Test-Case Calculation through Abstraction

Bernhard K. Aichernig

Institute for Software Technology, Graz University of Technology
Inffeldgasse 16b, A-8010 Graz, Austria
aichernig@ist.tu-graz.ac.at

Abstract: This paper discusses the calculation of test-cases for interactive systems. A novel approach is presented that treats the problem of test-case synthesis as an abstraction problem. The refinement calculus is used to formulate abstraction rules for calculating correct test-case scenarios from a formal contract. This abstraction calculus results in a synthesis method that, does not need to compute a finite state machine. This is in contrast to previous work on testing from state-based specifications. A well known example from the testing literature serves to demonstrate this unusual application of the refinement calculus in order to synthesize tests rather than implementations.

Keywords: testing, test-case synthesis, refinement calculus, abstraction rules, scenarios, contract.

1 Introduction

In the past of computer science a large gap between the testing and the formal methods community could be realized. Testers did not believe in the applicability of formal verification techniques to real world problems, and formal method's advocates could not accept testing-techniques as an adequate verification method for producing correct software. However, today the gap is closing.

Today, light-weight approaches to formal methods invite engineers to gain the advantages of formal specification techniques without focusing solely on correctness proofs. Having precise and unambiguous formal specifications available, is the pre-requisite in order to automate black-box testing. This functional testing approach, in which the system under test is considered as a black-box, has become more and more important: There is a growing awareness that a combination of black- and the more traditional white-box testing uncovers more defects than applying a technique solely. Especially, the object-oriented paradigm and the increasing use of Components Of The Shelf (COTS), shifted the focus of interest towards black-box approaches [8].

The process of testing, and so its automation, can be divided into two main activities: first, the synthesis of test-cases, second, the execution and evaluation of the tests. In [3] the author has presented techniques for automating the latter by using an abstract requirements specification in the Vienna Development Method Specification Language (VDM-SL) as a test-oracle for black-box testing. In this paper we focus on the synthesis of test-cases.

1.1 Motivation

The motivation for this work origins in two previous industrial projects, where explicit VDM specifications and IFAD VDMTools [13] have been used to support the conventional synthesis of system-level test-cases. Both projects in the area of voice communication for air-traffic control demonstrated the need for formality.

In the first project the formalization of the requirements uncovered 64 ambiguous, contradictory, unclear or erroneous issues in the informal system documents. Furthermore, the execution of the formal prototype with the system test-cases in use lead to the realization that only 80% of the system's radio functionality had been covered — an unacceptable low percentage for a safety-critical system [17, 18].

The second project's formalization raised 108 questions concerning the requirements, with 33 of them resulting in changes in the requirements document. Furthermore, 16 errors in the 65 original test-cases have been found [4]. This time, the conventional test-cases were designed in a very thorough manner: for each requirement one or more test-cases have been specified resulting in a 100% expression coverage. Thus, we realized that this coverage metric of VDMTools was too weak for finding more advanced test-cases.

The following observations motivate the formal test-synthesis method presented here. (1) The quality of system-level test-cases heavily relies on the quality of the based requirements. Formal specification techniques have proved to raise this quality. (2) Executable specifications tend to become too low-level with respect to abstraction. Therefore, a test-case synthesis approach should rather be based on more general relational than on functional specifications. (3) In our case, the safety-critical system is a highly interactive systems. Hence, the formalism used should be capable of capturing interaction. (4) Typical test-cases of such complex systems are scenarios of usage. Thus, a testing strategy must rather focus on scenarios than on input-output behavior. (5) In practice, many change requests have to be considered and test-cases should be easily adaptable. (5) Existing test-synthesis approaches do not satisfy our needs, as will be explained in more detail below.

1.2 Related Work on Testing

Since our approach is based on the refinement calculus, test-case generation from model-based specifications mostly relate to our own work. Model-based specifications use mathematical objects like sets, sequences and finite mappings for modeling a system's state explicitly.

