BRICS NS-01-4 Brinksma & Tretmans (eds.): FATES'01 Proceedings

Proceedings of the Workshop on

Formal Approaches to Testing of Software FATES'01

Aalborg, Denmark, August 25, 2001

Ed Brinksma
Jan Tretmans
(editors)

BRICS Notes Series NS-01-4

ISSN 0909-3206 August 2001

Ed Brinksma and Jan Tretmans (Eds.)

Formal Approaches to Testing of Software

FATES’01
A Satellite Workshop of CONCUR’01
Aalborg, Denmark, August 25, 2001
Proceedings

Preface

Testing is an important technique for validating and checking the correctness of software. However, effective and efficient testing turns out to be difficult, expensive, laborious, error-prone and time consuming. Formal methods are a way of specifying and verifying software systems by applying techniques from mathematics and logic. This enables the analysis of systems and the reasoning about them with mathematical precision and rigour. Formal methods and testing used to be a difficult combination. Formal methods aim at verifying and proving correctness, while testing can only show the presence of errors. Validation in practice is most often performed by testing, while academic research was concentrated on formal verification.
Recently, there is an increasing interest in the use of formal methods in software testing. It is recognized that formal methods and testing are complementary techniques which can, and should be used in combination. The use of formal methods can help in alleviating some of the challenges of software testing. In particular, a formal, and formally verified specification provides a more precise, more consistent and more complete starting point for the testing process and the obtained tests can be formally validated whether they test what should be tested. Moreover, the use of formal methods allows automating the generation of tests from formal specifications, thus leading to a faster, cheaper and less error-prone testing process. And finally, formal testing turns out to be a good starting point for introducing formal methods in software development.
The aim of the workshop FATES — *Formal Approaches to Testing of Software* — is to be a forum for researchers, developers and testers to present ideas about and discuss the use of formal methods in software testing. Topics of interest are formal test theory, test tools and applications of testing based on formal methods, including algorithmic generation of tests from formal specifications, test result analysis, test selection and coverage computation based on formal models, and all of this based on different formal methods, and applied in different application areas.

This volume contains the papers presented at FATES'01 which was held in Aalborg (Denmark) on August 25, 2001, as an affiliated workshop of CONCUR'01. Out of 18 submitted papers the programme committee selected 9 papers and 1 tool demonstration for presentation at the workshop. Together with the keynote presentation by David Lee from Bell Labs Research China they form the contents of these proceedings.
The papers present different approaches to using formal methods in software testing. The main theme is the generation of an efficient and effective set of test cases from a formal description. Different formalisms are used as the starting point, such as finite state machines, Z, Statecharts, SDL, constraint languages, grammars and timed

automata, and different algorithms are discussed for the generation process, ranging from formalization of the manual testing process to the (re)use of techniques from model checking.
The papers give insight in what has been achieved in the area of software testing with formal methods. Besides, they give clear indications of what has to be done before we can expect widespread use of formal techniques in software testing. The prospects for using formal methods to improve the quality and reduce the cost of software testing are good, but still more effort is needed, both in developing new theories and in making the existing methods and theories applicable, e.g., by providing tool support.

We would like to thank the programme committee and the additional reviewers for their support in selecting and composing the workshop programme, and we thank the authors for their contributions without which, of course, these proceedings would not exist.
Last, but not least, we thank Brian Nielsen and Arne Skou for arranging all local matters of organizing the workshop, Aalborg University for giving the opportunity to organize FATES'01 as a satellite of CONCUR'01, BRICS for supporting the printing and distribution of these proceedings, and Jan Feenstra en René de Vries for setting up the FATES'01 web page.

Enschede, August 2001

Ed Brinksma
Jan Tretmans

Programme committee

Ed Brinksma (University of Twente, The Netherlands), co-chair
Rocco De Nicola (Università degli Studi di Firenze, Italy)
Marie-Claude Gaudel (Université de Paris-Sud, France)
Jens Grabowski (Universität Lübeck, Germany)
Robert Hierons (Brunel University, United Kingdom)
Thierry Jéron (INRIA Rennes, France)
Doron Peled (Bell Laboratories, USA)
Alexandre Petrenko (CRIM, Canada)
Jan Tretmans (University of Twente, The Netherlands), co-chair
Antti Valmari (Tampere University of Technology, Finland)
Carsten Weise (Ericsson Eurolab Deutschland GmbH, Germany)

Referees

Timo Aaltonen	Ilkka Kokkarinen
Axel Belinfante	Mieke Massink
Machiel van der Bijl	Brian Nielsen
Sergiy Boroday	Vlad Rusu
Pedro R. D'Argenio	Ina Schieferdecker
Michael Ebner	Michael Schmitt
Jan Feenstra	Arne Skou
Stefan Heymer	René G. de Vries

Organization

Brian Nielsen,
Arne Skou,
Aalborg University,
Denmark

Contents

Efficient Algorithms for Test Sequence Selection

(Extended Abstract)

David Lee and Ruibing Hao
Bell Labs Research China, Beijing, China

Abstract: We study the test sequence selection problem. Given a large number of tests for a protocol system, we want to select a subset of tests so that the number of tests is reduced yet the test coverage is not sacrificed. We discuss the complexity of the test selection problem and propose a class of algorithms for different protocol system information requirements, test coverage criteria, and cost. This article is an extended abstract of [LH], and we refer the interested readers to the full paper for a detailed study and experimental results.

1. INTRODUCTION

With advanced computer technology protocol systems are getting larger to fulfil complicated tasks. However, they are also becoming less reliable. Testing is an indispensable part of system design and implementations; yet it has proved to be a formidable task for complex systems. Because of its practical importance and theoretical interest, there have been a lot of activities on protocol system testing. There are conformance testing, interoperability testing, and performance testing. Conformance testing is to test conformance of system implementations to their specifications [Br, LY1, PBG]. Interoperability testing tends to uncover faults when different system components are interoperating or interfacing with each other [GHLS, KK]. Conformance and interoperability testing are designed to check the correctness of system behaviors whereas performance testing is related to the system performance, such as its transmission rate.

For certain complicated legacy protocol systems, such as 5ESS (AT&T/Lucent No. 5 Electronic Switching System), tests have been generated and applied to the systems over the years at different development stages and by different test engineers. There are thousands of test sequences in the available test set. To test such systems, it is impractical to generate tests from scratch, since it is often impossible to model the systems because they are too complex. Therefore, we want to use the available test set accumulated over years. However, we do not want to execute all of them, since there are too many and to run each test takes a substantial amount of time. A natural solution in practice is to select test sequences among the available test set.

The test selection problem was studied in [LPB] based on the valuations of the test sequences to be selected, and it was reduced to an optimisation problem over Boolean algebra. For testing in context, the problem was studied in [YCL]. An important issue of test selection is the possible loss of coverage. This problem was investigated in [LPB] for partially specified machines and also in [CV, VC, ZV] with an elegant metric of coverage. The selection criteria and their notations were studied in [Pa].

In this paper, we study the following test selection problem. We have a large set of test sequences, and we want to select a minimal subset of tests to execute without sacrificing the fault coverage. For a formal study, we use extended finite state machine to model protocol systems. We discuss the complexity of the problem and propose efficient algorithms for the test sequence selection.

After describing an extended finite state machine model and its reachability graph, we formulate a problem of test sequence selection, discuss the coverage criteria, and study the problem complexity. We then discuss efficient algorithms for the test selection.

A finite state machine contains a finite number of states and produces outputs on state transitions after receiving inputs. It is often used to model control portions of protocol systems. However, data portions of protocols include variables and operations based on their values; ordinary finite state machines are not powerful enough to model in a succinct way the physical systems any more. Extended finite state machines (EFSM), which are finite state machines extended with variables, have emerged from the design and analysis of communication protocols [LY1]. For instance, IEEE 802.2 LLC [ANSI] is specified by 14 control states, a number of variables, and a set of transitions (pp. 75-117). For a formal definition of EFSM and the related concepts, see [LY1].

Each combination of a state and variable values is called a configuration. An EFSM usually has an initial state s_0 and all the variables have an initial value $\mathbf{x}_{init}$; we have the initial configuration $(s_0,\mathbf{x}_{init})$. A reachability graph consists of all the configurations and transitions, which are reachable from the initial configuration. It is a directed graph where the nodes and edges are the reachable configurations and transitions, respectively. Obviously, a control state may have multiple appearances in the nodes (along with different variable values) and each transition may appear many times as edges in the reachability graph.

For a path from the initial node (configuration) of the reachability graph, the input/output (I/O) labels on the transitions of the path provide an I/O sequence. Conversely, if an I/O sequence corresponds to a unique path from the initial node in the reachability graph, then the underlying EFSM is *deterministic*. Otherwise, it is *non-deterministic.* For clarity, in this paper we only consider systems, which are modelled by deterministic EFSM's. Our approaches can be modified to handle non-deterministic EFSM's, as we shall comment in the conclusion of the paper.

Given an input sequence, there is at most one path from the initial node in the reachability graph such that the transitions on the path are labelled with the input sequence, since the machine is deterministic. If such a path exists, then the given input sequence is *valid.* Otherwise, it is *invalid.* Obviously, a valid input sequence corresponds to a unique I/O sequence, which are the labels on the corresponding unique path from the initial node.

We first study the test selection problem, assuming that the underlying system reachability graph is available. We then relax the problem with an assumption that only the EFSM specification of the underlying system is available. Finally, we present algorithms for a general case when no information of the underlying system is available for test selection.

2. TEST SEQUENCE SELECTION PROBLEM AND ITS COMPLEXITY

A test sequence (or a scenario) is valid input sequence of a protocol system that is modelled by an EFSM. Since the system is deterministic, there is a one-to-one correspondence between test sequences and paths in the reachability graph from the initial configuration. In practice, a test sequence usually consists of an I/O sequence; the input sequence is applied to the system under test, and the output sequence is to be observed from the system response. Therefore, a test sequence can be represented by a valid input sequence, a valid I/O sequence, or a path from the initial configuration in the reachability graph of the underlying protocol system. For convenience, we shall use these terms interchangeably.

Informally, the test sequence selection problem is: Given a set of test sequences S, select from S a subset of tests S^* with a desirable fault coverage. Fault coverage is essential for testing. However, it is often specified differently from different system models and practical needs.

A commonly used criterion of fault coverage is to test each edge in the reachability graph at least once. It has been a well-accepted criterion in practice, and we first consider this criterion. Formally, given an EFSM **M**, let G be its reachability graph with an initial configuration $v_{0,}$ which corresponds to the initial state of **M** and the initial variable values. A test sequence of **M** is a path in G from v_0. A covering test set is a set of test sequences such that each edge of G is covered by at least one of the test sequences. We want to select a subset of tests such that it is still a covering test set and contains a minimal number of tests:

Problem 1. Test selection with a reachability graph. Given the reachability graph of an EFSM and a covering test set S, select a subset of test sequences S^* from S, such that S^* is a covering test set with a minimal cardinality.

The problem is NP-hard [LH]. Therefore, it is hard to obtain an optimal solution for test selection in general. We discuss heuristic methods next.

3. TEST SEQUENCE SELECTION

We first consider the case that the underlying system reachability graph is available, and then study the more general case without such reachability graphs.

3.1. Test Sequence Selection with a Reachability Graph

We consider the following greedy method. We first find a test sequence in S that covers a maximal number of uncovered edges of the reachability graph, and add this test to S^*. We then repeat the process until all the edges of the reachability graph are covered. Formally,

```
Algorithm 1.
input: reachability graph G and a covering test set S
output: a covering test set S*, which is a subset of S

1.  mark all edges in graph G as uncovered;
2.  S* = Λ;  /* an empty set */
3.  repeat
4.      find a test sequence s in S that covers a maximal number of
        uncovered edges of the graph G, and mark these edges as covered;
5.      S = S – {s}, S* = S* ∪ {s};
6.  until all edges in G are covered
7.  return S*
```

Figure 1. Algorithm 1

A routine analysis shows that Algorithm 1 takes time O(MN) to select a covering test set where M and N are the total length and number of all the test sequences in the given covering test set.

To reduce the cost we may want to avoid examining all the tests in the set. We can conduct the following preprocessing and variations of Algorithm 1.

We first sort the tests in S according to their lengths in a descending order, and then examine them in that order. The rationale is: longer tests correspond to longer paths in the reachability graph and may cover more edges of the graph. Furthermore, if we have found a test, which covers k uncovered edges, then there is no need to examine tests of length k or less. Since there are N tests in the set, the added cost of sorting is O(NlogN), which is negligible since it is dominated by O(MN).

Randomisation often helps. We can select each test sequence to be examined from S uniformly at random.

Another variation is that we sort the test sequences by their input symbols alphabetically rather than their lengths, remove all the tests which are a proper prefix of other tests, and then apply Algorithm 1. Obviously, this pre-processing reduces redundant tests and we have fewer tests to process. We shall further discuss this variation in the next section.

3.2. Test Sequence Selection without a Reachability Graph

Often we only have a test set available but do not have the corresponding reachability graph of the underlying system or it is too costly to construct such a graph. The problem becomes harder since we do not know whether a selected test set covers all the edges in the reachability graph.

Suppose we have the underlying system specification EFSM yet without its reachability graph. We have:

Problem 2. Test selection with an EFSM. Given an EFSM and a covering test set S, select a subset of test sequences S^* from S, such that S^* is a covering test set with a minimal cardinality.

Similar to Problem 1, the test selection problem 2 is NP-hard. Therefore, it is hard to obtain an optimal solution for test selection in general.

We can use the following heuristic procedure. We examine all the test sequences in S, trace each sequence in the EFSM, and record all the edges covered in the corresponding reachability graph. We then select test sequences in S until all the recorded edges are covered. The selection procedure is similar to Algorithm 1, and the variations of Algorithm 1 also apply here. Note that we do not construct the reachability graph explicitly.

```
Algorithm 2.
input: EFSM M and covering test set S
output: a covering test set S*, which is a subset of S

1.  $E = \Lambda$;  /* identify all edges of reachability graph; E is an empty edge set */
2.  for each test sequence $s_i$ in $S$, $i = 1, \dots, N$
3.      $u=u_0=< s_0\ x_{init} >$ ; /* $u_0$ is the initial configuration of M */
4.      for $j = 1, \dots, k_i$  /* $k_i$ is the number of transitions in $s_i$ */
5.          trace transition $t_{ij}$ in M and determine next configuration $v$;
6.          if corresponding edge in reachability graph $(u,v) \notin E$
7.              $E = E \cup \{(u,v)\}$;
8.          $u = v$;
9.  set all edges in set $E$ as uncovered; /* find a covering subset $S^*$ */
10. $S^* = \Lambda$;  /* an empty set */
11. repeat
12.     find a test sequence $s$ in $S$ that covers a maximal number of uncovered
        edges in set $E$, and mark these edges as covered;
13.     $S = S - \{s\}$, $S^* = S^* \cup \{s\}$;
14. until all edges in $E$ are covered
15. return $S^*$
```

Figure 2. Algorithm 2

Algorithm 2 takes time O(MNlogM) to select a covering test set where M and N are the total length and number of all the test sequences in the given covering test set.

Algorithm 2 is for a more general test selection Problem 2 where there is no reachability graph and also no need to construct one. However, there is an extra price to pay in terms of the run time, i.e., a factor of logM.

Often in practice the underlying system model may not be available, especially for those legacy systems, and we only have a set of tests to select. This is the most general case of the test selection problem:

Problem 3. Test selection without any information of the underlying protocol system. Given a covering test set S for a protocol system, for which there is no information available, select a subset of test sequences S^*, such that S^* is a covering test set.

Different from Problem 1 and 2, in general it is impossible to select a minimal covering test set since there is no information of the underlying protocol system.

For this problem we only consider the following redundancy criterion. Let s and s^* be two test sequences where s^* is a prefix of s. Let p and p^* be the corresponding paths from the initial configuration in the reachability graph, which we do not know. Since the corresponding EFSM is deterministic, p^* is a prefix of p as paths in the reachability graph, and we can remove s* from the test set without sacrificing the coverage. Consequently, if S is a covering test set, the test subset S^*, selected by discarding all the prefixes, is still a covering test set. As a matter of fact, in the worst case, we cannot reduce the tests anymore. Consider the following reachability graph. It consists of separate paths from the initial node. For test selection, we can only discard the paths (tests), which are a proper prefix of another path; any further reduction will lose the coverage.

The above observations lead to the following algorithm. Consider the alphabet set of all the input symbols. We sort all the input sequences in S alphabetically and then remove all the input sequences (test sequences), which is a prefix of another input sequence.

```
Algorithm 3.
input: test sequence set S and alphabet set of all the input symbols Σ
output: subset S* of S with redundant tests removed

1.  sort all input sequences of S alphabetically;
2.  S* = Λ;  /* an empty set */
3.  for i = 1, …, N-1   /* N is the cardinality of S */
4.      if $s_i$ is not a prefix of $s_{i+1}$
5.          $S^* = S^* \cup \{s_i\}$;
6.  $S^* = S^* \cup \{s_N\}$;
7.  return S*
```

Figure 3. Algorithm 3

Given a test set S, Algorithm 3 removes redundant tests and selects a subset of tests S^*. If S is a covering test set of the reachability graph of the underlying protocol system, then S^* remains a covering test set.

