

# Test-Case Generation as a Refinement Problem

**Bernhard K. Aichernig**  
Institute for Software Technology  
Technical University Graz  
Münzgrabenstr. 11,  
A-8010 Graz, Austria  
+43 316 873 5723  
aichernig@ist.tu-graz.ac.at

## ABSTRACT

This paper presents the idea to treat the problem of test-sequencing as a refinement problem. Given a formal interface specification of a system or component, the problem to generate a proper sequence of test-cases is formulated as a contract between the tester and the system under test. This contract followed by a test-goal assertion is then refined into a valid test-sequence using the refinement calculus. A simple example serves to demonstrate this unusual application of the refinement calculus in order to synthesize tests rather than implementations. This work is a first step towards the automated generation of correct test-cases based on a refinement calculator.

## Keywords

Black-box testing, test-sequencing, refinement calculus, formal methods

## 1 INTRODUCTION

During the last years the interest in formal description techniques has been growing in the “testing community”. The main reason is that formal specifications are the key-technology 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. Furthermore, the object-oriented paradigm and the increasing use of Components Of The Shelf (COTS), shifted the focus of interest towards black-box approaches [4].

The process of testing, and so its automation, can be divided into two main activities: first, the generation of test-cases, second, the execution and evaluation of the tests. The author has shown in [1] how an abstract

requirements specification in the Vienna Development Method Specification Language (VDM-SL) can be used as a test-oracle for black-box testing. In an industrial project summarized in [8] an explicit formal specification has been used to assess existing system test-cases of a safety-critical voice communication system for air traffic control. In this paper the focus is on the test-case synthesis phase.

Test-case synthesis can be further divided into domain partitioning and test-sequencing. Domain partitioning is the process of finding equivalence classes of the input domain. Test-sequencing is the synthesis of an appropriate sequence of tests in order to set a system into a desired internal state. This is necessary, if an operation is to be tested in a certain state that cannot be set directly.

The innovation in the work presented here is that the test-sequencing process is considered as a refinement problem. In general, refinement is known as the process of concretization from an abstract functional specification towards an implementation while preserving its correctness. In contrast to the usual application of refinement techniques (see [6, 9, 10, 3]), where a program is developed, in this approach the possible user interactions are specified on an abstract level and then refined towards a valid sequence of user-actions – the test-sequence. Hence, in this case, the refined implementation is the algorithm how user-actions are composed in order to reach a certain system’s state.

In particular, we demonstrate how the refinement calculus may form a base for a systematic test-case derivation. The formalism of Back and von Wright [3] is used throughout this paper. It is briefly introduced in the following Section 2. In Section 3 we show how an interactive system can be specified. Next, goal-oriented testing is discussed in Section 4 and formulated as a contract between the tester and the system under test. Section 5 presents rules necessary to calculate test-sequences from our formulation. Furthermore, a simple example serves to illustrate the application of the calculus. Finally, some conclusions are drawn in Section 6.

## 2 REFINEMENT CALCULUS

In this section a brief introduction into the formalism of the refinement calculus is given. For a systematic introduction into the refinement calculus and its foundations the reader is referred to [3]. *Simply typed higher-order logic* is used as the logical framework in this paper. The type of functions from a type  $\Sigma$  to a type  $\Gamma$  is denoted by  $\Sigma \rightarrow \Gamma$ . We write  $f.x$  for the application of function  $f$  to argument  $x$ . This section is taken from [2].

### States, Predicates and Relations

A computation can generally be seen as involving a number of agents (programs, modules, systems, users, etc.) who carry out actions according to a document (specification, program) that has been laid out in advance. When reasoning about a computation, we can view this document as a contract between the agents involved.

We assume that the world that contracts talk about is described as a state  $\sigma$ . The state space  $\Sigma$  is the set (type) of all possible states. The state has a number of program variables  $x_1, \dots, x_n$ , each of which can be observed and changed independently of the others. An assignment like  $x := x + 1$  denotes a state changing function that updates the value of  $x$  to the value of the expression  $x + 1$ .