One of the most cited works on test-case generation from model-based specifications is [11]. Dick and Faivre describe the method for an automated calculation of an finite state machine (FSM) based on a partitioning of the state spaces as well as the involved operations. A prolog tool has been developed for calculating the partitions based on a disjunctive normal form (DNF) transformation of the specification.

Most of the later work on testing from formal specifications is based on their observations. In Stocks' PhD thesis and his subsequent work, this formal partitioning technique is applied to Z [30, 29, 28, 9] for the first time.

Stepney realized the abstraction relation between test-cases and object-oriented Z specifications [27]. Her group developed a tool for interactively calculating partition abstractions and to structure them for reuse. Our work can be seen as an extension of this view on testing.

In [16] a Z-based tool for partitioning is presented. As in our previous work, he uses the specification as a test-oracle. [15] demonstrates that a theorem prover like Isabelle can be used to generate such test-classes based on DNF rewriting. In order to reduce the number of possible partitions the classification tree method is applied for selecting only interesting partitions in [26]. Furthermore, in [25] testing from combined Z and statechart specifications is discussed.

In [2] the DNF method is applied to B [1]. Furthermore, B prototypes have been used for preparing tests [31], similar to our own industrial projects using VDM. Recently, Derrick and Boiten [10] discussed the refinement of test-cases calculated from a Z specification. Again, the approach is based on calculating a FSM.

Another class of approaches starts directly from behavioral specifications like finite state machines or finite labeled transition systems. For example [23, 24] reports on the test-case generation from CSP specifications for embedded systems.

As we do, MacColl and Carrington are planing to develop a testing framework for interactive systems. They use a combination of Z and behavioral specifications (the process algebras CSP and CCS) for specifying such systems [21]. In [14] such behavioral specifications are combined with algebraic testing techniques.

As can be seen, a lot of work has been done on generating test-cases from formal specifications. However, this paper demonstrates that their is still space for new contributions.

1.3 Contribution

Unlike the work above, our approach is based on program synthesis techniques. Since test-case generation is a synthesis process we find it more natural to use a development method like the refinement calculus in order to calculate test-cases.

The innovation in the work presented here is that the synthesis of test-sequences (scenarios) is considered as an abstraction problem. Especially, the reverse application of a refinement calculus for calculating the test-cases is new. In general, refinement is known as the process of concretization from an abstract specification towards an implementation while preserving its correctness. In contrast to the usual application of refinement techniques (see [12, 20, 22, 6]), where a program is developed, in this approach the possible user interactions are specified on an abstract level and then further abstracted towards a valid sequence of user-actions – a test-scenario. Here Back's theory of refinement [6] is applied in order to calculate correct abstractions.

Finally, our sequencing technique does not calculate the complete FSM prior to test-case selection and thus differs from the approaches above. In an environment, where change requests are common, calculating the whole FSM is not very efficient. Our method focus on the impact on the possible scenarios by systematically analyzing interaction compositions.

2 Testing Based on Contracts

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 mathematical languages to define such a contract unambiguously and sound. In this work the formal contract language of [6] is used. It is a generalization of the conventional pre- and post-condition style of formal specifications of VDM, B and Z[1]. Higher-order logic (HOL) is its foundation.

A system is modeled by a global state x of type Σ denoted by $x : \Sigma$. Functionality is either expressed by functional state transformers $\langle f \rangle$ 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 a Boolean value (T or F). 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 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, 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 non-deterministic updates from the angel's point of view. Usually, we take the viewpoint of the angel.

Two contracts can be combined by sequential composition $C_1; C_2$ or choice operators. The angelic choice operator $C_1 \sqcup C_2$ and the demonic choice operator $C_1 \sqcap C_2$ define deterministic and non-deterministic choice 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 are possible for expressing iteration.

[1] In Z the pre-conditions are not explicitly stated, but calculated.

2.2 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 preconditions. We write $\Sigma \mapsto \Gamma \hat{=} (\Gamma \to \mathsf{Bool}) \to (\Sigma \to \mathsf{Bool})$ to denote a set of all predicate transformers from Σ to Γ. 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 $f(x)$. For details of the predicate transformer semantics, we refer to [6].