Note that Algorithm 3 may select test sequences that are redundant in terms of covering the reachability graph since we do not have its information. Therefore, the resulting number of tests may be larger than that from Algorithm 1 and 2.

It takes time O(MlogN) to select tests using Algorithm 3 where M and N are the total length and number of all the test sequences in the given test set.

The cost of the test selection of Algorithm 3 is dominated by sorting the test sequences. We now discuss another method, using trees. It is optimal in terms of run time; it takes time proportional to the total lengths of the test sequences to be selected.

We grow a tree as follows. Initially, we have a root node v_0. Each edge is labeled with an input symbol. For each input sequence (test), we walk down the tree as follows. At a node u of the tree with an input symbol a in the sequence, if there is an outgoing edge from u labeled with a, then we walk down along that edge, arrive at the end node v, and process the input symbol in the test sequence after a. However, if there are no outgoing edges labeled with a, then we add a tree edge (u,v) from u, label it with a, walk down along (u,v), and continue processing from v as before. Each test requires a tree walk. After all the tests have been processed, we can obtain the selected tests as follows. We only have to select the tests (or paths) from the root to a leaf node; all the other tests (paths) a prefix of these selected tests. Note that each step of the tree walk takes a constant time (if we have a bitmap at each node to keep track of the labels of all the outgoing edges). Therefore, it takes time proportional to the length of a test sequence to process. Hence the total cost is O(M).

It takes time O(M) to select tests using Algorithm 4 where M is the total length of all the test sequences in the given test set.

Obviously, Algorithm 4 removes the prefixes and selects the same test set as Algorithm 3. It is more efficient but involves an on-line tree construction process. Again, if the given test set is a covering set then the selected subset is also a covering. But it may contain redundant tests in terms of covering the reachability graph, of which we have no information.

Algorithm 4.
input: test sequence set S and alphabet set of all the input symbols Σ
output: subset S of S with redundant tests removed*

```
1.  $V = \{v_0\}$;   /* $V$ is node set of tree T; each node keeps track of all input
    symbols of its outgoing edges by a bitmap,  */
2.  for $i$ =1, …, $N$   /* N is the number of test sequences in $S$ */
3.      $u = v_0$; /* a tree walk from initial node $v_0$ */
4.          for $j$ = 1, …, $k_i$  /* $k_i$ is number of input symbols in sequence $s_i$ */
5.          let $t_{ij}$ be transition of $s_i$ under examination with input $t_{ij}$->$a$;
6.          if ($u$.bitmap($a$)=0)   /* not recorded yet */
7.              initialize a new node $v$;
8.              construct a new tree edge $(u, v)$;
9.              $u$.bitmap($a$) =1;  /* record this input symbol at $u$ */
10.             $u = v$; /* walk down tree to $v$ along newly constructed edge */
11.         else /* input has been recorded and tree edge exits for a walk */
12.             $u=v$;
13. $S^* = \Lambda$;  /* an empty set */
14. for each leaf node $v$ of tree $T$;
15.         let $s$ be test sequence corresponding to path from $v_0$ to $v$;
16.         $S^*=S^*\cup\{s\}$;
17. return $S^*$
```

Figure 4. Algorithm 4

4. CONCLUSION

In this paper, we have discussed the test sequence selection problem. Given a test sequence set, we want to select a subset of tests; without sacrificing the coverage we want to minimize the number of the selected tests to reduce the test execution time. We have proposed several algorithms with different required information and coverage, at different costs, and with different redundancy of the selected tests.

So far we assume that the underlying protocol system is deterministic, and in this case, a valid test sequence corresponds to a unique path in the reachability graph. If the system is non-deterministic, then a valid test sequence may correspond to multiple paths in the graph. Since the execution sequence is non-deterministic, the coverage of the edges in the reachability graph is probabilistic. While the union of all the possible paths, which are associated with a valid test sequence, is considered for coverage, we can only claim in a probabilistic sense.

ACKNOWLEDGEMENTS

Kun Du from Peking University implemented some of the algorithms. Our thanks go to the reviewers of PSTV-FORTE'2001 for their constructive comments and suggestions.

REFERENCES

[ANSI] International standard ISO 8802-2, ANSI/IEEE std 802.2, 1989.

[Br] E. Brinksma, *A Theory for the Derivation of Tests, Proc. PSTV*, pp. 63-74, 1988.

[BTV] E. Brinksma, J. Tretmans and L. Verhaard, *A Framework for Test Selection*, *Proc. PSTV*, pp. 233-248, 1991..

[CV] J. A. Curgus and S. T. Vuong, *A Metric Based Theory of Test Selection and Coverage, Proc. PSTV*, 1993.

[GJ] M. R. Garey and D. S. Johnson, *Computers and Intractability: a Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979.

[GHLS] N. Griffeth, R. Hao, D. Lee, and R. Sinha, *Integrated System Interoperability Testing with Applications to VoIP, Proc. FORTE/PSTV*, pp. 69-84, 2000.

[KK] S. Kang and M. Kim, *Interoperability Test Suite Derivation for Symmetric Communication Protocols, Proc. FORTE/PSTV*, 57-72, 1997.

[LH] D. Lee and R. Hao, *Test Sequence Selection*, *Proc. PSTV-FORTE*, 2001.

[LY1] D. Lee and M. Yannakakis, *Principles and Methods of Testing Finite State Machines - a Survey, The Proceedings of IEEE*, Vol. 84, No. 8, pp. 1089-1123, 1996.

[LY2] D. Lee and M. Yannakakis, *Optimization Problems from Feature Testing of Communication Protocols*, *Proc. ICNP*, pp. 66-75, 1996.

[LPB] G. Luo, A. Petrenko, and G. v. Bochmann, *Selecting Test Sequences for Partially Specified Nondeterministci Finite State Machines, Proc. IFIP 7th Int. Workshop on Protocol Test Systems*, 1994.

[Mo] E. F. Moore, *Gedanken-experiments on sequential machines, Automata Studies, Annals of Mathematics Studies*, Princeton University Press, no.34, pp.129-153, 1956.

[Pa] J. Pachl, *A Notation for Specifying Test Selection Criteria, Proc. PSTV*, pp. 71-84, 1990.

[PY] C. H. Papadimitriou, M. Yannakakis, *Optimization, Approximation and Complexity Classes, J. Comp. Sys. Sc.*, 43(3), 425-440, 1991.

[PBG] A. Petrenko, S. Boroday, R. Groz, *Confirming Configurations in EFSM, Proc. FORTE/PSTV,* 5-24, 1999.

[VC] S. T. Vuong and J. A. Curgus, *On Test Coverage Metrics for Communication Protocols, Proc. IWTCS*, 1991.

[YCL] N. Yevtushenko, A. Cavalli, L. Lima, *Test Suite Minimization for Testing in Context, Proc. IWTCS*, pp. 127-145, 1998.

[ZV] J. Zhu and S. T. Vuong, *Generalized Metric Based Test Selection and Coverage Measure for Communication Protocols*, *Proc. PSTV*, pp. 299-314, 1997.

Tool Demonstration Proposal: UML Validation Suite

Marc Lettrari[1] and Jochen Klose[2] and Udo Brockmeyer[3]

[1] OFFIS, Germany
[2] University of Oldenburg – Department of Computer Science
[3] OFFIS Systems and Consulting GmbH, Germany
e-mail:{marc.lettrari,jochen.klose}@informatik.uni-oldenburg.de
brockmeyer@o-s-c.de

1 Introduction

Distributed real-time computer systems are very complex and intrinsically difficult to specify and implement correctly. One cause is the lack of adequate methods and tools to deal with this complexity. The use of UML for developing such systems is gaining more and more attention both from research and industry.

A major goal in developing such systems is a validation that the design fulfills certain properties. Although there are some approaches with the goal of a formal proof of the correctness of a design [5, 4], testing still plays a dominant role. The reasons are the limited applicability of the methods which strive for a formal proof:

- Fully automatic methods (model checking) can only be applied to small designs and simple properties.
- Semi-automatic (with the help of theorem provers) or manual proofs are often very difficult and require significant knowledge and experience.

Conventional testing on the other hand is easier to do, but does not necessarily uncover all error situations. Therefore a fusion of the formal verification and testing approaches promises to improve the validation of software. Here we present a UVS (UML Validation Suite) tool for testing UML models designed with the Rhapsody tool of I-Logix, Inc. We add semantic rigor to the Rhapsody Sequence Diagrams (SDs) and use them to monitor or drive the interactive simulation of the UML model.

This proposal is organized as follows: We start with the formal foundation of Sequence Diagrams in section 2, followed by a quick overview of the features of the UVS tool and how it can be integrated into the design process in section 3. We conclude with information about the implementation of the tool in Sect. 4.

2 Sequence Diagrams

The Sequence Diagrams we are considering here are an enhanced version of the ones offered by the Rhapsody tool at the moment. In the context of monitoring and driving an UML design we need more expressiveness than that provided by the UML standard. For this reason we have added a feature which we feel is essential for this application area: the capability to specify when a scenario described in a SD should be activated, i.e. when should we start to monitor the system or generate certain inputs? The simplest possibility would be to only consider *initial* SDs, i.e. those which are activated at system start. This is clearly too restrictive, since we also want to be able to use scenarios which are active more than once. We therefore introduce the concept of *activation mode* which can be either *initial* or *iterative*, where the second choice indicates that the SD can be activated several times, whenever the activation condition is true during a run of the system and the SD is not already active. For the iterative case another feature is needed, which allows us to specify when exactly the SD should be activated. Here we use an *activation condition*, a Boolean expression which characterizes the state of the Rhapsody design when the scenario should start. Both the activation mode and condition we have adopted from *Live Sequence Charts* (LSCs) [1] which are the formal base of the UVS tool.

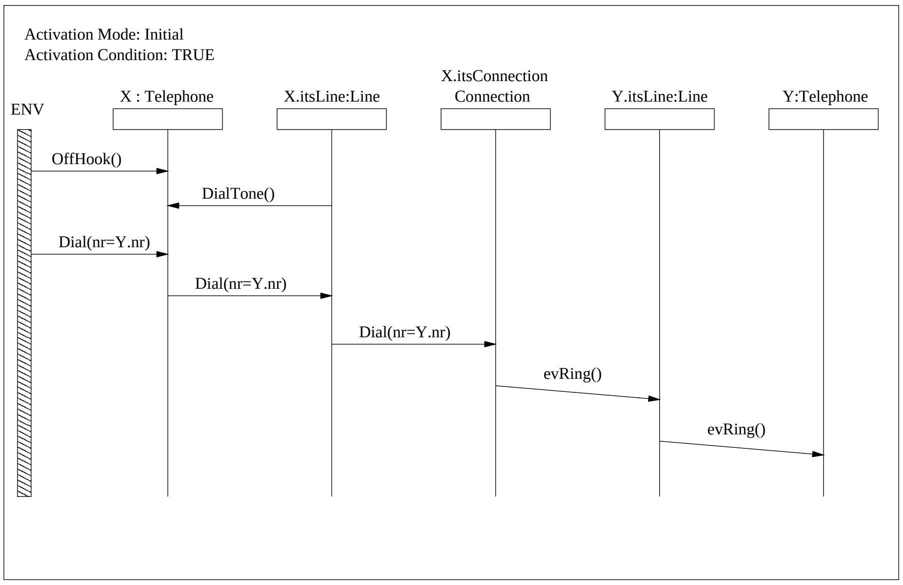


Fig. 1. Sequence diagram example SD1

Figure 1 shows an example SD, where a connection between two telephones is set up. The environment informs the telephone system that the receiver has been lifted off the hook, then the dial tone is heard in the receiver. Now the caller dials the telephone number of the callee which is propagated through the system. In the end the callee's phone rings.
The formal semantics for our SDs is given by an automaton which represents all the communication sequences allowed by the SD from which it is constructed. The algorithm for deriving this automaton from an SD is again taken from the LSC world (see [3] for a detailed description).

An error is indicated by the UVS tool when the order defined in the SD is violated by the simulation run, i.e. when an event or method call is observed at the wrong time. Consequently the only permissible order of the messages is the one shown in the SD. At this point also the activation of an SD comes into play, since an error can only be detected when the corresponding SD has been activated.

3 Methodology

The typical UML development process (e.g. Rapid Object-oriented Process for Embedded Systems (ROPES) [2]) is iterative, starting with an early, fairly abstract version and progressing to more and more concrete prototypes. The first version will often be incomplete having some classes which are already well developed and others which are just empty shells, i.e. the classes exist together with their (incomplete) set of events and methods which are not implemented yet. An appropriate validation technique has to support such a development process which is often not the case.

Therefore we aim at an approach which supports validation throughout the whole development process. We use Sequence Diagrams as a graphical language to describe certain functional and real-time properties. We believe that Sequence Diagrams are well suited for such a testing process because of several reasons, e.g

- SDs are used early in the development process to capture the relevant use cases of the desired system. Therefore the testing process can run in parallel with the development process which allows early detection of functional or timing errors.
- The graphical fashion of SDs eases the communication between users, developers and test engineers. Therefore the acceptance of the developed systems will increase.
- Most of the common real-time properties (e.g. response times) can be expressed using SDs.

We show the application of our testing approach at 3 typical stages in the development process where we incrementally create early prototypes (testing during design), a first fully implemented design (unit testing) and an enhanced design (regression testing).

- Suppose we have a system under construction where we want to test the already implemented classes. This is possible by designating the incomplete classes as *stubs* in the SD(s) and letting the test tool take over their behavior. The stubbed classes have to provide at least those interface objects (operations and event[1] receipts) which appear in the SD. The difference between monitoring and testing a SD is the way the environment is treated. If we want to monitor a SD, then we only observe if all described events and method calls occur in the right order and within the right time intervalls. If we want to test a SD, then the events coming from the environment (and the ones from stubbed instances) are generated automatically at the appropriate points in time, and all other messages are monitored.

[1] Signals in UML terminology.

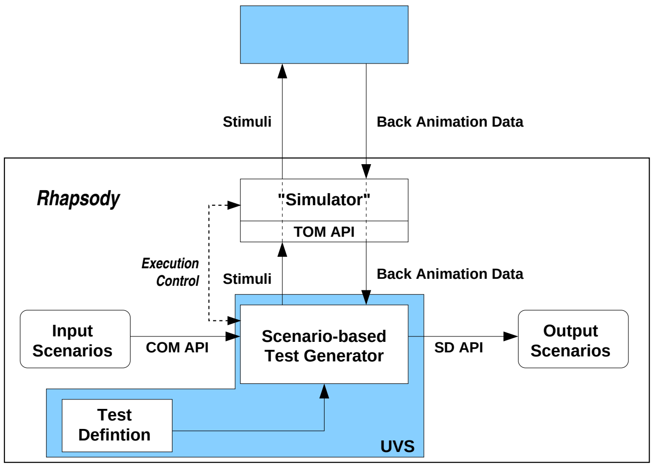


Fig. 2. Communication between UVS and Rhapsody

– In later iterations as the design becomes more and more complete the stubbed SD instances can be transformed into regular ones as the underlying classes become more complete. In general, SDs which have been specified in earlier iterations can be used for regression testing in the later phases, thereby becoming invariants of the model. Another important aspect of reusing SDs is the possibility to parameterize them. By using parameters such as "X" and "Y" as object names on SD instances all combinations of objects of the corresponding classes can be treated in one SD. If we want to use such an SD for monitoring or testing, we have to instantiate these parameters with concrete objects of the system.
– In each new iteration the existing SDs can be supplemented by new ones which specifically test those features which have been added in this iteration cycle. Of course, when inserting new functionality into a design, other functions should not be affected. To test this, we can use the SDs from the previous iterations for regression testing.

In our example we always used only one SDs at a time for testing, but our approach also allows complex tests consisting of several SDs whose activation can be controlled either implicitly by their activation mode and activation condition or explicitly by an ordering given by the user. Each test can consist of arbitrary many instances of SDs, which are SDs with instantiated parameters. This allows the definition of very complex tests where messages can be sent in parallel or sequentially to the application, whereas many monitors can observe the behaviour of the application.

4 Implementation

At present, the UVS tool is being integrated into Rhapsody and it will be part of the next release. In this section we briefly outline the software architecture of the UVS and its interaction with Rhapsody.

A prerequisite of our notion of testing is the possibility to interact with the system under test. For our approach it is important that the tester is informed about every event and method call which is described in the considered SDs. Furthermore the tester must have the possibility to send events to the system under test. These capabilities can be realized in different ways (e.g. code instrumentation, model executor etc.). Figure 2 shows how UVS and Rhapsody interact with each other. The UVS manages the definition of the test cases and controls the test execution sending stimuli when required and observing the actions taken in the simulator.

We have integrated the whole test environment directly in Rhapsody. For each project the user can define arbitrarily many tests which are automatically stored together with the project files. For each test it can be specified which SDs it should contain, how the parameters should be instantiated and which SDs should run in parallel or sequentially. During the execution of the test the user is informed about which parts of the test behave as expected and which not. If an error was detected, a SD is generated automatically to visualize the error.

Rhapsody generates instrumented code, which communicates with a simulation interface (TOM API) during runtime. Whenever relevant things occur with regard to the considered SDs, the simulation interface informs the testing tool about them. Based on these notifications the testing tool detects possible errors or generates events automatically which can be send back to the implementation through the simulation interface.

