definition of demonic relational assignment
$\forall\ \sigma\ r \cdot p.\sigma \wedge Q.\sigma \subseteq r \Leftarrow (x = i).\sigma \wedge (\forall\ y' \cdot (y' = o) \Rightarrow r[y := y'])$
$\equiv$ by simplification of update
$\forall\ \sigma\ r \cdot p.\sigma \wedge Q.\sigma \subseteq r \Leftarrow (x = i).\sigma \wedge r[y := o]$
$\equiv$ by definition of substitution $r := (y := y' | y' = o).\sigma$
$\forall\ \sigma\ \cdot p.\sigma \wedge Q.\sigma \subseteq (y := y' | y' = o).\sigma \Leftarrow (x = i).\sigma$
$\equiv$ distributivity, subset definition
$(\forall\ \sigma\ \cdot (x = i).\sigma \Rightarrow p.\sigma)\ \wedge$
$(\forall\ \sigma\ \sigma'\ \cdot (x = i).\sigma \wedge Q.\sigma.\sigma' \Rightarrow (y := y' | y' = o).\sigma.\sigma')$
$\equiv$ definitions
$(x = i) \subseteq p \wedge |x = i|; Q \subseteq |y := o|$
$\Box$

Proof of Theorem 9

$[x := x' | a \vee b]$
$=$ by the DNF-partitioning rule
$[x := x' | (a \wedge \neg b) \vee (\neg a \wedge b) \vee (a \wedge b)]$
$=$ by definition of demonic choice
$[x := x' | (a \wedge \neg b)] \sqcap [x := x' | \neg a \wedge b] \sqcap [x := x' | a \wedge b]$
$\sqsupseteq$ by $\sqsupseteq$-assumptions and monotonicity of $\sqcap$
$[Q_1] \sqcap [Q_2] \sqcap [Q_3]$
$=$ by explicitly stating the domains as assumption
$[\mathsf{dom}.Q_1]; [Q_1] \sqcap [\mathsf{dom}.Q_2]; [Q_2] \sqcap [\mathsf{dom}.Q_3]; [Q_3]$
$=$ by the fact that the choice is deterministic, since:
(1) the domains are disjoint by assumption,
(2) the whole input domain is covered,
since abstraction cannot reduce the domain of a relational update.
$\{\mathsf{dom}.Q_1\}; [Q_1] \sqcup \{\mathsf{dom}.Q_2\}; [Q_2] \sqcup \{\mathsf{dom}.Q_3\}; [Q_3]$
$\Box$