2.3 Refinement

The notion of contracts includes specification statements as well as programming statements. The latter 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 ensures 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

$$C \sqsubseteq C' \hat{=} \forall q \in \Gamma \cdot C.q \subseteq C'.q$$

. This ordering relation defines a lattice of predicate transformers (contracts) with the lattice operators meet $\sqcap$ and join $\sqcup$. The top element $\top$ is $\mathsf{magic}.q \hat{=} \mathsf{T}$, a statement that is not implementable since it can magically establish every postcondition. The bottom element $\bot$ of the lattice is $\mathsf{abort}.q \hat{=} \mathsf{F}$ 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 [6, 22]

3 Test-Cases are Abstractions

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

$$C \sqsupseteq C' \hat{=} C' \sqsubseteq C$$

This ordering relation of abstraction $\sqsupseteq$ defines the dual lattice on predicate transformers. Consequently, dual laws about the predicate transformer lattice can be constructed by interchanging $\sqsubseteq$ and $\sqsupseteq$, $\top$ and $\bot$, $\sqcup$ and $\sqcap$ in the original law.

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, here, we do not consider parameterized procedures, but only global state manipulations. In [6] 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 data. We can define such an input-output test-case TC as a contract between the user and the unit under test:

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

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:

$$\text{TCs} \mathrel{\hat{=}} \mathsf{TC}\ i_1\ o_1 \sqcup \ldots \sqcup \mathsf{TC}\ i_n\ o_n$$

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 $|p|$ and $|\langle f \rangle|$ denote the coercion of predicates and state transformers to relations. Furthermore the composition operator ; is overloaded for relations.

Proof.

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

$\equiv$ 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

$(x = i) \subseteq p \wedge |x = i|; Q \subseteq |y := o|$

$\Box$

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.

Corollary 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, 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 $o : \Gamma$. $\Box$

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. They are abstract, and thus easy to understand. Furthermore, 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 2.

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

where $1 <= k <= n$*.*

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

3.2 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. This 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 as already mentioned in the discussion of the related work in Section 1. 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

$$p_1 \cap \ldots \cap p_n = \emptyset$$

This 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 2. *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 $\sqcup$. $\square$

Above, 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 [6]). 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.

4 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 not sufficient 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 an interactive system. For a comprehensive introduction into the different roles of scenarios in software engineering see [19]. In this work, the focus is on validation and verification.

4.1 User Interaction

Testing interactive systems, typically involves the selection of a series of parameters. Some of this 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.

$$\mathrm{IA} \mathrel{\hat{=}} \{T\};[Q]$$

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 [6] 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 $\mathsf{TCI}\ i\ o$ involves the actual setting of the input variable to i.

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

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

Theorem 3. *Let $T : \Sigma \to \Gamma \to \mathsf{Bool}$ and $Q : \Gamma \to \Theta \to \mathsf{Bool}$ 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. □

Here, $T;Q$ denotes relation composition. The proof is similar to that of Theorem 1.

4.2 Iterative Choice

The application of an *iterative choice* statement for specifying and refining interactive systems have been extensively discussed in [5]. This statement, introduced in [6], is defined as a recursive selection of possible interactions S.

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

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.

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

$$S^{\Phi} \mathrel{\hat{=}} (\mu X \cdot S; X \sqcup skip)$$

Therefore, we have

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

Iterative choice should not be mixed with guarded command iterations used by Dijkstra [12]. Guarded command iterations are strong iterations defined by $S^{\omega} \hat{=} (\mu X \cdot S; X \sqcap skip)$ with, in contrast to angelic iteration, the termination out of a user's control.

In [5] refinement rules for iterative choice are given. However, for testing we need abstraction rules for the synthesis of test-cases — scenarios are our goal.

4.3 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.

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

Theorem 4.