References

1. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. In *FMOODS'99 IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems*, 1999.
2. Bruce P. Douglass. *Doing Hard Time.* Addison-Wesley, 1999.
3. Jochen Klose and Hartmut Wittke. An Automata Based Representation of Live Sequence Charts. In Tiziana Margaria and Wang Yi, editors, *Proceedings of TACAS 2001*, number 2031 in LNCS. Springer Verlag, 2001.
4. Diego Latella, Istvan Maijzik, and Mieke Massink. Towards a formal operational semantics of uml statechart diagrams. In *3rd International Conference on Formal Methods for Open Object-Oriented Distributed Systems*, Lecture Notes in Computer Science. Kluwer Academic Publishers, 1999.
5. Johan Lilius and Ivan Porres Paltor. Formalising uml state machines for model checking. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language: Beyond the Standard*, number 1723 in Lecture Notes in Computer Science. Springer-Verlag, 1999.

Automatic Test Generation from Statecharts Using Model Checking

Hyoung Seok Hong, Insup Lee, Oleg Sokolsky
Department of Computer and Information Science
University of Pennsylvania
{hshong,lee,sokolsky}@saul.cis.upenn.edu
Sung Deok Cha
Division of Computer Science and AITrc
Department of Electrical Engineering and Computer Science
Korea Advanced Institute of Science and Technology
cha@salmosa.kaist.ac.kr

Abstract

This paper discusses the application of model checking to test generation from specifications written in statecharts. We consider a family of coverage criteria based on the flow of control and data in statecharts and formulate the problem of test generation as finding counterexamples during the model checking of statecharts. The ability of model checkers to construct counterexamples allows test generation to be automatic.

To illustrate our approach, we are based on the temporal logic CTL and its symbolic model checker SMV. We describe how to translate statecharts to inputs to SMV after defining the semantics of statecharts in terms of Kripke structures. We, then, describe how to express various coverage criteria in CTL and show how SMV can be used to generate only executable tests.

1 Introduction

This paper addresses the problem of test generation from statecharts [12] that have been widely used for specifying reactive systems. Statecharts can be regarded as extended finite state machines (EFSM) that support the hierarchical and concurrent structure on states and the communication mechanism through event broadcasting. Among several variants of statecharts considered in the literature [2], we concentrate on the STATEMATE semantics for statecharts [13]. Our approach, however, can be immediately applied to other variants of statecharts semantics, for example, the UML statecharts [24].

A statechart specification typically allows an infinite number of executions and hence exhaustive testing is impossible, which requires all the possible executions be performed. The prevalent testing practice is to construct a test suite, that is, a finite set of test sequences according to certain coverage criteria. For test coverage, we adapt the notions of control flow and data flow coverage used traditionally in software and protocol testing to statecharts. For test generation, we present an approach that involves the application of the temporal logic CTL [8] and its symbolic model checker SMV [21] to statecharts.

An overview of our approach is shown in Figure 1. The problem of test generation is formulated as a CTL model checking problem. A given coverage criterion is expressed as a parameterized collection of formulas in CTL that are instantiated for a given statecharts specification. Each formula describes a test sequence in abstract terms in such a way that the formula is true if and only if a statechart specification does *not* allow the test sequence. If the test sequence described by the formula can be performed by the specification, model checking will fail and the tool will generate a counterexample giving an execution sequence that explains why the formula cannot be satisfied. This counterexample is easily mapped into the test sequence by projecting it onto the observable events of the specification.

The contributions of this paper can be summarized as follows. We give a formal semantics for statecharts consistent with the STATEMATE informal interpretation. We apply a family of control-flow and data-

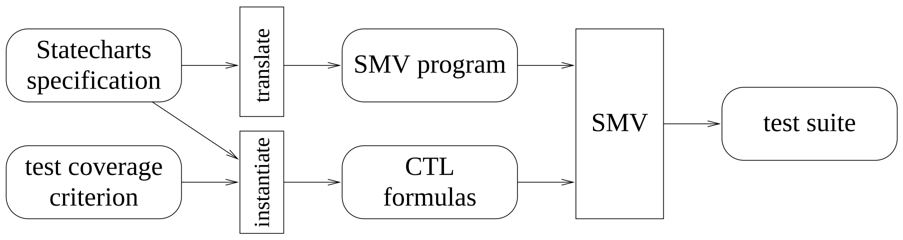


Figure 1: Overview of test generation

flow coverage criteria to statecharts and give a CTL characterization of each coverage criterion. Finally, we demonstrate how to use SMV, an off-the-shelf CTL model checking tool, for the purpose of automatic generation of test suites from statecharts.

Related work. Widely-used models for reactive systems in the testing literature include finite state machines (FSM), especially in hardware and protocol conformance testing. FSM-based testing methods primarily focus on the control-flow oriented test generation such as transition tour, unique-input-output sequence, distinguishing sequence, and characterizing sequence (see [3, 20] for survey). In protocol conformance testing, these methods have been extensively applied to formal description techniques [17] such as SDL and Estelle, and a number of automated tools have been developed (see [9] for survey).

EFSMs extend FSMs with variables to support the succint specification of data-dependent behaviors. If the state space of an EFSM is finite, one can obtain the equivalent FSM by unfolding the EFSM. Thus testing based on EFSMs with finite state space is reduced in principle to testing based on ordinary FSMs. This approach, however, suffers from the well-known state explosion problem which makes test generation often impractical. Even when test generation is feasible, this approach is often impractical because of the test explosion problem, i.e., the number of constructed tests might be too huge to be applied to implementations under test.

A promising alternative is to apply conventional software testing techniques to the generation of tests from EFSMs. In this approach, an EFSM is transformed into a flow graph that models the flow of both control and data of the EFSM and then tests are generated by identifying control and data flow information such as definitions and uses of variables in the flow graph [25]. The flow-graph test generation method is also applied to statecharts [15]. This approach abstracts from the values of variables and hence it can be applicable even if the state space is infinite. The approach, however, requires posterior analysis such as symbolic execution or constraint solving to determine the executability of tests and for the the selection of variable values which make tests executable.

The approach we advocate here is based on translating statecharts into Kripke structures and also suffers from the state explosion problem. However, the formulation of test generation as model checking in our approach enables the use of symbolic model checking [5], a technique that has been proven successful for controlling the state explosion problem. Second, our approach overcomes the test explosion problem by using the flow information of both control and data like the flow-graph approach. Finally, our approach can be seen as complementary to the flow-graph approach. On the one hand, flow graphs can be constructed for systems that are not finite-state. On the other hand, our approach has the advantage that only executable tests are produced.

Recently connections between test generation and model checking has been considered in the literature. A tool that uses test generation algorithms inspired by model checking algorithms is described in [18]. Test generation using counterexamples constructed by model checkers has been applied in several contexts. Mutation analysis is used in the approach of [1]. In [6], test generation is performed from user-specified temporal formulas, while in [10] testing purposes are used to generate tests. No consideration is given to coverage criteria. Some control-flow coverage criteria are considered in [11]. We are not aware of any work that considers the model checking approach to data-flow oriented test generation.

Organization of the paper. Section 2 reviews statecharts and CTL model checking. Section 3 gives a formal definition of the STATEMATE semantics. Section 4 introduces several notions relevant to specification-based testing with statecharts. Section 5 and 6 describe a family of coverage criteria and a test generation method, respectively. Finally, Section 7 concludes the paper with a description of future work.

2 Preliminaries

This section provides a brief introduction to statecharts and CTL model checking.

2.1 Statecharts

A *statechart* is a tuple $Z = (S, \Pi, V, \Theta, T)$ where S, Π, V, and T are sets of states, events, variables, and transitions. Θ is an interpretation of V which assigns to each variable its initial value. To demonstrate the main features of statecharts, we use as the running example the statechart shown in Figure 2, which specifies a simple coffee vending machine. The variable m in Figure 2 is of integer subrange [0,10] and its initial value is defined by $\Theta(m) = 0$.

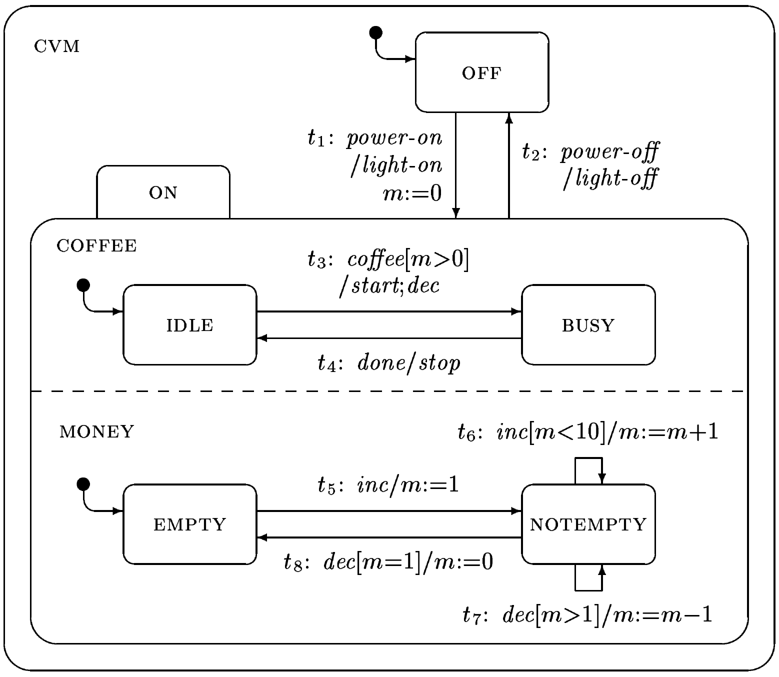

CVM
OFF
t_1: power-on /light-on m:=0
t_2: power-off /light-off
ON
COFFEE
t_3: coffee[m>0] /start; dec
IDLE
BUSY
t_4: done/stop
MONEY
t_6: inc[m<10]/m:=m+1
t_5: inc/m:=1
EMPTY
NOTEMPTY
t_8: dec[m=1]/m:=0
t_7: dec[m>1]/m:=m−1

Figure 2: Example of Statecharts specification

One of the main features of statecharts is the hierarchical and concurrent structure on states. A state is either basic or composite. A composite state is classified as either OR-state or AND-state. An OR-state has substates related by exclusive-or-relation and has exactly one default substate. For example, the OR-state CVM in Figure 2 consists of OFF and ON with OFF as its default substate. Being in CVM implies being in OFF or in ON, but not in both. An AND-state has substates related by and-relation. Being in the AND-state ON implies being in COFFEE and MONEY at the same time. There is a unique state called the *root* at the highest level on the state hierarchy, say CVM.

For a state s, define *children*(s) as the set of substates of s and *children** as the reflexive-transitive closure of *children*. For two states s_1 and s_2, s_1 is an *ancestor* of s_2 if $s_2 \in$ *children*$^*(s_1)$. If, in addition, $s_1 \neq s_2$, we say that s_1 is a *strict ancestor* of s_2.

A *configuration* is a maximal set of states in which a system can be simultaneously. Precisely, $C \subseteq S$ is called a configuration if (i) C contains the root state; (ii) for every AND-state s, either s and all substates of s are in C or they are all not in C; (iii) for every OR-state s, either s and exactly one substate of s are in C or s and all substates of s are not in C. Each configuration can be uniquely characterized by its basic states. In Figure 2, we have the following configurations with {OFF} as the initial configuration: {OFF}, {IDLE, EMPTY}, {IDLE, NOTEMPTY}, {BUSY, EMPTY}, {BUSY, NOTEMPTY}.

We partition the set Π of events into three disjoint subsets Π_I, Π_L and Π_O comprising *input*, *local*, and *output* events, respectively. The occurrence of input events is determined by the environment of a system while local and output events are generated by the system itself. Local events are used for internal communications and are assumed to be invisible to the environment. Input and output events are visible to the environment and constitute the *observables* of a statechart. In Figure 2, we have Π_I = {*power-on*, *power-off*, *coffee*, *done*, *inc*}, Π_L = {*dec*}, and Π_O = {*light-on*, *light-off*, *start*, *stop*}.

A transition t is a tuple (*source*(t), *trigger*(t) *guard*(t), *action*(t), *target*(t)) where *source*(t), *target*(t) $\in S$, *trigger*(t) is a predicate on Π, *guard*(t) is a predicate on V, *action*(t) consists of a set of assignments to V, denoted by *assignments*(t), and a set of events in $\Pi_L \cup \Pi_O$, denoted by *generated*(t).

For a transition t, define *Exits*(t) (resp. *Enters*(t)) as the set of states that a system exits (resp. enters) on taking transition t. For example, we have that *Exits*(t_1) = {OFF} and *Enters*(t_1) = {ON, COFFEE, IDLE, MONEY, EMPTY}. The formal definition for *Exits*(t) and *Enters*(t) can be found in [7]. The *scope* of a transition t, denoted by *scope*(t), is defined as the lowest OR-state in the state hierarchy that is a proper ancestor of both *source*(t) and *target*(t). For example, *scope*(t_2) = CVM and *scope*(t_3) = COFFEE. Two transitions t and t' *conflict* if *scope*(t) is an ancestor of *scope*(t')). For example, t_2 and t_3 conflict because CVM is an ancestor of COFFEE.

2.2 CTL Model Checking

Symbolic model checking [5] is a proven successful technique for the automatic verification of finite state systems. A widely-used temporal logic for symbolic model checking is the branching time temporal logic CTL [8]. Let AP be the underlying set of atomic propositions. The syntax for CTL is defined by the following grammar:

$$\phi ::= p \mid \neg\phi \mid \phi \wedge \phi' \mid EX\phi \mid AX\phi \mid E[\phi U \phi'] \mid A[\phi U \phi']$$

where $p \in AP$ and ϕ, ϕ' range over CTL formulas. The remaining temporal operators are defined by the equivalence rules: $EF\phi \equiv E[trueU\phi]$; $AF\phi \equiv A[trueU\phi]$; $EG\phi \equiv \neg AF(\neg\phi)$; $AG\phi \equiv \neg EF(\neg\phi)$.

The semantics of CTL is defined with respect to a *Kripke structure* $M = (Q, Q_0, L, R)$ where Q is a finite set of states; $Q_0 \subseteq Q$ is the set of initial states; L: $Q \rightarrow 2^{AP}$ is the state-labeling function; and $R \subseteq Q \times Q$ is the set of transitions. A sequence q_0, q_1, q_2, ... of states is a *path* if $(q_i, q_{i+1}) \in R$ for all $i \geq 0$. A path ρ is a q-path if $\rho(0) = q$. The satisfaction relation $\models$ is inductively defined as follows:

- $q \models p$ iff $p \in L(q)$; $q \models \neg\phi$ iff $\neg(q \models \phi)$; $q \models \phi \wedge \phi'$ iff $q \models \phi$ and $q \models \phi'$;
- $q \models EX\phi$ (resp. $AX\phi$) iff for some (resp. all) q-path ρ, $\rho(1) \models \phi$;
- $q \models E[\phi U \phi']$ (resp. $A[\phi U \phi']$) iff for some (resp. all) q-path ρ, there exists $i \geq 0$ such that $\rho(i) \models \phi'$ and $\rho(j) \models \phi$ for all $0 \leq j < i$.

SMV [21] is a symbolic model checker for CTL which represents the state space and transition relation of Kripke structures using OBDDs [4]. An SMV program contains a set of variable declarations to determine its state space and descriptions of the initial states and transition relation, as well as a list of CTL formulas to be checked. Given a system model and a CTL formula, SMV automatically provides either a claim that the formula is satisfied in the system model or else a counterexample falsifying the formula.

Let V be a set of variables. We call v' as the primed version of a variable $v \in V$ and use V' to denote the set of primed versions of all variables in V. We define a *SMV program* as a tuple (V, *Init*, *Trans*) where V is a finite set of variables; *Init* is a predicate on V; and *Trans* is a predicate on $V \cup V'$. Let $\Sigma(V)$ be the set of all interpretations of V. A SMV program (V, *Init*, *Trans*) defines the Kripke structure (Q, Q_0, L, R) such that $Q = \Sigma(V)$; $Q_0 = \{\sigma \in \Sigma(V) \mid \sigma \models Init\}$; $L(\sigma) = \{v{=}\sigma(v) \mid v \in V\}$, for each $\sigma \in \Sigma(V)$; $(\sigma, \sigma') \in R$ if and only if $\langle\sigma, \sigma'\rangle \models Trans$, where $\langle\sigma, \sigma'\rangle$ is the interpretation that assigns $\sigma(v)$ to $v \in V$ and $\sigma'(v)$ to $v' \in V'$.

3 A Formal Definition of the STATEMATE Semantics

This section formally defines a statechart as a Kripke structure based on the STATEMATE semantics. We call each element in Q of a Kripke structure $(Q,\ Q_0,\ L,\ R)$ a *global state* to distinguish it from a state of Statecharts. Similarly we call each element in R as a *global transition.* The formalization is used as the semantic foundation of the test coverage criteria and test generation method presented in the following sections.

The STATEMATE semantics uses the set of nonnegative integers N as the time domain and provides two models of time: synchronous and asynchronous. The main notion of the STATEMATE semantics is a step. A step represents the response of a system to the input events generated by the environment or the local events generated by the system itself. Both time models assume that the execution of a step takes zero time and differ in the way of how time is advanced relative to the execution of steps. In the former model, time is incremented by one time unit after the execution of each step. This time model is mainly used for highly synchronous systems such as synchronous circuits. In the latter, several steps are allowed to take place within a single point in time and time is incremented only when the system becomes stable. Intuitively, stability means that further steps are impossible without new input events. This paper focuses on the asynchronous time model.