A state predicate  $p : \Sigma \rightarrow \text{Bool}$  is a boolean function on the state. Since a predicate corresponds to a set of states, we use set notation ( $\cap$ ,  $\cup$ , etc.) for predicates. Similarly, a state relation  $R : \Sigma \rightarrow \Sigma \rightarrow \text{Bool}$  relates a state  $\sigma$  to a state  $\sigma'$  whenever  $R.\sigma.\sigma'$  holds.

### Contracts

Contracts are built from state changing functions, predicates and relations. The update  $\langle f \rangle$  changes the state according to  $f : \Sigma \rightarrow \Sigma$ . If the initial state is  $\sigma_0$  then the agent must produce a final state  $f.\sigma_0$ .

The assertion  $\{p\}$  of a state predicate  $p$  is a requirement that the agent must satisfy in a given state. If the assertion does not hold, then the agent has breached the contract. The assumption  $[p]$  is dual to an assertion; if the condition  $p$  does not hold, then the agent is released from any obligation to carry out his part of the contract.

In the sequential action  $S_1; S_2$  the action  $S_1$  is carried out first, followed by  $S_2$ . A deterministic choice  $S_1 \sqcup S_2$  allows the agent to choose between carrying out  $S_1$  or  $S_2$ . In a non-deterministic choice  $S_1 \sqcap S_2$  another agent (a demon) is able to choose. Thus, our agent (the angel) has to be prepared to carry out both of the actions.

The relational update statement  $\{R\}$  is a contract statement that permits an agent to choose between all final states related by the state relation  $R$  to the initial state. A recursive contract statement  $\mu X \cdot S$  is interpreted as the contract statement  $S$ , but with each occurrence of statement variable  $X$  in  $S$  treated as a recursive invo-

cation of the whole contract  $(\mu X \cdot S)$ . If the recursion unfolds infinitely, then the agent has breached the contract.

## 3 SPECIFYING INTERFACES

In this section it is demonstrated how interface are specified by means of pre-conditions and post-conditions.

### Pre- and Post-Conditions

A standard technique in formal methods to specify functionality implicitly is to use pre- and post-conditions. “Implicitly” means that it is specified *what* an operation should do and not *how*. A pre-condition is predicate of the input which must hold before a operation is invoked. The post-condition relation characterizes the output by relating it to the input.

A contract defined by means of a pre-condition  $p$  and a post-condition  $Q$  is defined as follows:

$$\{p\};[Q]$$

Here, the pre-condition is an assertion about the input state of an operation and the post-condition an assumption over the output state calculated by the specified component (agent).

The interface of a system is characterized by its operations available. Thus an interface can be specified by a set of  $n$  operations using pre- and post-condition contracts:

$$\begin{aligned} Op_1 &= \{p_1\};[Q_1] \\ Op_2 &= \{p_2\};[Q_2] \\ &\vdots \\ Op_n &= \{p_n\};[Q_n] \end{aligned}$$

However, in order to formulate the testing of such an interface it is necessary to define user interaction.

### User Interaction

The deterministic choice of the user to activate an operation can be expressed by the following contract:

$$Action = Op_1 \sqcup Op_2 \sqcup \dots \sqcup Op_n$$

Taking into account that the user is interacting with a system by performing a series of actions before he decides to stop yields:

$$Interaction = \mu I \cdot Action; (I \sqcup \text{skip})$$

A recursive contract statement is used to specify the users interaction. After performing an action, the user can decide if he wants to stop and to do nothing (*skip*), or to carry on. Note, that interaction is defined by at least performing one action. In [2] an alternative formalism for modeling interactive systems is discussed.

## 4 TEST-GOALS

In general, goal-oriented testing (GOT) is an interaction of the tester resulting in a specified result — the goal.

In a contract such a goal might be a predicate  $g$  or a relation  $G$  over the state space  $\Sigma$ . The test-goal is a requirement that the contract must satisfy. Consequently, it is formulated as an assertion  $\{g\}$  or  $\{G\}$ , respectively. The contract of goal-oriented testing is defined by

$$GOTesting = Interaction; \{g\}$$

The problem of finding a sequence of actions in order to set the system's internal state such that an operation can be tested can be expressed by means of a test-goal. The test-goal  $g$  of an operation  $op_i$  is then its pre-condition  $p_i$ , or formally:

$$GOTesting-op_i = Interaction; \{p_i\}$$

Here, the test-sequencing problem is specified from the testers perspective. It is the tester's responsibility to choose the appropriate sequence of operations not breaching the contract. In finding a valid sequence of actions the tester assumes that the operations deliver results as specified in their post-condition. If this assumption is violated the tester has found a failure with respect to the specification, or in other words the system under test has breached the contract.