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

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

$$\text{do } \lozenge_i^n \; g_i \; :: \; S_i \text{ od}$$
$$\equiv \; skip \sqcup \{g_1\}; S_1 \sqcup \{g_i\}; S_1 \sqcup \{g_1\}; S_1; \{g_1\}; S_1 \sqcup \{g_1\} S_1; \{g_2\} S_2 \sqcup \ldots$$

Hence, by definition of $\sqcup$ any arbitrary choice of sequence is an abstraction. $\square$

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

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

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 $\sqsupseteq_T$

$$S \sqsupseteq_T T \mathrel{\hat{=}} S \sqsupseteq T \sqsupset \text{abort}$$

and get the abstraction rule for testing scenarios:

Theorem 5. *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 interactions $S_i.true \subseteq g_i$. Furthermore, $g(l+1) \neq false$ should be an arbitrary predicate called the **goal**.*

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

Proof. Abstraction follows from Theorem 4. Termination is valid by induction: The weakest precondition of the first interaction is true, due to the assumption that for all interactions $S_i.true \subseteq g_i$ 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. $\square$

This abstraction rule defines the calculation of valid test scenarios. The goal predicate is a condition towards a sequence should be developed. Trivially, it is 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 compositionable. From this observation a new testing strategy will be derived in the next section.

5 Calculating Scenarios for Testing

5.1 Critics on FSM approaches

In previous related work on test-sequencing for model-oriented specifications, authors have been concentrated solely on the approach proposed by Dick and Faivre in [11]. This strategy first calculates partitions of the available operations and states. Then a finite state machine (FSM) is calculated by searching transitions from one state to the other. In this graph, nodes are state partitions and transitions are operation partitions. To derive test-sequences (scenarios) the tester follows the paths in the resulting graph. See the related work summarized in Section 1 for examples of this approach.

One disadvantage of this approach 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.

5.2 Compositions

We define a composition of an interactive system as a terminating sequential composition of two interactions. The following corollary follows directly from Theorem 5 and defines a rule for calculating such *compositions*.

Corollary 3. *For a consistent specification of interactions we have that*

$$(p \cap g_i) \subseteq S_i.g_j \Rightarrow \mathsf{do}\ \lozenge_i^n\ g_i\ ::\ S_i\ \mathsf{od}\ \sqsupseteq_T \{p \cap g_i\}; S_i; \{g_j\}; S_j$$

where p is an arbitrary predicate such that $p \neq$ 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. This compositions should be calculated for all interaction partitions of interest. Next, this compositions are combined into scenarios.

5.3 Scenario Synthesis

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

Corollary 4. *Let the interactions with indices $1 \leq i, j, k \leq n$ be interactions of an interactive system with n interaction partitions, then*

$$\begin{aligned}\mathsf{do}\ \lozenge_i^n\ g_i\ ::\ S_i\ \mathsf{od}\ \sqsupseteq_T \{p_1 \cap g_i\}; S_i; \{g_j\}; S_j \sqcap \{p_2 \cap g_j\}; S_j; \{g_k\}; S_k \wedge \\ p \cap p_1 \cap g_1 \subseteq S_j.(p_2 \cap g_2) \Rightarrow \\ \mathsf{do}\ \lozenge_i^n\ g_i\ ::\ S_i\ \mathsf{od}\ \sqsupseteq_T \{g_i\}; S_i; \{g_j\}; S_j; \{g_k\}; S_k\end{aligned}$$

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.

5.4 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 compositions to valid 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. An example will serve to demonstrate the approach at work.