State space. We give a set of rules that identify each component of a Kripke structure from a given statechart. First we represent the state space of a statechart $Z = (S,\ \Pi,\ V,\ \Theta,\ T)$ using the following set of global states.

$$Q = \mathit{Config} \times \Sigma(V) \times 2^{\Pi} \times 2^{T \cup IT}$$

where *Config* is the set of all configurations of Z, $\Sigma(V)$ is the set of all interpretations of V, and IT is the set of implicit transitions which shall be discussed in the next section. The set Q of global states captures the following information about a statechart: (i) the states in which a system is; (ii) the values of variables; (iii) the events generated; (iv) the transitions taken.

The set of initial global states is defined as follows: $(C_0, \sigma_0, E_0, \tau_0) \in Q_0$ if C_0 is the initial configuration, $\sigma_0 = \Theta$, $E_0 \in \Pi_I$, and $\tau_0 = \emptyset$. This definition states that only input events can be generated and no transitions are taken prior to the system initialization.

The definition of the label of each global state (C, σ, E, τ) is straightforward:

$$L((C, \sigma, E, \tau)) = in(C) \cup \{v{=}\sigma(v) \mid v \in V\} \cup E \cup \tau$$

where $in(C)$ is a set of propositions defined as $\{in(s) \mid s \in C\}$.

Transition relation. In the asynchronous time model, input events can be introduced to a system only when the system becomes stable. Once input events are introduced, a sequence of steps is executed until the system becomes stable again. A global state (C, σ, E, τ) is *stable* if there exists no generated input or local event, i.e., $E \cap (\Pi_I \cup \Pi_L) = \emptyset$, and there exists no transition that may occur at that state.

We represent the transition relation of a statechart by the set of global transitions $R = R_1 \cup R_2$, where each global transition in R_1 (resp. R_2) is called *step* (resp. *tick*). A step transition starts from a non stable global state and manipulates configurations, variables and local and output events, while a tick transition starts from a stable global state and manipulates input events.

Definition 3.1 Let (C, σ, E, τ) and (C', σ', E', τ') be global states. $((C, \sigma, E, \tau),\ (C', \sigma', E', \tau'))$ is a *step transition* if and only if (1) (C, σ, E, τ) is not stable; (2) $C' = (C - \bigcup_{t \in \tau'} \mathit{Exits}(t)) \cup \bigcup_{t \in \tau'} \mathit{Enters}(t)$; (3) $\sigma' = a(\sigma)$, where $a = \bigcup_{t \in \tau'} \mathit{assignments}(t)$; (4) $E' = \bigcup_{t \in \tau'} \mathit{generated}(t)$; (5) each transition $t \in \tau'$ is enabled at (C, σ), i.e., $\mathit{source}(t) \in C$ and $\sigma \models \mathit{guard}(t)$, and is triggered by E, i.e., $\mathit{trigger}(t)$ evaluates to true for E; no two transitions in τ' conflict; and τ' is maximal, i.e., each transition not in τ' but triggered by E and enabled at (C, σ) conflicts with some transition in τ'.

Definition 3.2 Let (C, σ, E, τ) and (C', σ', E', τ') be global states. $((C, \sigma, E, \tau),\ (C', \sigma', E', \tau'))$ is a *tick transition* if and only if (C, σ, E, τ) is stable; $C' = C$; $\sigma' = \sigma$; $E' \subseteq \Pi_I$; $\tau' = \emptyset$.

Intuitively, a step transition $((C, \sigma, E, \tau), (C', \sigma', E', \tau'))$ corresponds to the execution of the transitions in τ'. A tick transition corresponds to the introduction of input events to a system.

Nondeterminism. A statechart Z is *deterministic* if each non-stable global state has only one successor. Note that stable global states may have more than one successor because tick transitions correspond to the introduction of input events. In this paper, we consider test generation only for deterministic Statecharts. Section 7 discusses a way to extend our approach to non-deterministic Statecharts.

In order to resolve certain classes of nondeterminism, the STATEMATE semantics provides a priority scheme based on the scope of transitions. Let t and t' be transitions conflicting each other. If *scope*(t) is a strict ancestor of *scope*(t'), then t has priority over t'. If *scope*(t) is equivalent to *scope*(t'), t and t' have the same priority. For example, in Figure 2 t_2 has priority over t_3, t_4, t_5, t_6, t_7, and t_8, while t_6 and t_7 have the same priority. With this priority scheme, the coffee vending machines becomes deterministic with respect to $\langle$\{*power-on*\},\{*inc*\},\{*power-off,coffee*\}$\rangle$ because now we have $\langle$\{*light-on*\},$\emptyset$,\{*light-off*\}$\rangle$ as the only possible output sequence.

4 Specification-Based Testing with Statecharts

Runs. Let $Z = (S, \Pi, V, \Theta, T)$ be a statechart. A naive approach to characterizing the behavior of a statechart is to use all the finite paths of its Kripke structure $M(Z)$. This approach, however, is of little use because a path ending at a non stable global state may not provide the information of the output sequence that is supposed to be generated as the response to an input sequence. Therefore we are concerned about only finite paths ending at a stable global state, which we call *runs*. Figure 3 shows a run of the coffee vending machine in which double rectangles represent stable global states. The run corresponds to the execution of the transition sequence t_1, t_5, t_3, t_8 in Figure 2.

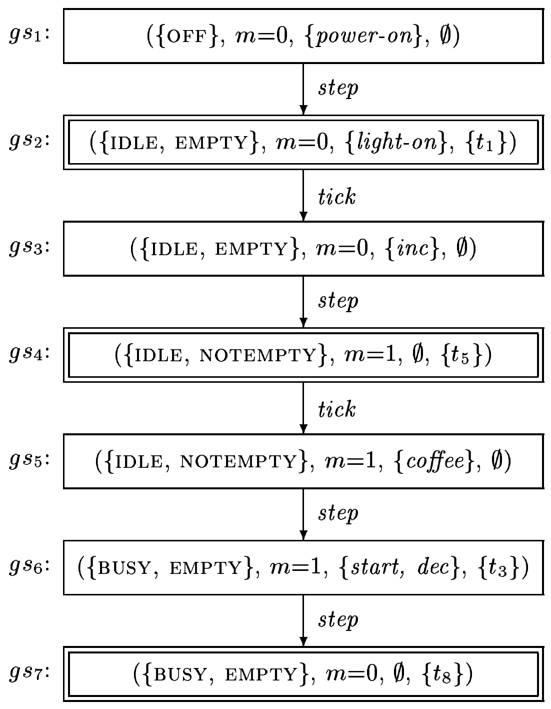

gs1: ({OFF}, m=0, {power-on}, ∅)
step
gs2: ({IDLE, EMPTY}, m=0, {light-on}, {t1})
tick
gs3: ({IDLE, EMPTY}, m=0, {inc}, ∅)
step
gs4: ({IDLE, NOTEMPTY}, m=1, ∅, {t5})
tick
gs5: ({IDLE, NOTEMPTY}, m=1, {coffee}, ∅)
step
gs6: ({BUSY, EMPTY}, m=1, {start, dec}, {t3})
step
gs7: ({BUSY, EMPTY}, m=0, ∅, {t8})

Figure 3: A run for test sequence \{*power-on*\},\{*inc*\},\{*coffee*\}/\{*light-on*\},$\emptyset$,\{*start*\}

A subsequence $(C_i, \sigma_i, E_i, \tau_i)$, ..., $(C_j, \sigma_j, E_j, \tau_j)$ of a run $(C_0, \sigma_0, E_0, \tau_0)$, ..., $(C_n, \sigma_n, E_n, \tau_n)$ is a *superstep* if $(C_{i-1}, \sigma_{i-1}, E_{i-1}, \tau_{i-1})$ is stable, $(C_k, \sigma_k, E_k, \tau_k)$ is not stable for $i \leq k < j$, and $(C_j, \sigma_j, E_j, \tau_j)$ is stable.

We refer to E_i as the input of the superstep and $\Pi_O \cap \bigcup_{i<k\leq j} E_k$ as the output of the superstep. For example, the following shows the three supersteps in Figure 3.

superstep	input	output
gs_1, gs_2	$\{$*power-on*$\}$	$\{$*light-on*$\}$
gs_3, gs_4	$\{$*inc*$\}$	$\emptyset$
gs_5, gs_6, gs_7	$\{$*coffee*$\}$	$\{$*start*$\}$

Test sequences. We refer to a finite word $\bar{i} = i_1...i_n$ over 2^{Π_I} as *input sequence* and a finite word $\bar{o} = o_1...o_n$ over 2^{Π_O} as *output sequence.* We say that a pair of input and output sequences, written as $\bar{i}/\bar{o}$, is a *test sequence* if there is a run such that i_j and o_j are the input and output of the j-th superstep of the run, respectively. Note that since we consider only deterministic statecharts, there is only one output sequence corresponding to each input sequence. Intuitively a test sequence $\bar{i}/\bar{o}$ describes the expected or required response $\bar{o}$ of implementations under test to the input sequence $\bar{i}$. We say that a *test suite* is a set of test sequences. For example, we have the test sequence $\{$*power-on*$\}$,$\{$*inc*$\}$,$\{$*coffee*$\}$/$\{$*light-on*$\}$,$\emptyset$,$\{$*start*$\}$ from the run in Figure 3.

Exit nodes. We compare the nature of test sequences for reactive systems with those for transformational systems. Most of analysis and testing models for transformational systems, e.g., flow graphs[14] and program dependency graphs[19], contain a distinguished node called exit node to model the terminating behavior of such systems. Test sequences in such graphs are naturally defined in terms of paths whose first node is the entry node and last node is the exit node of the graphs. On the other hand, there is no corresponding notion in reactive system models, because the behavior of reactive systems is characterized by their non-terminating computations.

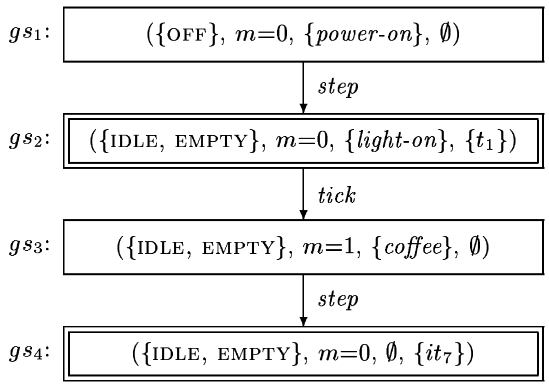
When defining test sequences for statecharts, we do not put any constraint on input sequences and thus allow the execution of test sequences may end at any stable global state. That is, we regard each stable global state as a pseudo-exit node. There are, however, more elaborate approaches to defining exit nodes. A widely used approach in EFSM-based testing is to require that the execution of test sequences end at an initial state from which another test sequence can be applied. In general, testers may want to designate an arbitrary state as an exit node. An interesting notion is marker state in the supervisory control theory by Ramadge and Wonham[22]. This theory distinguishes the paths ending at a state designated as a marker from others and interprets such paths as completed tasks of the modeled system. For example, a tester may want to designate the configuration {IDLE, EMPTY} as a marker for the coffee vending machine and require that the execution of every test sequence end at the marker. In this case, the input sequences in Figure 3 and can be extended with $\{$*done*$\}$ to guarantee that the machine end at the marker.

Testing quiescence. Often we need to test that the system does *not* produce any output in response to some input. For example, the coffee vending machine in configuration {IDLE, EMPTY} when $m = 0$ does not respond to input $\{$*coffee*$\}$ simply because there are no enabled transitions in the corresponding global state. However, if we want to generate a test for such quiescent behavior using the same technique as for observable behaviors, we need to make the absence of output explicit. For this, we extend the set of transitions of the statechart with *implicit transitions.* Implicit transitions are always self-loops with the empty set of output events. The following shows the implicit transitions for the coffee vending machine.

$it_1 = it(\text{OFF}, \textit{power-off}) = (\text{OFF}, \textit{power-off}, \textit{true}, \emptyset, \text{OFF})$
$it_2 = it(\text{OFF}, \textit{coffee}) = (\text{OFF}, \textit{coffee}, \textit{true}, \emptyset, \text{OFF})$
$it_3 = it(\text{OFF}, \textit{done}) = (\text{OFF}, \textit{done}, \textit{true}, \emptyset, \text{OFF})$
$it_4 = it(\text{OFF}, \textit{inc}) = (\text{OFF}, \textit{inc}, \textit{true}, \emptyset, \text{OFF})$
$it_5 = it(\text{OFF}, \textit{dec}) = (\text{OFF}, \textit{dec}, \textit{true}, \emptyset, \text{OFF})$
$it_6 = it(\text{ON}, \textit{power-on}) = (\text{ON}, \textit{power-on}, \textit{true}, \emptyset, \text{ON})$
$it_7 = it(\text{IDLE}, \textit{coffee}) = (\text{IDLE}, \textit{coffee}, \neg(m > 0), \emptyset, \text{IDLE})$
$it_8 = it(\text{BUSY}, \textit{coffee}) = (\text{BUSY}, \textit{coffee}, \textit{true}, \emptyset, \text{BUSY})$
$it_9 = it(\text{IDLE}, \textit{done}) = (\text{IDLE}, \textit{done}, \textit{true}, \emptyset, \text{IDLE})$

$it_{10} = it(\textsc{notempty},\ inc) = (\textsc{notempty}, inc,\ \neg(m < 10),\ \emptyset,\ \textsc{nonempty})$
$it_{11} = it(\textsc{empty}, dec) = (\textsc{empty}, dec,\ true,\ \emptyset,\ \textsc{empty})$
$it_{12} = it(\textsc{notempty},\ dec) = (\textsc{notempty}, dec,\ \neg(m > 1 \vee m = 1),\ \emptyset,\ \textsc{nonempty})$

An input sequence $\bar{i}$ is *explicit* if there exists a test sequence $\bar{i}/\bar{o}$ such that each step transition of its run corresponds to the execution of an explicit transition. Otherwise, it is *implicit.* For example, the input sequence $\langle\{power\text{-}on\},\ \{inc\},\ \{coffee\}\rangle$ is explicit (see Figure 3), while $\langle\{power\text{-}on\},\ \{coffee\}\rangle$ is implicit because the step transition (gs_3, gs_4) in Figure 4 corresponds to the execution of it_7.



gs1: ({OFF}, m=0, {power-on}, ∅)
step
gs2: ({IDLE, EMPTY}, m=0, {light-on}, {t1})
tick
gs3: ({IDLE, EMPTY}, m=1, {coffee}, ∅)
step
gs4: ({IDLE, EMPTY}, m=0, ∅, {it7})

Figure 4: A run for test sequence $\{power\text{-}on\}$,$\{coffee\}/\{light\text{-}on\}$,$\emptyset$

Conformance. Finally we present a conformance relation between specifications written in determinsitic statecharts and implementations under test. For a statechart Z and implementation I, we write $Z\langle\bar{i}\rangle$ and $I\langle\bar{i}\rangle$ for the output sequences of Z and I to an input sequence $\bar{i}$, respectively. We say that I *weakly conforms* to Z if $Z\langle\bar{i}\rangle = I\langle\bar{i}\rangle$, for all explicit input sequences $\bar{i}$. We say that I *strongly conforms* to Z if $Z\langle\bar{i}\rangle = I\langle\bar{i}\rangle$, for all input sequences $\bar{i}$.

5 Test Coverage Criteria for Statecharts

Statecharts specify the required behavior of implementations under test by describing the possible sequences of input and output events in terms of control and data dependencies between the events. Specification-based testing with statecharts aims at determining whether an implementation establishes the desired flow of both control and data expressed in its specification.

5.1 Control Flow Oriented Coverage Criteria

Obviously the strongest test coverage criteria is *path coverage* which requires that all the runs of a statechart be traversed, or equivalently all the input sequences of the statechart be applied to implementations under test. Because there is an infinite number of input sequences, we need to have systematic coverage criteria that select a finite and reasonable number of test sequences satisfying certain conditions. This paper presents a family of test coverage criteria based on the flow information of both control and data in statecharts.

We say that a test sequence $\bar{i}/\bar{o}$ covers a state s (resp. configuration C and transition t) if its run contains global state $(C_i, \sigma_i, E_i, \tau_i)$ such that $s \in C_i$ (resp. $C = C_i$ and $t \in \tau$).

State coverage. A test suite P satisfies *state coverage* if each state is covered by a test sequence in P.

Configuration coverage. A stronger criterion than state coverage may be defined which requires the traversal of each configuration. A test suite P satisfies *configuration coverage* if each configuration is covered by a test sequence in P.

Transition coverage. A test suite P satisfies *weak transition coverage* if each explicit transition is covered by a test sequence in P. A test suite P satisfies *strong transition coverage* if each explicit and implicit transition is covered by a test sequence in P.

5.2 Data Flow Oriented Coverage Criteria

We adopt the following convention which classifies each variable occurrence of a transition as being a definition, computation use (c-use), and predicate use (p-use). Let v be a variable and t be a transition. v is *defined* at t if *assignments*(t) includes an assignment that defines v; v is *c-used* at t if *assignments*(t) includes an assignment that references v; v is *p-used* at t if *guard*(t) references v. We denote by *def*(v), *c-use*(v), and *p-use*(v) the sets of transitions that define, c-use, and p-use v, respectively. In the coffee vending machine, we have *def*$(m) = \{t_1, t_5, t_6, t_7, t_8\}$, *c-use*$(m) = \{t_6, t_7\}$, *p-use*$(m) = \{t_3, t_6, t_7, t_8\}$.