## 5 TEST-SEQUENCE CALCULATION

In this section the actual calculation of valid test-sequences towards a goal  $g$  is demonstrated. The novelty is that finding a sequence is considered as a refinement problem. Thus the refinement calculus is used to calculate a sequence of actions that is correct by construction.

### Manual vs. Automated Testing

In order to calculate a test-sequence by refinement it is necessary to modify our contract of goal-oriented testing: The contract in the previous section specifies a manual testing from the testers point of view. This has been expressed by a deterministic (angelic) choice of actions. Specifying a goal oriented test-generator it is necessary to leave the freedom to choose an appropriate action to the generator. Thus, for an automated generation of test-sequences, the choices of operations performed appear non-deterministically from the testers point of view. Indeed, the delivery of a non-deterministical sequence of actions is the purpose of a test-case generator. The new contract incorporates this demonic choice of actions:

$$TS-op_i = \mu I \cdot Op_1 \sqcap Op_2 \sqcap \dots \sqcap Op_n; (I \sqcap skip); \{p_i\}$$

It is also necessary to change the point of view concerning the interface operations. A generator, not the tester should choose the valid input of an operation  $op_i$

satisfying its precondition  $p_i$ . Consequently, the precondition is formulated as an assumption  $[p_i]$ . On the other hand it is the tester's responsibility to provide a correct implementation of the operation under test. Specifying the relational update (the post-condition) as an assertion contract, releases the generator from the responsibility if a failure in the operation under test occurs. Therefore, the post-condition becomes the contract statement  $\{Q_i\}$

From the previous it follows that an operation involved in test-case generation has to be specified by

$$[p_i]; \{Q_i\}$$

The interpretation of a demonic pre-post-condition specification is that if test-sequencer finds the correct input, then the post-condition assertion must hold. Before our testing-approach is demonstrated, the necessary theorems and concepts are introduced in the following.

### Refinement Properties

An important refinement law in our approach is the following:

$$Op_1 \sqcap Op_2 \sqcap \dots \sqcap Op_n \sqsubseteq Op_i$$

for  $1 \leq i \leq n$ . Any choice of a non-deterministic contract is a correct refinement. The same holds for recursive interaction contracts a presented above. Any sequence of actions is a refinement of a recursively defined contract of the same actions:

$$\begin{aligned} & (\mu I \cdot Op_1 \sqcap \dots \sqcap Op_n; (I \sqcap skip)) \\ \sqsubseteq & Op_i, \dots Op_j \end{aligned}$$

for  $1 \leq i, j \leq n$ . The rest of the refinement laws are shown as {comments} in the following example derivation of a valid test-sequence.

### Example

Consider, a simple component toggling a bit of information. The component's state space  $\Sigma$  consists of a single program variable  $x$ . The interface of this component consists of three operations.

$$\begin{aligned} Init &= [\text{true}]; \{x = 0\} \\ Set &= [x = 0]; \{x = 1\} \\ Unset &= [x = 1]; \{x = 0\} \end{aligned}$$

The initialization operation *Init* can be performed at any time, which is expressed by the pre-condition  $[\text{true}]$ . After initialization the  $x$  is set to value 0. The *Set* operation sets  $x$  to 1, the *Unset* operation changes the value of  $x$  back to 0. The following calculation shows a

valid derivation of the test-sequence in order to test the *Unset* operation.