6 Example: Process Scheduler

In this section, the application of our test synthesis method is demonstrated by an example, well known in the formal methods testing literature. It is the process scheduler introduced in [11] and translated to Back's contract notation. We have chosen this example, although it is not new and not of the type of system we envisage our method to apply to industrially. However, it allows a comparison to the traditional FSM approach most easily. Test-cases for industrial examples from our projects have been calculated, too. These will be published in future publications.

$$Init \mathrel{\hat{=}} [a, r, w := a', r', w' | a' = \mathsf{nil} \wedge r' \cup w' = \emptyset]$$

$$New \mathrel{\hat{=}} \{p := p' | p' \neq a \wedge p' \notin (r \cup w)\}; [w := w' | w' = w \cup p_{set}]$$

$$\begin{aligned} Ready \mathrel{\hat{=}} \{p := p' | p' \in w\}; [a, r, w := a', r', w' | w' = w - p_{set} \wedge \\ a = \mathsf{nil} \Rightarrow (r' = r \wedge a' = p) \wedge \\ a \neq \mathsf{nil} \Rightarrow (r' = r \cup p_{set} \wedge a' = a)] \end{aligned}$$

$$\begin{aligned} Swap \mathrel{\hat{=}} \{a \neq \mathsf{nil}\}; [a, r, w := a', r', w' | r = \emptyset \Rightarrow (a' = \mathsf{nil} \wedge r' = \emptyset) \wedge \\ r \neq \emptyset \Rightarrow (a' \in ready \wedge r' = r - a_set') \wedge \\ w' = w \cup a_{set}] \end{aligned}$$

Fig. 1. Contracts of the process scheduler's initialization and interactions.

6.1 Interactive Process Scheduler

The system consists of processes either *ready* to be scheduled or *waiting* to become ready and, optionally, a single *active* process. These processes are identified by a unique $\mathsf{Pid} \mathrel{\hat{=}} \mathsf{Nat}$. A process cannot be both ready and waiting, and the active process is neither ready nor waiting. In addition, there must be an active process whenever there are processes ready to be scheduled. The scheduling algorithm is not further specified.

We can model the interactions with this process scheduler as shown in Fig. 1. In this specification $a : \mathsf{Pid} \mid \mathsf{nil}$ is a global variable representing an optional active process, the global variable $r : \mathsf{set\ of\ Pid}$ and $r : \mathsf{set\ of\ Pid}$ represent the sets of ready and waiting processes. Furthermore, $p : \mathsf{Pid}$ is a global input variable solely used for setting a parameter.

New introduces another process, *Ready* puts a process into the ready state, and *Swap* changes the active process. *Swap* is a good example, how we separate preconditions on the internal state from conditions for the parameter selection. Here, the fact that *Swap* is only defined if $\{a \neq \mathsf{nil}\}$ is documented as a precondition.

The interactive process scheduler can be defined by iterative choice of these interactions. The initialization statement should be executed once prior to user interaction. In Figure 2 this model is shown. Note that the precondition of *Swap* has become a guard. Furthermore, necessary conditions such that a parameter

$$\begin{aligned} Scheduler \mathrel{\hat{=}} Init; \mathsf{do}\ & \mathsf{true} :: New \\ \lozenge\ & w \neq \emptyset :: Ready \\ \lozenge\ & a \neq \mathsf{nil} :: Swap \\ \mathsf{od}\ & \end{aligned}$$

Fig. 2. Contract of the interactive process scheduler.

$$Ready_1 \mathrel{\hat{=}} \{a = \mathsf{nil}\}; \{p := p' | p' \in w\}; [a, r, w := a', r', w' | w' = w - p_{set} \wedge \\ r' = r \wedge a' = p\ ]$$

$$Ready_2 \mathrel{\hat{=}} \{a \neq \mathsf{nil}\}; \{p := p' | p' \in w\}; [a, r, w := a', r', w' | w' = w - p_{set} \wedge \\ r' = r \cup p_{set} \wedge a' = a]$$

$$Swap_1 \mathrel{\hat{=}} \{a \neq \mathsf{nil} \wedge r = \emptyset\}; [a, r, w := a', r', w' | (a' = \mathsf{nil} \wedge r' = \emptyset) \wedge \\ w' = w \cup a_{set}]$$

$$Swap_2 \mathrel{\hat{=}} \{a \neq \mathsf{nil} \wedge r \neq \emptyset\}; [a, r, w := a', r', w' | (a' \in ready \wedge r' = r - a'_{set}) \wedge \\ w' = w \cup a_{set}]$$

Fig. 3. Partitioned process scheduler.

selection is possible are documented in the guards. Here $w \neq \emptyset$ is such a precondition of *Ready*.