Let t and t' be transitions and $gs_0, \ldots, gs_n$ be a run. Suppose that the step transitions $(gs_i,\ gs_{i+1})$ and $(gs_j,\ gs_{j+1})$ such that $0 \leq i < j < n$ correspond to the execution of t and t', respectively. The run is a *definition-clear run* with respect to v from t to t' if each step transition $(gs_k,\ gs_{k+1})$ does not correspond to the execution of any transition at which v is defined, for $i < k < j$.

We define associations between definitions and uses of a variable as follows: a tuple (v, t, t') is a *def-c-use association* (resp. *def-p-use association*) if $t \in$ *def*(v), $t' \in$ *c-use*(v) (resp. $t' \in$ *p-use*(v)), and there exists a definition-clear run with respect to v from t to t'. A *def-use association* is either a def-c-use association or def-p-use association. Table 1 shows the def-use associations for the coffee vending machine. For example, consider the def-p-use association (m, t_5, t_3). The definition of m at t_5 can reach the use of m at t_3 through the definition-clear run shown in Figure 3. In general, there are three types of associations between definitions and uses in statecharts. The first type includes associations which occur within an OR-state, e.g., $(m,\ t_5,\ t_6)$. Associations occurring in ordinary EFSMs belong to this type. The second type is caused by the hierarchical structure on states, e.g., $(m,\ t_1,\ it_7)$. The third type is caused by the concurrent structure on states, e.g., $(m,\ t_5,\ t_3)$.

Table 1: The def-use associations for the coffee vending machine

def-c-use associations	def-p-use associations
$(m,\ t_5,\ t_6)$	$(m,\ t_5,\ t_3)$, $(m,\ t_5,\ t_6)$, $(m,\ t_5,\ t_8)$
$(m,\ t_6,\ t_6)$, $(m,\ t_6,\ t_7)$	$(m,\ t_6,\ t_3)$, $(m,\ t_6,\ t_6)$, $(m,\ t_6,\ t_7)$
$(m,\ t_7,\ t_6)$, $(m,\ t_7,\ t_7)$	$(m,\ t_7,\ t_3)$, $(m,\ t_7,\ t_6)$, $(m,\ t_7,\ t_7)$, $(m,\ t_7,\ t_8)$

We also account for the data flow caused by implicit transitions. A def-use association for a variable v is called *implicit* if v is referenced by the guard of an implicit transition. Note that only p-uses are possible for implicit transitions. In the coffee vending machine, we have the following implicit def-p-use associations: (m, t_1, it_7), (m, t_6, it_{10}), (m, t_8, it_7).

We say that a test sequence $\bar{i}/\bar{o}$ covers a def-use association (v, t, t') if its run is a definition-clear run with respect to v from t and t'.

All-def coverage. A test suite P satisfies *weak all-def coverage* if for each variable v and each transition t such that $t \in$ *def*(v), some def-use association (v, t, t') is covered by a test sequence in P. A test suite P satisfies *strong all-def coverage* if for each variable v and each transition t such that $t \in$ *def*(v), some explicit def-use association (v, t, t') or implicit def-use association (v, t, it) is covered by a test sequence in P.

All-use coverage. A test suite P satisfies *weak all-use coverage* if for each variable v and each transition t such that $t \in$ *def*(v), each def-use association (v, t, t') is covered by a test sequence in P. A test suite P

satisfies *strong all-use coverage* if for each variable v and each transition t such that $t \in def(v)$, each explicit def-use association (v, t, t') and implicit def-use association (v, t, it) is covered by a test sequence in P.

6 A Test Generation Method for Statecharts

This section shows that test generation from statecharts can be automatically performed by using the SMV's ability to construct counterexamples. Briefly the generation of a test suite from a statechart and a test coverage criterion consists of the following steps.

- An SMV program is constructed from the statechart.
- A set of CTL formulas is constructed from the criterion.
- A test suite is constructed by model-checking the CTL formulas against the SMV program and projecting the obtained counterexamples onto the observable events.

6.1 Statecharts as SMV Programs

Because we use the symbolic model checker SMV, we do not enumerate Kripke structures explicitly but encode them symbolically using a set of variables and predicates. That is, we translate a statechart into a SMV program, which is a symbolic representation of Kripke structure, by encoding the semantics given in Section 3 in terms of variables and predicates. A detailed description of the translation is straightforward but tedious and is omited here due to the space limit. Full details, along with a correctness proof, can be found in [16].

During the translation, we introduce several predicates that are used in the CTL formulas expressing the coverage criteria. For each state s of a statechart, we define predicate $in(s)$ which is true in a global state if it corresponds to a configuration that contains s. A predicate *stable* is true in stable global states. Finally the following predicates are used to encode the information on definitions and uses of a variable v.

$$\begin{aligned}&d(v) ::= \textstyle\bigvee_{t \in def(v)} t\\ &u(v) ::= \textstyle\bigvee_{t \in p\text{-}use(v) \cup c\text{-}use(v)} t\\ &im\text{-}u(v) ::= \textstyle\bigvee_{it \in implicit\text{-}p\text{-}use(v)} it\end{aligned}$$

For example, we have $d(m) ::= t_1 \vee t_5 \vee t_6 \vee t_7 \vee t_8$, $u(m) ::= t_3 \vee t_6 \vee t_7 \vee t_8$, and $im\text{-}u(m) ::= it_7 \vee it_{10} \vee it_{12}$ for the coffee vending machine.

6.2 Coverage Criteria as CTL Formulas

Each coverage criterion is represented as a set of CTL templates. For a given statechart, the set of templates is instantiated into a set of CTL formulas with the predicates defined for the statechart. The resulting set of CTL formulas captures exactly the coverage criterion for the given statechart.

6.2.1 CTL Formulas for Control Flow Coverage Criteria

We begin with the state coverage criteria which requires that for each state s of a statechart, there exists at least one run covering s. The Kripke structure corresponding to a statechart has a run covering s if and only if (i) there exists global state gs_i which is reachable from an initial global state gs_0 and at which $in(s)$ is satisfied and (ii) there exists a global state gs_j which is reachable from gs_i and at which *stable* is satisfied (see Figure 5). We express the requirement as the CTL formula $EF\,(in(s) \wedge EF\,stable)$.

Now we take the negation of the above formula and run SMV against the negated formula $\neg EF\,(in(s) \wedge EF\,stable)$ because we are interested in generating runs covering s instead of checking the satisfiability of the original formula. If there exists a run covering s, SMV generates a counterexample which corresponds to a run covering s. Otherwise, SMV provides the result of *true*. There are two cases in which SMV provides

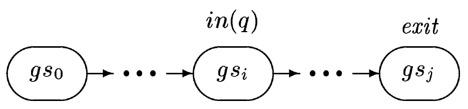


Figure 5: A test sequence covering state q

true against the negated formula. First, the global state gs_i is not reachable from any initial global state, i.e., $EF\, in(s)$ is not satisfied. Second, a statechart cannot reach a global state at which *stable* is satisfied, i.e., $EF\ stable$ is not satisfied. For example, consider $\neg EF\ (in(\text{B}) \wedge EF\ stable)$ whose purpose is to cover the state B in Figure 6. Although B is reachable from an initial global state, there is no run covering B because there is an infinite sequence consisting of only step transitions

$$(\{\text{A}\}, m = 1, \{\alpha\}, \emptyset), (\{\text{B}\}, m = 1, \{\beta\}, \{t_1\}), (\{\text{A}\}, m = 1, \{\alpha\}, \{t_2\}), \ldots$$

and hence the statechart cannot reach a stable global state.

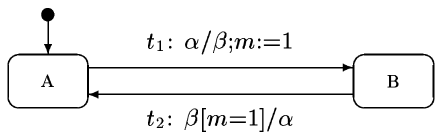


Figure 6: A non-terminating statechart

As mentioned before, it may be required that each run end at an initial state or an arbitrary state marked by testers. Let *exit* be a predicate defined as *stable* if there is no marker, and $stable \wedge in(C)$ if C is a configuration designated as a marker. We can simply express the requirement of markers using $EF\ (in(s) \wedge EF\ exit)$.

Let $BS \subseteq S$ be the set of basic states. We note that a test suite covering all basic states also covers all other states because of the hierarchy. To generate test sequences satisfying state coverage, we use the following set of CTL formulas.

State coverage. $\{\neg EF\ (in(s) \wedge EF\ exit) \mid s \in BS\}$

We can define the CTL formulas for other control-flow oriented criteria in a similar way.

Configuration coverage. $\{\neg EF\ (in(c) \wedge EF\ exit) \mid c \in Config\}$

Weak transition coverage. $\{\neg EF\ (t \wedge EF\ exit) \mid t \in T\}$

Strong transition coverage. $\{\neg EF\ (t \wedge EF\ exit) \mid t \in T\} \cup \{\neg EF\ (it \wedge EF\ exit) \mid it \in IT\}$

6.2.2 CTL Formulas for Data Flow Coverage Criteria.

The requirement for a def-use association (v, t, t') can be stated as follows: (i) there exists a global state gs_i which is reachable from an initial global state gs_0 and at which t is satisfied; (ii) there exists a path gs_{i+1} ... gs_{j-1} which starts from a successor of gs_i and contains no definition of v until gs_j at which t' is satisfied; (iii) there exists a global state gs_k which is reachable from the global state gs_j and at which *exit* is satisfied (see Figure 7). We express this requirement as $EF\ (t \wedge EX\ E\ [\neg d(v)\ U\ (t' \wedge EF\ exit)])$.

We determine whether each tuple (v, t, t') such that $t \in def(v)$ and $t' \in use(v)$ is a def-use association or not by associating the negation of the above formula $\neg EF\ (t \wedge EX\ E\ [\neg d(v)\ U\ (t' \wedge EF\ exit)])$ with the

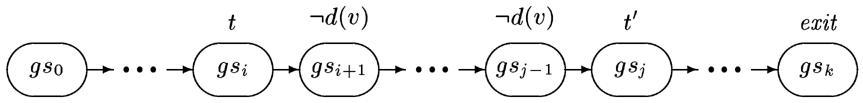


Figure 7: A test sequence covering def-use association (v, t, t')

tuple. If SMV generates the result of *true* against the negated formula, the tuple is not a def-use association. Otherwise, the counterexample generated by SMV corresponds to a run covering the def-use association (v, t, t').

Weak all-def coverage. $\{\neg EF\,(t \wedge EX\,E\,[\neg d(v)\,U\,(u(v) \wedge EF\,exit)]) \mid v \in V, t \in def(v)\}$

Strong all-def coverage.

$$\{\neg EF\,(t \wedge EX\,E\,[\neg d(v)\,U\,((u(v) \vee im\text{-}u(v)) \wedge EF\,exit)]) \mid v \in V, t \in def(v)\}$$

Weak all-use coverage.

$$\{\neg EF\,(t \wedge EX\,E\,[\neg d(v)\,U\,(t' \wedge EF\,exit)]) \mid v \in V, t \in def(v), t' \in use(v)\}$$

Strong all-use coverage.

$$\{\neg EF\,(t \wedge EX\,E\,[\neg d(v)\,U\,(t' \wedge EF\,exit)]) \mid v \in V, t \in def(v), t' \in use(v) \cup implicit\text{-}p\text{-}use(v)\}$$

6.3 Test generation as Model Checking

Each formula in the set representing the coverage criterion for a statechart is model checked against the SMV program of the statechart. If the formula is false, SMV produces a counterexample which, projected onto the input and output event of the statechart, yields a test sequence to be included in a test suite. If the formula is true, no counterexample is produced which implies that there is no test sequence for the coverage expressed by the formula.

For example, consider the formula $\neg EF\,(t_3 \wedge EF\,stable)$ from the transition coverage criterion formula set for the coffee vending machine. Model checking of this formula produces the counterexample shown in Appendix A. The counterexample is the symbolic representation of the run in Figure 3 and covers the transition t_3. When generating counterexamples, SMV describes an initial state by providing the values of all variables and predicates. The other states are described in terms of only the values that are changed from one state to the next. The counterexample is mapped into the test sequence {*power-on*},{*inc*},{*coffee*}/{*light-on*},$\emptyset$,{*start*}. Test suites for the coffee vending machine with respect to several coverage criteria are shown in Appendix B.

7 Conclusions and Future Work

We have presented a model checking approach to automatic test generation from statecharts. Test suites are generated according to a set of widely used coverage criteria based on the flow of control and data. Each coverage criterion is expressed as a set of formulas in the temporal logic CTL. Each formula defines one test sequence in such a way that the formula is satisfied by a statechart if and only if the test sequence is infeasible in the specification. Otherwise, the model checker produces a counterexample for the formula. The counterexample, projected onto the observable events of the statechart, yields a test sequence. The main advantage of the approach is that only feasible or executable test sequences are generated, which obviates the need of posterior analysis such as symbolic execution or constraint solving.

Formalisms. This paper does not consider several important features of statecharts such as actions associated with states, transitions with multiple source and target states, compound transitions, histories, and real-time constructs such as timeout events and scheduled actions. However, it is fairly simple to extend our test generation method once we have a formal definition for these features in terms of Kripke structures.

Many variants of semantics have been proposed for statecharts [2]. Semantic differences affect only the translation method from statecharts into inputs to a model checker. In fact, our formalization of coverage criteria as collections of CTL formulas is language-independent and is applicable with minor modifications to any kind of specification languages based on EFSMs, e.g., SDL or Estelle.

Other coverage criteria. A number of other coverage criteria based on control and data flow analysis have been proposed in the software testing literature (see, for example, [23]). Some of these coverage criteria cannot be handled using SMV, because their CTL properties contain universal path quantifiers. For example, all-du-paths coverage criterion requires that all definition-clear runs for a definition-use pair be traversed. To generate tests for this criterion correctly, SMV would have to produce all counterexamples to each CTL formula instead of only one. Such coverage criteria can be handled by extending SMV to produce multiple counterexamples or by using a different model checker that has this facility.

Nondeterminism. In the case of non-deterministic statecharts, there may be more than one possible output sequence for a given input sequence. In this situation, a single counterexample produced by the model checker is not enough for the input sequence, since it will identify only one output sequence among all possible ones. A possible solution to this problem is to treat the counterexample as prescribing only the *input* sequence. An extra step is then needed to find all output sequences corresponding to this input sequence. If we have a model checker that produces multiple counterexamples to a formula, as discussed in the previous paragraph, we can express the input sequence as a CTL formula and give its negation to the model checker. The set of counterexamples produced by the model checker will contain all feasible output sequences.

Optimizations. Often the test suite constructed by our approach will contain redundant tests. For example, in the transition coverage test suite for the coffee vending machine, the test sequence to cover transition t_7: $\{\textit{power-on}\},\{\textit{inc}\},\{\textit{inc}\},\{\textit{dec}\}/\{\textit{light-on}\},\emptyset,\emptyset,\emptyset$ will also cover transitions t_1, t_5, and t_6. It is, therefore, necessary to have heuristics to minimize the number of generated tests without sacrificing the coverage they provide.

References

[1] P. Ammann, P. Black, and W. Majurski, "Using Model Checking to Generate Tests from Specifications," in *Proceedings of 2nd IEEE International Conference on Formal Engineering Methods*, pp. 46-54, 1998.

[2] M. von der Beeck, "A Comparison of Statecharts Variants," in *Formal Techniques in Real-Time Fault-Tolerant Systems*, Lecture Notes in Computer Science, Vol. 863, pp. 128-148, Springer-Verlag, 1994.

[3] G.v. Bochmann and A. Petrenko, "Protocol Testing: Review of Methods and Relevance for Software Testing," in *Proceedings of the 1994 International Symposium on Software Testing and Analysis*, pp. 109-124, 1994.

[4] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, Vol. C-35, No. 6, pp. 677-691, Aug. 1986.

[5] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond," in *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, 1990.

[6] J. Callahan, F. Schneider, and S. Easterbrook, "Specification-based Testing Using Model Checking," in *Proceedings of 1996 SPIN Workshop*, also Technical Report NASA-IVV-96-022, West Virginia Univeristy, 1996.

[7] W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J.D. Reese, "Model Checking Large Software Specifications," *IEEE Transactions on Software Engineering*, Vol. 24, No. 7, pp. 498-520, July 1998.

[8] E.M. Clarke, E.A. Emerson, and A.P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 2, pp. 244-263, Apr. 1986.

[9] R. Dssouli, K. Saleh, E. Aboulhamid, A. En-Nouaary, and C. Bourhfir, "Test Development for Communication Protocols: towards Automation," *Computer Networks*, Vol. 31, Vol. 17, pp. 1835-1872, 1999.

[10] A. Engels, L. Feijs, and S. Mauw, "Test Generation for Intelligent Networks Using Model Checking," in *Proceedings of TACAS '97*, Lecture Notes in Computer Science, Vol. 1217, pp. 384-398, Springer-Verlag, 1997.

[11] A. Gargantini and C. Heitmeyer, "Using Model Checking to Generate Tests from Requirements Specifications," in *Proceedings of the Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 6-10, 1999.

[12] D. Harel, "Statecharts: a Visual Formalism for Complex Systems," *Science of Computer Programming*, Vol. 8, pp. 231-274, 1987.