$$\begin{aligned}
& TS\text{-}Unset \\
= & \\
& (\mu I \cdot Init \sqcap Set \sqcap Unset; (I \sqcap skip)); \{x = 1\} \\
\sqsubseteq & \{Init \text{ chosen}\} \\
& Init; (\mu I \cdot Init \sqcap Set \sqcap Unset; (I \sqcap skip)); \{x = 1\} \\
\sqsubseteq & \{skip \text{ chosen}\} \\
& Init; (Init \sqcap Set \sqcap Unset; skip); \{x = 1\} \\
= & \{skip \text{ is unit of composition}\} \\
& Init; (Init \sqcap Set \sqcap Unset); \{x = 1\} \\
= & \{\text{distributivity}\} \\
& Init; (Init; \{x = 1\} \sqcap Set; \{x = 1\} \sqcap Unset; \{x = 1\}) \\
= & \\
& Init; \\
& ([true]; \{x = 0\}; \{x = 1\} \\
& \sqcap [x = 0]; \{x = 1\}; \{x = 1\} \\
& \sqcap [x = 1]; \{x = 0\}; \{x = 1\}) \\
= & \{\{p\}; \{q\} = \{p \sqcap q\}\} \\
& Init; \\
& ([true]; false \\
& \sqcap [x = 0]; \{x = 1\} \\
& \sqcap [x = 1]; \{false\}) \\
\sqsubseteq & \\
& Init; [x = 0]; \{x = 1\} \\
= & \\
& Init; Set
\end{aligned}$$

## 6 CONCLUDING REMARKS

In this work we have explained and demonstrated the idea to apply the refinement calculus in order to derive test-sequences for a certain test-goal.

To our present knowledge no work has been published previously, where test-sequencing has been considered as a refinement problem.

In [5] an algorithm is given how test-sequences may be derived from pre- and post-condition specifications formally defined in the Vienna Development Method Specification Language (VDM-SL). In contrast to our work, the graph of the abstract state machine is manually calculated by partitioning both, the global state

and the operations, into equivalence classes or suboperations respectively.

For event-based systems, test-sequences are computed automatically from an explicit state-machine representation of the behaviour [7]. Starting from a pre-post-condition specification and using refinement techniques keeps a possibly infinite state-automaton implicitly.

The approach presented here targets complex systems involved in complex data-processing with possibly infinite state spaces, where model-checking based approaches [11, 12] to test-case generation are not applicable.

We hope that this paper stimulates a new research area, where refinement techniques and tools are investigated to be applied for test-case generation. This work is a first step towards the automated generation of correct test-cases based on a refinement calculator.

## ACKNOWLEDGEMENTS

Many thanks go to Ralph-Johan Back for pointing me in a fruitful discussion during the FM'99 symposium to the very useful Chapter 28 in his book: "Refinement in Context" [3].

## REFERENCES

- [1] B. 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.
- [2] R. Back, A. Mikhajlova, and J. von Wright. Reasoning about interactive systems. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 — Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, number 1709 in Lecture Notes in Computer Science*, pages 1460–1476. Springer, September 1999.
- [3] R.-J. Back and J. von Wright. *Refinement Calculus, a Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998.
- [4] B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, Inc., 1995.
- [5] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In J. Woodcock and P. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*. Springer-Verlag, April 1993.

- [6] E. Dijkstra. *A Discipline of Programming*. Prentice-Hall International, 1976.
- [7] G. Droschl. Design and application of a test case generator for VDM-SL (extended abstract). In J. Fitzgerald and P. G. Larsen, editors, *Workshop Materials: VDM in Practice!, Part of the FM'99 World Congress on Formal Methods, Toulouse, September 1999*.
- [8] J. Hörl and B. K. Aichernig. Formal specification of a voice communication system used in air traffic control, an industrial application of lightweight formal methods using VDM++. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 — Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France*, number 1709 in Lecture Notes in Computer Science. Springer, September 1999.
- [9] C. B. Jones. *Systematic Software Development Using VDM*. Series in Computer Science. Prentice-Hall, second edition, 1990.
- [10] C. C. Morgan. *Programming from Specifications*. Series in Computer Science. Prentice-Hall International, 1990.
- [11] J. Peleska. Test Automation for Safety-Critical Systems: Industrial Application and Future Developments. In M.-C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 39–59. Springer-Verlag, March 1996.
- [12] J. Peleska and M. Siegel. From Testing Theory to Test Driver Implementation. In M.-C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 538–556. Springer-Verlag, March 1996.