6.2 Interaction Partitioning

For test-case synthesis, we first partition the basic interactions. Here, our partition strategy will be based solely on the case distinctions in the contract. As a consequence, the interaction *New* is not partitioned. Figure 3 presents the new partitions.

Further partitioning based on interesting states would be possible. Here, for example, *New* may be further partitioned into cases where $w = \mathsf{nil}$ and $w \neq nil$. Any partition is possible and can be formulated as a rewriting rule, such that the resulting partitions are correct abstractions as stated in Theorem 2.

As a consequence of this partitioning, a new interactive system contract can be given. In this new description shown in Fig. 4, the partition preconditions are incorporated into the guards. This is necessary such that our scenario synthesis approach works.

$$\begin{array}{lll} Scheduler' \mathrel{\hat{=}} Init; & \mathsf{do} & \mathsf{true} :: New \\ & \Diamond & w \neq \emptyset \wedge a = \mathsf{nil} :: Ready_1 \\ & \Diamond & w \neq \emptyset \wedge a \neq \mathsf{nil} :: Ready_2 \\ & \Diamond & a \neq \mathsf{nil} \wedge r = \emptyset :: Swap_1 \\ & \Diamond & a \neq \mathsf{nil} \wedge r \neq \emptyset :: Swap_2 \\ & \mathsf{od} & \end{array}$$

Fig. 4. Partitioned contract of the interactive process scheduler.

$$
\begin{array}{rr}
\{a = \mathsf{nil} \ \wedge g\}; New; Ready_1 & \{a \neq \mathsf{nil} \ \wedge g\}; New; Ready_2 \\
Swap_1; Ready_1 & \{\text{card } w > 1 \wedge g\}; Ready_1; Ready_2 \\
 & \{\text{card } w > 1 \wedge g\}; Ready_2; Ready_2 \\
 & Swap_2; Ready_2 \\
 & \\
\{r = \emptyset \wedge g\}; Ready_1; Swap_1 & \{\text{card } r > 1 \wedge g\}; Swap_2; Swap_2 \\
\{r = \emptyset \wedge a \neq \mathsf{nil} \wedge g\}; New; Swap_1 & Ready_2; Swap_2 \\
\{\text{card } r = 1 \wedge g\} Swap_2; Swap_1 & \{a \neq \mathsf{nil} \wedge g\}; New; Swap_2 \\
 & \\
S; New &
\end{array}
$$

Fig. 5. Compositions of the process scheduler.

6.3 Compositions

The next step, is the calculation of atomic scenarios — the compositions. The calculation is done by applying the rule of Corollary 3 to the partitioned interactive system contract. In many cases a precondition g of a composition has to be strengthened. In the formula this means that $p \neq \mathsf{true}$. This fits perfectly into our approach, since precondition strengthening is in fact abstraction. However, such strengthening indicates paths that are more difficult to establish.

The compositions of the scheduler are listed in Fig. 5. In this presentation the guard assertions g are skipped. Only if a guard g has been strengthened by a precondition p the additional assertion is shown as $\{p \wedge g\}$.

This collection of possible compositions has several advantages: (1) Several scenarios can be calculated stepwise, without calculating the weakest precondition for the whole sequence of interactions again and again. (2) It carries the information which interactions are easily established and which are difficult to set up. For setting up a goal as quick as possible, choosing simple compositions will lead to shorter scenarios. On the other hand, strong preconditions indicate that this combinations are complicated to carry out. A tester should include such complex combinations.

The compositions are grouped by the second statement, which is more practically for searching scenarios backwards. Backwards scenario development is more useful if scenarios are used to reach a certain test goal, as will be seen next.