[13] D. Harel and A. Naamad, "The STATEMATE Semantics of Statecharts," *ACM Transactions on Software Engineering and Methodologies*, Vol. 5, No. 4, pp. 293-333, Oct. 1996.

[14] M.S. Hecht, *Flow Analysis of Computer Programs*, Elsevier North-Holland, 1977.

[15] H.S. Hong, Y.G. Kim, S.D. Cha, D.H. Bae, and H. Ural, "A Test Sequence Selection Method for Statecharts," *Journal of Software Testing, Verification, and Reliability*, Vol. 10, No. 4, pp. 203-227, Dec. 2000.

[16] H.S. Hong, I. Lee, O. Sokolsky, and S.D. Cha, "Automatic Test Generation from Statecharts Using Model Checking," Technical Report MS-CIS-01-07, Department of Computer and Information Science, University of Pennsylvania, 2001. Available at `http://www.cis.upenn.edu/techreports.html`.

[17] ITU-T, Recommendation X. 904 - Information Technology - Open Distributed Processing - Reference Model: Architectural Semantics, Dec. 1997.

[18] T. Jeron and P. Morel, "Test Generation Derived From Model Checking," in *Proceedings of CAV '99*, LNCS 1633, pp. 108-121, 1999.

[19] B. Korel, "The Program Dependence Graph in Static Program Testing," *Information Processing Letters*, Vol. 24, pp. 103-108, Jan. 1987.

[20] D. Lee and M. Yannakakis, "Principles and Methods of Testing Finite State Machines - A Survey," *Proceedings of the IEEE*, Vol. 84, No. 8, pp. 1090-1123, Aug. 1996.

[21] K.L. McMillan, *Symbolic Model Checking – an Approach to the State Explosion Problem*, Kluwer Academic Publishers, 1993.

[22] P.J. Ramadge and W.M. Wonham, "Supervisory Control of a Class of Discrete Event Processes," *SIAM Journal of Control and Optimization*, Vol. 25, No. 1, pp. 206-230, Jan. 1987.

[23] S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 4, pp. 367-375, Apr. 1985.

[24] J. Rumbaugh, G. Booch, and I. Jacobson, *The UML Reference Guide*, Addison Wesley Longman, 1999.

[25] H. Ural, K. Sale, and A. Williams, "Test Generation Based on Control and Data Dependencies within System Specifications in SDL," *Computer Communications*, Vol. 23, p. 609-627, 2000.

A The counterexample for $\neg EF(t_3 \wedge EF\,stable)$

```
-- specification !EF (t3 & EF stable) is false
-- as demonstrated by the following execution sequence
state 1.1:
stable=0
CVM=off COFFEE=idle MONEY=empty
in-notempty=0 in-empty=0 in-money=0 in-busy=0 in-idle=0 in-coffee=0 in-on=0 in-off=1 in-cvm=1
m=0
power-on=1 power-off=0 coffee=0 done=0 inc=0 dec=0 light-on=0 light-off=0 start=0 stop=0
conflict-t8=0 conflict-t7=0 conflict-t6=0 conflict-t5=0
conflict-t4=0 conflict-t3=0 conflict-t2=0 conflict-t1=0
mayoccur-t8=0 mayoccur-t7=0 mayoccur-t6=0 mayoccur-t5=0
mayoccur-t4=0 mayoccur-t3=0 mayoccur-t2=0 mayoccur-t1=1
t1=0 t2=0 t3=0 t4=0 t5=0 t6=0 t7=0 t8=0
it1=0 it2=0 it3=0 it4=0 it5=0 it6=0 it7=0 it8=0 it9=0 it10=0 it11=0 it12=0

state 1.2:
stable=1
CVM=on in-empty=1 in-money=1 in-idle=1 in-coffee=1 in-on=1 in-off=0 power-on=0 light-on=1 mayoccur-t1=0 t1=1

state 1.3:
stable=0 inc=1 light-on=0 mayoccur-t5=1 t1=0

state 1.4:
stable=1 MONEY=notempty in-notempty=1 in-empty=0 inc=0 m=1 mayoccur-t5=0 t5=1

state 1.5:
stable=0 coffee=1 mayoccur-t3=1 t5=0

state 1.6:
COFFEE=busy in-busy=1 in-idle=0 coffee=0 dec=1 start=1 mayoccur-t8=1 mayoccur-t3=0 t3=1

state 1.7:
stable=1 MONEY=empty in-notempty=0 in-empty=1 m=0 dec=0 start=0 mayoccur-t8=0 t3=0 t8=1

resources used:
user time: 0.36 s, system time: 0.03 s
BDD nodes allocated: 14070
Bytes allocated: 1376256
BDD nodes representing transition relation: 4496 + 6
```

B Test Suites for the Coffee Vending Machine

State Coverage

state	result
OFF	$\emptyset/\emptyset$
IDLE	$\{\textit{power-on}\}$ / $\{\textit{light-on}\}$
BUSY	$\{\textit{power-on}\},\{\textit{inc}\},\{\textit{coffee}\}$ / $\{\textit{light-on}\},\emptyset,\{\textit{start}\}$
EMPTY	$\{\textit{power-on}\}$ / $\{\textit{light-on}\}$
NOTEMPTY	$\{\textit{power-on}\},\{\textit{inc}\},\{\textit{inc}\}$ / $\{\textit{light-on}\},\emptyset,\emptyset$

Configuration Coverage

state	result
{OFF}	$\emptyset/\emptyset$
{IDLE, EMPTY}	$\{\textit{power-on}\}$ / $\{\textit{light-on}\}$
{IDLE, NOTEMPTY}	$\{\textit{power-on}\},\{\textit{inc}\},\{\textit{inc}\}$ / $\{\textit{light-on}\},\emptyset,\emptyset$
{BUSY, EMPTY}	$\{\textit{power-on}\},\{\textit{inc}\},\{\textit{coffee}\}$ / $\{\textit{light-on}\},\emptyset,\{\textit{start}\}$
{BUSY, NOTEMPTY}	$\{\textit{power-on}\},\{\textit{inc}\},\{\textit{coffee}\}$ / $\{\textit{light-on}\},\emptyset,\{\textit{start}\}$

Strong Transition Coverage

transition	result
t_1	$\{\mathit{power\text{-}on}\}$ / $\{\mathit{light\text{-}on}\}$
t_2	$\{\mathit{power\text{-}on}\},\{\mathit{power\text{-}off}\}$ / $\{\mathit{light\text{-}on}\},\{\mathit{light\text{-}off}\}$
t_3	$\{\mathit{power\text{-}on}\},\{\mathit{inc}\},\{\mathit{coffee}\}$ / $\{\mathit{light\text{-}on}\},\emptyset,\{\mathit{start}\}$
t_4	$\{\mathit{power\text{-}on}\},\{\mathit{inc}\},\{\mathit{coffee}\},\{\mathit{done}\}$ / $\{\mathit{light\text{-}on}\},\emptyset,\{\mathit{start}\},\{\mathit{stop}\}$
t_5	$\{\mathit{power\text{-}on}\},\{\mathit{inc}\}$ / $\{\mathit{light\text{-}on}\},\emptyset$
t_6	$\{\mathit{power\text{-}on}\},\{\mathit{inc}\},\{\mathit{inc}\}$ / $\{\mathit{light\text{-}on}\},\emptyset,\emptyset$
t_7	$\{\mathit{power\text{-}on}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{dec}\}$ / $\{\mathit{light\text{-}on}\},\emptyset,\emptyset,\emptyset$
t_8	$\{\mathit{power\text{-}on}\},\{\mathit{inc}\},\{\mathit{dec}\}$ / $\{\mathit{light\text{-}on}\},\emptyset,\emptyset$
it_1	$\{\mathit{power\text{-}off}\}$ / $\emptyset$
it_2	$\{\mathit{coffee}\}$ / $\emptyset$
it_3	$\{\mathit{done}\}$ / $\emptyset$
it_4	$\{\mathit{inc}\}$ / $\emptyset$
it_5	*infeasible*
it_6	$\{\mathit{power\text{-}on}\},\{\mathit{power\text{-}on}\}$ / $\{\mathit{light\text{-}on}\},\emptyset$
it_7	$\{\mathit{power\text{-}on}\},\{\mathit{coffee}\}$ / $\{\mathit{light\text{-}on}\},\emptyset$
it_8	$\{\mathit{power\text{-}on}\},\{\mathit{inc}\},\{\mathit{coffee}\},\{\mathit{coffee}\}$ / $\{\mathit{light\text{-}on}\},\emptyset,\{\mathit{start}\},\emptyset$
it_9	$\{\mathit{power\text{-}on}\},\{\mathit{done}\}$ / $\{\mathit{light\text{-}on}\},\emptyset$
it_{10}	$\{\mathit{power\text{-}on}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{inc}\}$ / $\{\mathit{light\text{-}on}\},\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset$
it_{11}	*infeasible*
it_{12}	*infeasible*

Strong all-use coverage

tuple	result
(m, t_1, it_7)	$\{\mathit{power\text{-}on}\},\{\mathit{coffee}\}/\{\mathit{light\text{-}on}\},\emptyset$
(m, t_5, t_3)	$\{\mathit{power\text{-}on}\},\{\mathit{inc}\},\{\mathit{coffee}\}$ / $\{\mathit{light\text{-}on}\},\emptyset,\{\mathit{start}\}$
(m, t_5, t_6)	$\{\mathit{power\text{-}on}\},\{\mathit{inc}\},\{\mathit{inc}\}$ / $\{\mathit{light\text{-}on}\},\emptyset,\emptyset$
(m, t_5, t_8)	$\{\mathit{power\text{-}on}\},\{\mathit{inc}\},\{\mathit{dec}\}$ / $\{\mathit{light\text{-}on}\},\emptyset,\emptyset$
(m, t_6, t_3)	$\{\mathit{power\text{-}on}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{coffee}\}$ / $\{\mathit{light\text{-}on}\},\emptyset,\emptyset,\{\mathit{start}\}$
(m, t_6, t_6)	$\{\mathit{power\text{-}on}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{inc}\}$ / $\{\mathit{light\text{-}on}\},\emptyset,\emptyset,\emptyset$
(m, t_6, t_7)	$\{\mathit{power\text{-}on}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{dec}\}$ / $\{\mathit{light\text{-}on}\},\emptyset,\emptyset,\emptyset$
(m, t_6, it_{10})	$\{\mathit{power\text{-}on}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{inc}\}$ / $\{\mathit{light\text{-}on}\},\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset$
(m, t_7, t_3)	$\{\mathit{power\text{-}on}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{dec}\},\{\mathit{coffee}\}$ / $\{\mathit{light\text{-}on}\},\emptyset,\emptyset,\emptyset,\{\mathit{start}\}$
(m, t_7, t_6)	$\{\mathit{power\text{-}on}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{dec}\},\{\mathit{inc}\}$ / $\{\mathit{light\text{-}on}\},\emptyset,\emptyset,\emptyset,\emptyset$
(m, t_7, t_7)	$\{\mathit{power\text{-}on}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{dec}\},\{\mathit{dec}\}$ / $\{\mathit{light\text{-}on}\},\emptyset,\emptyset,\emptyset,\emptyset,\emptyset$
(m, t_7, t_8)	$\{\mathit{power\text{-}on}\},\{\mathit{inc}\},\{\mathit{inc}\},\{\mathit{dec}\},\{\mathit{dec}\}$ / $\{\mathit{light\text{-}on}\},\emptyset,\emptyset,\emptyset,\emptyset$
(m, t_8, it_7)	$\{\mathit{power\text{-}on}\},\{\mathit{inc}\},\{\mathit{dec}\},\{\mathit{coffee}\}$ / $\{\mathit{light\text{-}on}\},\emptyset,\emptyset,\emptyset$

Infeasible associations: (m, t_1, t_3), (m, t_1, t_6), (m, t_1, t_7), (m, t_1, t_8), (m, t_1, it_{10}), (m, t_1, it_{12}), (m, t_5, t_7), (m, t_5, it_7), (m, t_5, it_{10}), (m, t_5, it_{12}), (m, t_6, t_8), (m, t_6, it_7), (m, t_6, it_{12}), (m, t_7, it_7), (m, t_7, it_{10}), (m, t_7, it_{12}), (m, t_8, t_3), (m, t_8, t_6), (m, t_8, t_7), (m, t_8, t_8), (m, t_8, it_{10}), (m, t_8, it_{12}).

Automatic generation of tests from Statechart specifications

Simon Burton, John Clark, John McDermid

Department of Computer Science.
University of York, Heslington, York. YO10 5DD, England.
{Simon.Burton, John.Clark, John.McDermid}@cs.york.ac.uk

Abstract

This paper describes how formal methods can be exploited in the automatic generation of tests from Statechart specifications, and in a manner that can be extended to other graphical and tabular notations. Intermediate formal representations of both the specification and the testing conditions are generated. Based on these formalisations, properties of the specification and the tests are analysed and a combination of constraint solvers and automated theorem proving tools are used to generate the tests. The paper also summarises experience in applying the techniques to an industrially relevant case study.

1 Introduction

Graphical notations such as Statecharts [13], SDL [35] and UML [9] and tabular notations such as the SCR method [16] are becoming increasingly popular as a means of specifying software systems. Noteworthy applications of these techniques are telecommunication systems and embedded controllers for safety-critical systems. The functional correctness of such systems is crucial to ensure their safe and/or economic operation. Testing is the primary means of verifying these systems at present. However, the methods currently employed by industry are expensive, time consuming and error prone. A more reliable and automated approach to testing systems designed using graphical and tabular specifications is therefore required.

Formal methods can greatly benefit the testing process. They allow for a precise and unambiguous representation of the system specification and are amenable to rigorous and automated analysis. However, formal methods have still to gain widespread use in the software industry. One of the perceived barriers to the acceptance of formal methods is their reliance on mathematical notations that are not well accepted by the domain engineers who are typically not from a computer science background. There is a need, as Ould [27] put it, to "disguise" the formality so that an impractical amount of formal methods skill is not a pre-requisite to effective V&V. Graphical and tabular specification notations are one means of bridging the "semantic gap" between the engineers and the formal methods.

This paper describes how combining graphical and tabular notations with formal methods can facilitate rigorous and automated testing. The paper presents the results of an industrially sponsored research program to develop a framework of automated testing techniques that can be applied to a number of graphical and tabular notations. The framework consists of the following 3 elements. An intermediate formal representation of the specification is automatically generated based on an understanding of the semantics of the source notation. Testing heuristics (both partitioning and fault-based) are also formalised using the same intermediate notation. This allows test

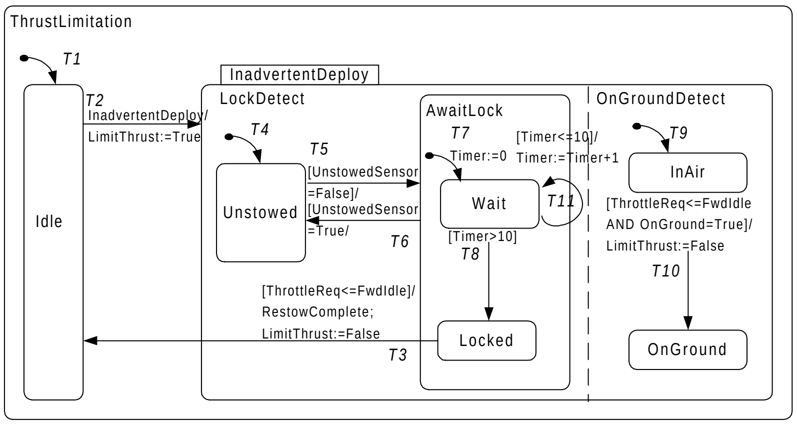


Figure 1: Example Statechart - Thrust Limitation

cases to be automatically generated that are correct through construction with respect to the original specification and testing heuristics. Where the specification includes reference to a persistent data state, the test cases are sequenced to allow the initial state for a test to be set and the effects of a test on the state to be verified. The formalisation also allows for properties of the specifications and testing heuristics to be analysed. The notions of weak and strong detectability, normally associated with program-based mutation testing, are described in the context of specification-based testing. Weak and strong fault detection conditions are used to assess the testability of the specifications and to select appropriate testing strategies. Automation is achieved through the integration of a number of constraint solving and theorem proving technologies. By using the same intermediate notation, the techniques can be re-used across a number of source notations. The framework is illustrated in this paper by describing its application to the Statechart notation. In addition to the use of non-Statechart specific testing techniques, this paper differs from others which discuss test generation from Statecharts in the testing methods used. For example, Bogdanov et al. [4] assume that transition operations are independently testable while Hong et al. [18] apply data flow analysis to derive test sequences.

In sections 2 to 4 we show how specifications written in a subset of Statecharts can be formalised, how testing heuristics can be formalised and how constraint solvers and theorem provers can be used to generate the test case instances. Section 5 describes experience in applying the techniques to realistic systems and the paper concludes with a summary of the key contributions of the work, a discussion of further applications of the techniques and directions for future work.

2 Formalising Statechart transitions

A subset of the Statecharts [13] notation was adopted for the study that was sufficient to model the applications of interest (aerospace control systems) yet restrictive enough to allow a straightforward formalisation. The semantics of the subset were taken from Harel and Naamad's definition [15] and are also implemented by the STATEMATE tool [14] which was used in the production of the case study material. The example Statechart in Figure 1 is typical of those specified in the target domain and describes a portion of the software in an Electronic Engine Controller (EEC). The Statechart describes the cancellation of a thrust limiting mechanism following an inadvertent deployment (e.g. due to a mechanical fault) of the engine thrust reverser doors.