6.4 Scenarios

Applying the rule for composing two compositions, correct scenarios can be synthesized in a systematic way. In Fig. 6 one scenario for testing each partition is presented. For each scenario the additional precondition to be established is documented. $Scenario_3$ serves to discuss the synthesis process in more detail.

The actual scenario synthesis starts with the last statement. Here, this is $Ready_2$, the interaction to be tested. From Fig. 5 it can be seen that four com-

$$
\begin{array}{lll}
Scenario_1 \mathrel{\hat=} Init; & & \text{(Testing } New\text{)} \\
\quad\quad New & & \\
 & & \\
Scenario_2 \mathrel{\hat=} Init; & & \text{(Testing } Ready_1\text{)} \\
\quad\quad \{a = \mathsf{nil}\}; New; & & \\
\quad\quad Ready1 & & \\
 & & \\
Scenario_3 \mathrel{\hat=} Init; & & \text{(Testing } Ready_2\text{)} \\
\quad\quad \{a = \mathsf{nil}\}; New; & & \\
\quad\quad \{a = \mathsf{nil} \land \text{card } w > 0\}; New; & & \\
\quad\quad \{\text{card } w > 1 \land g\}; Ready_1; & & \\
\quad\quad Ready_2 & & \\
 & & \\
Scenario_4 \mathrel{\hat=} Scenario_2; & & \text{(Testing } Swap_1\text{)} \\
\quad\quad Swap_1 & & \\
 & & \\
Scenario_5 \mathrel{\hat=} Scenario_3; & & \text{(Testing } Swap_2\text{)} \\
\quad\quad Swap_2 & &
\end{array}
$$

Fig. 6. Testing scenarios for the process scheduler.

positions are available. $Ready_1$ is chosen because a scenario for $Ready_1$ is already available. However, the new precondition needs to choose New twice.

New is chosen, because it is the most promising: Here, New can follow each statement S, since it has the precondition true. It is a general strategy to choose interactions with weak guards.

$Scenario_4$ and $Scenario_5$ shows that scenarios can be reused for setting a system in a state of interest.

Based on the tabular of possible compositions, all scenarios according to one of the selection strategies that have been presented in the previous section can be calculated. Here, we applied the first strategy: One scenario for each partition.

It should be noted that for this simple strategy, not all compositions have to be calculated in advance. However, compositions carry the information which combinations are most easily achieved, those with the weakest additional precondition. Trivially, it equals true.

7 Conclusions

What we have presented, is to our present knowledge, the first application of the refinement calculus for generating test-cases. We formally defined our notion of test-cases for simple input-output tests, partition tests and extended this definition to test scenarios for interactive systems.

For all these classes of test-cases, we demonstrated that they are in fact formal abstractions. This realization lead to formal abstraction rules for calculating correct test-cases. The presented synthesis rules define an alternative method for finding scenarios 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. A well-know example has been translated into an interactive contract specification and served for illustrating purposes of the method.

As future work we intend to investigate the application of abstraction techniques further. We will apply the method to our industrial projects. However, for large industrial examples the method needs automation. Theorem provers and model-checkers could be used for interactively verifying the abstraction relations.

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 topic of future research.

References

1. Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, Trumpington Street, Cambridge CB2 1RP, Great Britain, 1996.
2. Lionel Van Aertryck. *Une méthode et un outil pour l'aide à la génération de jeux de tests de logiciels*. PhD thesis, Université de Rennes, January 1998.
3. Bernhard K. Aichernig. Automated black-box testing with abstract VDM oracles. In M. Felici, K. Kanoun, and A. Pasquini, editors, *Computer Safety, Reliability and Security: proceedings of the 18th International Conference, SAFECOMP'99, Toulouse, France, September 1999*, volume 1698 of *Lecture Notes in Computer Science*, pages 250–259. Springer, 1999.
4. 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, 5th–7th November 2000, Albuquerque, New Mexico, USA*. IEEE, 2000. To be published.
5. 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 *Lecturen Notes in Computer Science*. Springer, 1999.
6. Ralph-Johan Back and Joakim von Wright. *Refinement Calculus, a Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998.
7. Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 2nd edition, 1990.
8. Boris Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, Inc., 1995.
9. D. Carrington and P. Stocks. A tale of two paradigms: Formal methods and software testing. In *Proceedings of the 8th Z User Meeting*. Springer-Verlag, 1994.
10. John Derrick and Eerke Boiten. Testing refinements of state-based formal specifications. *Software Testing, Verification and Reliability*, 9:27–50, July 1999.
11. 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.

12. E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall International, 1976.
13. John Fitzgerald and Peter Gorm Larsen. *Modelling Sytems, Practical Tools and Techniques.* Cambridge University Press, 1998.
14. Marie-Claude Gaudel and Perry R. James. Testing algebraic data types and processes: A unifying theory. *Formal Aspects of Computing*, 10(5 & 6):436–451, 1998.
15. Steffen Helke, Thomas Neustupny, and Thomas Santen. Automating test case generation from Z specifications with Isabelle. In *ZUM'97*, 1997.
16. E.M. Hörcher and E. Mikk. Test automation using Z specifications. In *"Tools for System Development and verification", Workshop, Bremen, Germany*, BISS Monographs. Shaker Verlag, 1996.
17. Johann Hörl and Bernhard K. Aichernig. Formal specification of a voice communication system used in air traffic control, an industrial application of lightweight 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. Full report at ftp://ftp.ist.tu-graz.ac.at/pub/publications/IST-TEC-99-03.ps.gz.
18. Johann Hörl and Bernhard K. Aichernig. Validating voice communication requirements using lightweight formal methods. *IEEE Software*, pages 21–27, May/June 2000.
19. Mathias Jarke and Reino Kurki-Suoni (editors). Special issue on scenario management. *IEEE Transactions on Software Engineering*, 24(12), 1998.
20. Cliff B. Jones. *Systematic Software Development Using VDM.* Series in Computer Science. Prentice-Hall, second edition, 1990.
21. Ian MacColl and David Carrington. A model of specification-based testing of interactive systems (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 1862. Springer, 1999. Full report at http://www.csee.uq.edu.au/ ianm/model.ps.gz.
22. Carrol C. Morgan. *Programming from Specifications.* Series in Computer Science. Prentice-Hall International, 1990.
23. Jan Peleska. Test Automation for Safety-Critical Systems: Industrial Application and Future Developments. In Marie-Claude Gaudel and Jim Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 39–59. Springer-Verlag, March 1996.
24. Jan Peleska and Michael Siegel. From Testing Theory to Test Driver Implementation. In Marie-Claude Gaudel and Jim Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 538–556. Springer-Verlag, March 1996.
25. S. Sadeghipour and H. Singh. Test strategies on the basis of extended finite state machines. Technical report, Daimler-Benz AG, Research and Technology, 1998.
26. Harbhajan Singh, Mirko Conrad, and Sadegh Sadeghipour. Test case design based on Z and the classification-tree method. In *Proceedings of the 1st International Conference on Formal Engineering Methods (ICFEM '97)*, 1997.
27. Susan Stepney. Testing as abstraction. In J. P. Bowen and M. G. Hinchey, editors, *ZUM '95: 9th International Conference of Z Users, Limerick 1995*, volume 967 of *Lecture Notes in Computer Science*. Springer, 1995.

28. P.A. Stocks and D.A. Carrington. Test templates: A specification-based testing framework. In *Proceedings of the 15th International Conference on Software Engineering*, pages 405–414, August 1993.
29. Phil Stocks and David Carrington. Test template framework: A specification-based testing case study. In *Proceeddings of the International Symposium on Software Testing and Analysis*, pages 11–18, 1993.
30. Philip Alan Stocks. *Applying formal methods to software testing*. PhD thesis, The Department of computer science, The University of Queensland, 1993.
31. H. Treharne, J. Draper, and S. Schneider. Test case preparation using a prototype. In Didier Bert, editor, *B'98*, volume 1393 of *LNCS*, pages 293–311. Springer, 1998.