Statecharts are an extension of finite state machines that include concurrency, state hierarchy and data. States in the system can be OR-states, AND-states or basic states. When an OR-state is active, one and only one of its immediate sub-states is active (where the hierarchy is enforced through diagrammatic inclusion). The OR-states in Figure 1 are *ThrustLimitation, LockDetect, AwaitLock* and *OnGroundDetect*. AND-states are used to model concurrency and allow sets of transitions to be simultaneously enabled. Whenever an AND-state is active, all of its immediate sub-states are active. *InadvertentDeploy* is the only AND-state in the example and its sub-states (*LockDetect* and *OnGroundDetect*) are partitioned using the dotted line. A basic state is a state with no sub-states. The basic states of Figure 1 are *Idle, Unstowed, Wait, Locked, InAir* and *OnGround*.

At any point in time, a set of states in the Statechart is active, known as a configuration. The set of valid configurations (VC) is restricted by the semantic definition of OR and AND-states. Control is passed between configurations via transitions. Each transition is labeled with an event expression, a guarding condition and an action, each of which is optional. Event expressions consist of predicates defining the presence or absence of internal or external events. Guarding conditions are formed by predicates over input parameters and internal variables, which in the subset discussed here may take boolean, integer or enumerated types. Actions consist of a combination of event broadcast expressions and assignments to output parameters and internal variables. If more than one action is associated with a transition, all actions are considered to be taken simultaneously. The syntax used for forming transition labels is as follows:

EventExpression [GuardCondition]/Actions

The semantics of a label can be described as follows:

$$EventExpr(I, V) \wedge GuardCond(I, V) \Rightarrow Action(I, V, V', O)$$

where I, V, V', O represent possible combinations of the individual input parameters, internal variables, updated internal variables and output parameters respectively.[1] $EventExpr$ returns $true$ if the currently active input or internal events satisfy the event expression. $GuardCond$ returns $true$ if the present values of the input parameters and internal variables satisfy the guard condition. $Action$ defines the relation between the input parameters, internal variables, updated internal variables and the output parameters as defined in the transition actions. Each OR-state has a default transition used to set the initially active sub-state on entry to the OR-state (unless the entering transition explicitly terminates at a sub-state). The default transitions of Figure 1 are *T1, T4, T7* and *T9*.

The computation that is performed by the Statechart is represented by a series of *statuses* and a trace of input and output parameters. A status consists of a set of currently active states, the set of internal events generated in the last computation step and the values of all internal variables. The synchronous time model was chosen for the subset and specifies that the system executes a single step every time unit, reacting to changes made in the status or to the parameters since the last step. The changes to the status caused by the transitions enabled by the current inputs are processed in the next step along with the values of any input parameters that may have been altered during the execution of the step.

The Statechart semantics impose constraints on the behaviour which are left implicit in the diagrams. However, in order to mechanically derive accurate and complete tests using generic test automation techniques, this behaviour must be made explicit in any formal notation on which the test generation techniques will be based. Generic tools can then be used that do not require an understanding of the semantics of the notation from which the formal specification was generated. For the subset of Statecharts under consideration, this implicit behaviour occurs in the following four situations.

[1]For example, I is defined as $(I_1 \times I_2 \times \ldots \times I_n)$, where $I_1, \ldots I_n$ represent the types of each of the input parameters.

- **Transitions terminating at a non-basic state.** If a transition terminates at a non-basic state, the transition is composed with the default transition of the target state. The composition is continued until a default transition terminates at a basic state. Example transitions of this kind in Figure 1 are *T2* and *T5*.

- **Transitions originating from non-basic states.** If a transition originating at a non-basic state is active, then it must be possible to take the transition if any one of the sub-states of its source state is enabled. The transition is partitioned such that each newly created transition originates at a distinct basic state that is a descendent of the original transition's source state. *T6* is the only transition of this kind in Figure 1.

- **Priority resolution mechanism.** If two conflicting transitions are enabled simultaneously, the transition with the higher scope is given priority. The scope of a transition is defined as the lowest common OR-state that is an ancestor of all its source and target states. The guard of the lower priority transition is updated by conjoining with it the negation of the guard of the higher priority transition. In the example, transition *T3* would take priority over transitions *T6* and *T10*.

- **Completeness assumption.** The semantics of Statecharts specify that if a state is active and no transitions are enabled in a step, events generated in the previous step are consumed and the active state is maintained. Additional transitions are added to explicitly define this behaviour.

Once the above semantic issues have been resolved, the resulting transitions are translated into Z schemas using the following template:

$$\begin{array}{|l}
\hline
\textit{StepTemplate} \\
\Delta \textit{Status} \\
\textit{Parameters} \\
\hline
\textit{SourceConfiguration} \subseteq \textit{ActiveStates} \\
\textit{EventExpression} \\
\textit{GuardingCondition} \\
\textit{Actions} \\
\textit{TargetConfiguration} \subseteq \textit{ActiveStates}' \\
\hline
\end{array}$$

Status is a schema used to store the currently active states (configuration), internal events generated in the last step and the values of internal variables. *Status* also contains an invariant that ensures the set of active states conform with the semantics of the Statechart. *Parameters* is a schema defining the input and output parameters (as events or data-items). *EventExpression*, *GuardingCondition* and *Actions* are derived from the transition label elements, updated appropriately to resolve transition priorities. $\Delta \textit{Status}$ is used to specify that the contents of the status are updated by the operation and the " ' " decoration is used to refer to the updated values of the status. Based on this template, the transition *T3* would be specified as in Figure 2.[2] Feasible combinations of concurrently active transitions are represented by sets of transitions whose Z representations, when conjoined, do not conflict.

The automatic generation of the Z specification was acheived using an Application Programming Interface (API) to the STATEMATE tool. A program was written that extracts a model of the Statechart, performs the semantic resolution steps described above and writes out the full Z specification including operations based on the step template. Several other variants of Statecharts

[2]where ? and ! are standard Z convention for labeling inputs and outputs respectively.

$$\begin{array}{|l}
\multicolumn{1}{l}{T3} \\
\Delta Status \\
Parameters \\
\hline
\{ThrustLimitation, InadvertentDeploy, \\
LockDetect, OnGroundDetect, AwaitLock, \\
Locked\} \subseteq ActiveStates \\
ThrottleReq? \leq FwdIdle \\
RestowComplete! = Present \\
LimitThrust! = False \\
\{ThrustLimitation, Idle\} \subseteq ActiveStates' \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{T3BVA1} \\
\Delta Status \\
Parameters \\
\hline
\{ThrustLimitation, InadvertentDeploy, \\
LockDetect, OnGroundDetect, AwaitLock, \\
Locked\} \subseteq ActiveStates \\
\boxed{ThrottleReq? < FwdIdle - 1} \\
ThrottleReq? \leq FwdIdle \\
RestowComplete! = Present \\
LimitThrust! = False \\
\{ThrustLimitation, Idle\} \subseteq ActiveStates' \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{T3BVA2} \\
\Delta Status \\
Parameters \\
\hline
\{ThrustLimitation, InadvertentDeploy, \\
LockDetect, OnGroundDetect, AwaitLock, \\
Locked\} \subseteq ActiveStates \\
\boxed{ThrottleReq? = FwdIdle - 1} \\
ThrottleReq? \leq FwdIdle \\
RestowComplete! = Present \\
LimitThrust! = False \\
\{ThrustLimitation, Idle\} \subseteq ActiveStates' \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{T3BVA3} \\
\Delta Status \\
Parameters \\
\hline
\{ThrustLimitation, InadvertentDeploy, \\
LockDetect, OnGroundDetect, AwaitLock, \\
Locked\} \subseteq ActiveStates \\
\boxed{ThrottleReq? = FwdIdle} \\
ThrottleReq? \leq FwdIdle \\
RestowComplete! = Present \\
LimitThrust! = False \\
\{ThrustLimitation, Idle\} \subseteq ActiveStates' \\
\hline
\end{array}$$

Figure 2: Z specification for transition T3 and boundary value analysis partitions

(MATLAB/Stateflow [31], PFS Statecharts [11]) have been formalised in a similar way, as well as a tabular notation (PFS tables [11]) used for specifying the reactive components of aerospace control software. The same subset of Z was used to represent all of these notations. This allowed the test generation toolset described in Section 4 to be re-used for all of the source notations. Predicates in the Z subset consist of combinations of logical, relational and arithmetic operators and the $\subseteq$ operator used to constrain state configurations. The following sections now describe techniques that are not particular to Statecharts but have been developed for the subset of Z used to model the above mentioned set of notations.

3 Formalising testing heuristics

3.1 Equivalence class testing

Testing techniques that have been developed for model-based notations such as Z (e.g. [1, 10, 30, 17, 29]) are typically based on the principle of *equivalence classes* [12]. Equivalence classes are formed by partitioning the input domain into sets of data that, for testing purposes, exhibit the same behaviour. Based on the *uniformity hypothesis* only one test point need be selected from each equivalence class to verify the implementation against the specification. In the methodology described here, equivalence classes are identified using *testing heuristics*. A partitioning heuristic partitions the specification into equivalence classes that completely cover the input space of the specification and can be based on the signature (variable type declarations) or predicate part of the specification. Selecting data from each subdomain is then assumed to reveal a certain class of faults in the implementation. A fault-based heuristic identifies an equivalence class in the specification which contains those values that reveal a particular hypothesised fault.

Automatically applying testing heuristics to the specification requires a method of describing the heuristics. In a high integrity process it should also be possible to demonstrate that the resulting test cases accurately represent the testing heuristic and maintain conformance with the original specification. It may also be desirable to compare the various fault-finding abilities of different heuristics (e.g. partitioning vs. fault-based heuristics) in order to optimise the set of heuristics that must be applied to each specification for a particular fault coverage. Formally specifying the heuristics allows all of these issues to be addressed.

3.2 Partitioning heurisitcs

Partitioning heuristics can be formally specified as follows, where the input space defined by predicate P is partitioned into the subdomains $P_1, ..., P_n$ and $Vars(P)$ represents the declarations of all variables in the predicate to be partitioned:

$$\forall\ Vars(P) \bullet P \Leftrightarrow P_1 \lor ... \lor P_n$$

A boundary value analysis partitioning heuristic based on the integer $\leq$ operator could therefore be defined as follows:

$$\forall\ A, B : \mathbb{Z} \bullet A \leq B \Leftrightarrow (A < B - 1) \lor (A = B - 1) \lor (A = B)$$

Applying this heuristic to the Z schema representation of transition *T3* would result in the three test cases shown in Figure 2, where the equivalence class defining each test case is highlighted and the operands to the partitioned predicate ($ThrottleReq?$ and $FwdIdle$) have been substituted for the generic values A and B in the testing heuristic. If a partitioning heuristic can be shown, through proof, to be complete and result in disjoint sub-domains, test cases based on this template will inherit these properties. An automated method of applying the heuristics that guarantees that these properties is described in Section 4.1.

3.3 Fault-based heuristics

Fault-based (or mutation) testing can be an effective means of assessing other testing strategies, such as partition testing. A test set can be evaluated against a number of hypothesised faults that are considered representative of a large proportion of the actual faults likely to appear in the implementation. The effectiveness of the target test set at detecting these faults is then used as a measure of test effectiveness or completeness. Performing this evaluation using mutations of the specification allows such an analysis to be performed much earlier in the life-cycle as it does not rely on the pre-existence of an implementation. Fault-based testing can also be an effective means of generating the test set itself based on the intuition that if a test is adequate to detect a slight variation in the correct behaviour, that part of the system is being exercised adequately [2]. Due to the large number of potential mutations, this approach may lead to an infeasibly large number of test cases. However, selective (program-based) mutation testing studies [23, 22, 25] suggest that a relatively small number of mutations can lead to the detection of a large class of faults. Automation may allow significant examination of similar results applied to specification-based mutants.

If a fault in the implementation can reveal itself as a mutation of the sub-expression E to E' in the specification, then the *necessary condition* for detecting the fault can be generically described as follows (based on a definition by Kuhn [24]):

$$\exists\ Vars(E);\ \ Vars(E') \bullet E \neq E'$$

Informally, there exists a set of values for the variables in E and E' such that E and E' evaluate differently. Selecting values from this set will ensure that the fault reveals itself in the evaluation of the sub-expression. By allowing the sets $Vars(E)$ and $Vars(E')$ to differ, the class of faults covered by variable replacement can also be modelled. If no values can be found to satisfy the necessary condition, the fault does not reveal itself and can therefore be classed as an equivalent mutant. As an example, the following heuristic defines the fault condition for a variable replacement mutation applied to an addition expression:

$$\exists\ A, B, C : \mathbb{Z} \bullet A + B \neq A + C$$

This heuristic can be instantiated with variables from Figure 1 to detect a mutation of transition *T11* where *Timer* is incremented by itself rather than 1. The instantiation would result in the following fault detection condition:

$$\exists\ Timer : \mathbb{Z} \bullet\ Timer + 1 \neq\ Timer +\ Timer$$

Any number of such generic fault-based heuristics can be defined, limited only by the subset of the notation used to specify the system under test. A hypothesised fault can be shown to be *weakly detectable* if the formalisation of the necessary condition can be shown to be a valid premise. That is, some values can be found that reduce the expression to true. The fault is said to be weakly detectable because there is no guarantee that other expressions in the system will not mask the fault at the outputs. However, the probability of detecting the fault is strongly increased if the necessary condition is satisfied. The manner in which faults in the implementation reveal themselves as mutations of the specification will depend on the refinement (between specification and code). Examination of this relationship may therefore point towards more testable implementations of the specifications. Fault-based heuristics can also be used to generate the abstract test cases in a similar fashion to partitioning heuristics. The parameters are instantiated with the operands of the expression containing the hypothesised fault. The resulting predicate then defines the equivalence class of the test.

The formulation of necessary fault detection conditions can be used to assess the fault detection ability of various partition and fault-based heuristics and therefore be used to identify an efficient set of heuristics for a particular subset of Z (based on the possible mutations of specifications written in that subset). For example, a partitioning heuristic that results in the subdomains $(P_1, ..., P_n)$ weakly detects the mutation of E to E' if:

$$(P_1 \Rightarrow \neg(E \Leftrightarrow E')) \vee ... \vee (P_n \Rightarrow \neg(E \Leftrightarrow E'))$$

The results of such an analysis applied to mutants of $A \vee B$ and the partitioning strategy $A \vee B \Leftrightarrow (A \wedge B) \vee (A \wedge \neg B) \vee (\neg A \wedge B)$ are summarised in Table 1. The analysis demonstrates that the partitioning strategy is not adequate to detect certain mutations that include additional or replaced expressions (C).

In order to guarantee detection of a fault in situations where monitoring the evaluation of all sub-expressions in the implementation is not possible, the *sufficient condition* must be constructed. Whereas the necessary condition restricts the range of values of the variables to those that distinguish between the original and mutated expression, the sufficient condition must restrict the values of the inputs to those that can distinguish between the original and mutated relation between the

Mutated form of $A \vee B$	**Detected by**
A	$\neg A \wedge B$
B	$A \wedge \neg B$
$A \wedge B$	$A \wedge \neg B, \neg A \wedge B$
A *XOR* B	$A \wedge B$
$\neg(A \vee B)$	$A \wedge B, \neg A \wedge B, A \wedge \neg B$
$A \vee C$	Not necessarily detected
$C \vee B$	Not necessarily detected
$A \vee B \vee C$	Not necessarily detected (never by this heuristic)
$A \vee B \wedge C$	Not necessarily detected

Table 1: Effectiveness of the disjunction partitioning heuristic

controllable inputs and observable outputs. The sufficient condition is constructed using the following pattern:

$$\exists\, Inputs\_Outputs(P) \bullet \neg(P \Leftrightarrow P')$$

where P denotes the original predicate part of the operation and P' denotes the predicate part of the schema with the mutation applied and $Inputs\_Outputs(P)$ represents the controllable inputs and observable outputs of P. If the sufficient condition can be satisfied, the fault is said to be *strongly detectable*.

The number of possible solutions to the sufficient conditions can be used as a measure of how testable an implementation is against faults that reveal themselves as mutations in the specification. The larger the number of solutions to the sufficient conditions, the higher the probability that one of these solutions would be selected during standard (e.g. random or partition) testing procedures. The construction of sufficient test data may involve solving greatly more complicated constraints than for necessary conditions. Therefore in some cases, the use of necessary conditions may be preferred as a trade-off between confidence in the test results and the effort required to generate the test data.

The formulation of sufficient conditions for state-based systems is complicated by the fact that updated internal variables are typically not referenced until a later operation. Under these circumstances, the sufficient condition as presented above may not be satisfiable. Testing under these conditions involves finding *sequences* of operations that are sufficient to detect mutations of the system state. This is the implicit assumption behind many of the the FSM-based test sequence generation techniques such as the W-Method [7] and the UIO method [28]. However, for Extended Finite State Machine (EFSM) based notations such as Statecharts, the system state may consist of a combination of state (in the classic FSM definition of the term) and internal variables and events. A fault in the system state can be said to be *strongly detectable in n steps* if the sufficient condition for detecting the fault requires a sequence of no more than n operations.

4 Automatically generating tests

This section describes how a combination of techniques have been used to generate test cases and sequences from the subset of Z used to model the specifications of interest. The generation procedure follows three steps. Partitioning and fault-based test cases are first generated from the Z specification using an automated theorem prover. If the specification includes a persistent state and faults in the state are only strongly detectable in n-steps, an abstract finite state machine is constructed from the test cases and used to derive checking sequences for equivalence classes of the system state that are not directly observable at the testing interface. A combination of constraint solvers is then used to populate the test cases and sequences with test data.

4.1 Applying testing heuristics

The formalised testing heuristics are automatically applied to the specification using the general purpose theorem prover CADiℤ. CADiℤ [32, 34] is a Z type checker and theorem prover that allows a user to interactively browse, type check and perform proofs upon a Z specification. CADiℤ also allows proof tactics to be written in a lazy functional notation [33]. The tactic language includes parameterisation and pattern matching facilities to enable generic tactics to be written for implementing particular proof *patterns*. When such a tactic is applied, the parameters are instantiated with the currently selected parts of the specification (or typed input from the user). Generic proof tactics were used to automate the generation of test cases based on the formal specification of the testing heuristic.

Testing heuristics are specified as named “lemmas” and stored in a separate Z file that is included within the scope of any specifications being tested. Completeness, disjointness and satisfiability properties of these heuristics can be proven using the CADiℤ interactive proof mode. The corresponding proofs can be recorded as proof tactics to be replayed at a later date (for example, by a new user who is as yet unconvinced of the heuristics’ validity). A proof tactic was written that, given an expression in the specification and the name of a testing heuristic, instantiates the generic parameters of the chosen heuristic with the operands of the expression and conjoins the result with the predicate part of the operation under test (this is a valid proof step if the heuristic has been previously proven to be a tautology). The result is then simplified to produce a new Z schema for each test case. Testing heuristics can be repeatedly applied in order to generate a hierarchy of test cases for an operation based on a number of different testing hypotheses. If all (hypothesised) faults are strongly detectable in 1 step, these test cases can be used to verify the implementation against the set of testing hypotheses. Otherwise, the test cases must be sequenced to detect faults in the system state.

4.2 Test sequences

4.2.1 Constructing the abstract finite state machine

Dick and Faivre [10] proposed the construction of an abstract finite state machine (AFSM) as a means of sequencing test cases derived from and specified using the VDM-SL notation [21]. Murray et al. [26] have since shown how the technique can be applied to Z specifications. These techniques are extended here to include abstraction of the inputs and outputs and to apply standard FSM-based test sequence generation techniques to the resulting AFSM. States in the AFSM correspond to the equivalence classes to be tested in the system state, identified using partitioning and fault-based heuristics, as described earlier in this paper. The operations in the specification are used to calculate the transitions between the abstract states by defining the transformation of the system state from one equivalence class to another. Particular equivalence classes in the system state can be indirectly tested by finding sequences through the AFSM that cover and check the abstract states corresponding to the equivalence classes of interest.

Assuming that the elements of *Status* are not controllable or observable in the testing environment, the abstract states for the specification described in Section 2 would be based on the pre and postconditions in the test cases projected on to the elements of *Status*. The abstract states are calculated by finding the minimal partition of these predicates. For any overlapping predicates, the minimal partition is calculated using the following partitioning strategy [10] (automated using the method described above):

$$A \lor B \Leftrightarrow (A \land B) \lor (\neg A \land B) \lor (A \land \neg B)$$

As an example, the post-condition of transition $T5$ from Figure 1 (conjoined with $T7$ during the translation to Z) would result in the following equivalence class for $Status$:

$$\begin{array}{|l}
\multicolumn{1}{l}{AbstractState_n} \\
Status \\
\hline
\{ThrustLimitation, InadvertentDeploy, LockDetect, \\
OnGroundDetect, AwaitLock, Locked\} \subseteq ActiveStates \wedge Timer = 0 \\
\hline
\end{array}$$

A similar abstraction is applied to each of the input and output parameters resulting in the set of disjoint combinations of the equivalence classes of the parameters. The AFSM is specified in terms of the abstract states, inputs and outputs using the following tuple and definitions. An AFSM M is represented by the tuple $(S, s_0, X, Y, \delta, \lambda)$ where:

- S is the set of abstract states, where s_0 is the initial state derived from the post-condition of the initialisation operation,
- X is the input language (minimal partition of the input predicates),
- Y is the output language (minimal partition of the output predicates),
- δ is the non-deterministic state transfer relation, $X \times S \leftrightarrow S$,
- and λ is the non-deterministic output relation, $X \times S \leftrightarrow Y$.

The elements of the δ relation are calculated by checking, for each combination of states S_i and S_j (including equivalent states) and input X_i, whether there exists an operation Op such that the following predicate is true:

$$X_i \wedge S_i \wedge S_j' \wedge Op$$

Similarly, the elements of the λ function are calculated by checking, for each combination of state S_i, input X_i and output Y_i, whether there exists an operation Op such that the following predicate is true:

$$X_i \wedge S_i \wedge Y_i \wedge Op$$

Two possible approaches to modeling concurrency in the models were considered. The simple abstraction described above results in forming the product machine of any orthogonal components (such as *InadvertentDeploy* in Figure 1). However, this can lead to state explosion and greatly reduces the efficiency of the test sequence generation techniques. Instead, concurrency is modelled using a system of communicating finite state machines. If references to shared state can be assumed to be restricted to a single writer, multiple reader relationship, the communications between the components can be represented as references to the other component's abstract state (constraints) in the pre-condition of the transitions. The definitions of δ and λ are updated as follows:

- $\delta = X \times S \times C_1 \times ... \times C_n \leftrightarrow S$
- $\lambda = X \times S \times C_1 \times ... \times C_n \leftrightarrow Y$

for n orthogonal components where C_j represents the set of abstract states for each AFSM j $(i \neq j)$ that are referenced in the pre-condition of the transition. Testing the system based on the set of testing hypotheses used in selecting the partitioning and fault-based heuristics reduces to covering each abstract state and transition. In order to ensure strong detectability of faults in the system state, the value of the abstract state must be checked after the execution of each transition.

The derivation of the AFSM may introduce non-determinism. However, this will typically only apply to a subset of the state variables that are used to form the abstract state. Several approaches were explored for resolving this non-determinism, including the use of a model checker (see below) for generating concrete feasible sequences based on those derived from the non-deterministic AFSM.

4.2.2 Generating test sequences

State checking sequences can be generated from the AFSMs using traditional FSM-based techniques [7, 28, 36]. The authors chose to use the Unique Input/Output (UIO) sequence algorithm [28]. The algorithm attempts to find, for each state, a sequence of inputs and outputs that is unique to that state. Such a sequence can then be used to distinguish that state from all others. UIO sequences are generated for each AFSM separately. A sequence is said to be *output distinguishable* from another if the sequences produce different traces of outputs. The UIO sequence generation algorithm is applied to each state s_i in turn and can be summarised as follows:

1. Set the length (l) of the current sequence to 1.

2. For all sequences of length l originating at state s_i check for output distinguishability against all sequences that can be triggered by the same trace of inputs and constraints originating at all states s_j, where $s_j \neq s_i$. If the sequence is output distinguishable against all these sequences, it is the UIO sequence for state s_i.

3. If no output distinguishing sequence is found of length l, increase l by 1 and repeat steps 2-3.

The use of shared outputs (more than one orthogonal component writes to the same output) complicates the calculation of output distinguishability as the output may have been produced by the transition under examination or an orthogonal component in the system. Such shared outputs could be removed from the output abstraction and therefore not used in the calculation of distinguishable transitions. However, this may result in no UIO sequences being generated for certain components that wrote to only shared outputs. Therefore, the outputs are still used to generate the UIO sequences and the SMV model checker (see below) was used to validate whether other components could update the shared output at the same point in the sequence.

Model checking is a technique for verifying properties of state-based systems. A computation tree of a model of the system is enumerated and checked against a specification of the properties to be verified. If the model violates a property, a counter-example showing a sequence that leads to a state that does not satisfy the property is generated. The ability to generate counter-examples makes model-checking a convenient means of generating test sequences from finite state machines. The finite state machine is defined as the model and the negation of the path constraint for the desired test sequence is defined as a liveness property to be checked against the model [6]. The model checker will then attempt to find a counter-example to this property that corresponds to a test sequence satisfying the path constraint. Model checking also has the advantage that tests according to different criteria can be generated by simply varying the property to be checked against the model. This results in a similar level of flexibility as the pattern-based operation testing techniques described earlier.

The model checker SMV [3] was used to explore the possiblities of model checking-based test sequence generation. Each AFSM is specified as a SMV module. A module consists of a number of variables which are used to represent the inputs, outputs and state variables of the AFSM. The δ and λ functions are implemented as a pair of case statements over pairings of S and X. Each case statement defines the next value of S and Y respectively. The specification property places constraints on the abstract states, inputs and outputs. Test sequences can be generated by finding counter-examples to the negated specification of the desired properties of the sequence. This approach to test sequence generation was found useful in generating initialisation sequences, validating UIO sequences (particularly dependencies between orthogonal components that may make the sequence infeasible) and to generate sequences based on a partial specification of the ordering of certain inputs, outputs or states. The SMV model was automatically generated based on the same XML (eXtensible Markup Language) [8] specification of the AFSM as was used to generate the UIO sequences (where the minimal partition of the inputs and outputs was calculated automatically). The generation of the XML specification from the Z specification is yet to be automated but is considered feasible given an extension of the current tool set. XML is becoming increasingly popular as a common interchange format for different modelling tools. Future work aims to capitalise on the standardisation of these interchange formats to extend the applicability of the toolset described here to a wider range of commercial modelling tools and notations.

4.3 Constraint solvers

A number of constraint solvers were used to generate the test data. The CADi$\mathbb{Z}$ built in constraint solvers (a linear constraint solver, simulated annealing optimisation-based constraint solver and the SMV model checker) were used wherever possible. In unit testing situations or for reactive components where no test sequences were needed, the test data was generated directly from the test case descriptions. This was achieved by writing a proof tactic that automatically selects the constraint solver applicable to the particular constraint being solved. Using an API to CADi$\mathbb{Z}$ the process of generating test data for unit tests was automated by writing a program to apply the test data generation tactic to each of the test case specifications in a particular Z file and write the results out as an AdaTEST [20] test script. After some transformation of the resulting script to incorporate refinement, the script could then be run against the programs under test. In some cases a non-linear constraint solver was required, for these predicates, the stand-alone constraint solver *Lingo* [19] was found to be very effective.

The test sequence generation techniques result in sequences of constraints over the inputs and outputs described as sequences of abstract inputs and outputs. Where the concrete values were not trivial to calculate, the same set of constraint solvers as described above were used to generate the test data.

5 Results

The techniques described in this paper were evaluated as part of a case study to develop an EEC software application using model-based specifications. Z specifications were automatically generated from approximately 100 pages of Statecharts, reactive tabular requirements and supporting text. These were then used as the basis for automatically validating the completeness and determinism of the specifications (described in more detail in [5]). Tests were then generated from these validated specifications. Objectives of the case study included evaluating the efficiency of the automated test generation, assessing the testability of the specifications and evaluating the effectiveness of standard testing heuristics.

The case study specifications consisted of 9 Statecharts (48 states, 112 transitions) and 34 reactive specification tables. These were translated into 146 Z operation schemas from which 499 test cases were generated using boundary value analysis and disjunction partitioning heuristics. These test cases were automatically populated with test data and converted into AdaTEST [20] test scripts to be run against the implementation as part of unit testing. This process was almost fully automated and the large majority of the effort involved selecting the testing heuristics to apply to each Z schema (a process that also has the potential for automation). The automatic generation of test sequences (for integration testing) was not as fully automated as the production of unit tests and the XML specification of the AFSMs was produced by hand based from the Z specification of the test cases. However, once this XML specification was written, the generation of UIO sequences and the SMV model was fully automatic.

For many of the Statecharts studied, no UIO sequences were found for at least one abstract state. The length and feasibility of the test sequences were found to be a good measure of the testability of the specifications. Factors that affected the testability of the Statecharts are thought to be a combination of ratios of the inputs and states to the outputs, corresponding to the domain to range ratio of the δ and λ functions of the AFSMs. The higher the ratios, the longer the sequences tended to be and the greater the probability that no sequences were found. The Statecharts used in the case study were not designed with testability in mind and the information was useful in defining guidelines for future Statechart specifications and also justifying the expense of current unit testing practices. Due to these limitations, integration testing was mainly limited to scenario style testing using sequences generated from the SMV model and use cases derived from the high level requirements of the system. Work is continuing to investigate the factors affecting the testability of the specifications.

Part of the code was subjected to mutation analysis as a means of assessing the effectiveness of the partitioning heuristics used. The mutation adequacy of the test set ranged from 82.98% to 91.3% based on different versions of the program. Two versions of the program were written by hand using alternative refinement patterns and three versions of the program were automatically generated from the Z specification with varying levels of optimisation using an experimental tool developed by other members of the research group. The fault detectability of the partitioning heuristics was also evaluated based on the formalised partitioning heuristics and fault detection conditions (as described in Section 3.3). This analysis confirmed the empirical results and suggests that traditional decision coverage and MCDC coverage approaches to unit testing should be augmented with additional tests for better fault coverage.

6 Conclusions and further work

This paper has described how tests can be automatically generated from Statechart specifications using an intermediate formal representation and a formal approach to specifying testing heuristics. The toolset is applicable to a number of graphical and tabular notations due to the common intermediate formal notation and makes use of theorem provers and constraint solvers to generate the tests. The use of graphical and tabular notations is intended as a means of bridging the gap between specialist domain engineers and formal methods. Therefore, future work will investigate how the automated Z-based testing techniques described in this paper can be presented to the engineers in a way such that the flexibility (in terms of testing heuristics) of the methods are maintained, while not requiring the engineers to work directly with the Z representation of the specifications. This work may include the specification, and subsequent automatic translation to Z, of testing heuristics in similarly intuitive notations to Statecharts and the presentation of the results of the test generation process using similar notations (e.g. test sequences could be presented as message sequence charts).

Formalising both the specifications and the testing heuristics facilitated a high degree of automation and a rigorous method of analysing the testability of the specifications and the comparative fault finding abilities of different testing heuristics. Section 5 discussed how the techniques were applied to an industrially relevant case study. The techniques resulted in not only the large scale automation of test cases and (to a lesser extent) sequences but also the identification of factors that affect the testability of Statechart specifications and highlighted the shortcomings of several commonly used testing heuristics. The formal definition of the the testing heuristics and fault detection conditions also allowed this latter analysis to be performed formally as well as empirically. Ongoing work is looking to generate conditions that will enhance the range of faults detected by these heuristics and investigate in more detail the factors affecting the testability of Statecharts.

UIO sequences can only be used to detect operation and state transfer faults. The detection of additional states in the implementation cannot be gauranteed. Therefore as part of future work, alternative approaches, such as the W-method should be investigated. Furthermore, a better understanding of how faults in the implementation can reveal themselves as mutations of specification and the relationship of this mapping to the method of refinement into code will provide a more informed choice as to which testing methods are most suitable for a particular application.

Future work also aims to better integrate the different areas of automation, in particular, extending the range of integrated constraint solvers to handle non-linear constraints and automating the generation of the abstract finite state machine from the formal specification of the test cases. The applicability of the techniques to a wider range of graphical notations (such as SDL and UML) and other application domains is also under consideration.

7 Acknowledgements

This work was funded by the High Integrity Systems and Software Centre, Rolls-Royce Plc.

References

[1] Nina Amla and Paul Ammann. Using Z specifications in category partition testing. *Proceeding of COMPASS 1992, Seventh Annual Conference On Computer Assurance*, pages 3–10, 1992.

[2] P. Ammann, P. Black, and W. Majurski. Using model checking to generate tests from specications. December 1998.

[3] Sergey Berezin. The SMV web site. http://www.cs.cmu.edu/~modelcheck/smv.html/, 2000. The latest version of SMV and its documentation may be downloaded from this site.

[4] K. Bogdanov, M. Holcombe, and H. Singh. Automated test set generation for statecharts. In *Proceedings International Workshop on Current Trends in Applied Formal Methods*, 1998.

[5] Simon Burton, John Clark, Andy Galloway, and John McDermid. Automated V&V for high integrity systems, a targeted formal methods approach. In *Proceedings of the 5th NASA Langley Formal Methods Workshop*, June 2000.

[6] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using model checking. In *Proceedings 1996 SPIN Workshop*, August 1996. Also WVU Technical Report NASA-IVV-96-022.

[7] Tsun S Chow. Testing software design modeled by finite-state machines. *IEEE Transactions On Software Engineering*, SE-4(3):178–187, May 1978.

[8] World Wide Web Consortium. eXtensible Markup Language. http://w3c.org/XML/.

[9] Rational Cooperation. Unified modeling language. Rational Software Cooperation. Unified Modeling Language. Available at http://www.rational.com, 1999.

[10] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. *FME'93:Industrial Strength Formal Methods. LCNS 670*, pages 268–284, April 1993.

[11] Andy Galloway, Trevor Cockram, and John McDermid. Experiences with the application of discrete formal methods to the development of engine control software. *Proceedings of DCCS '98. IFAC*, 1998.

[12] John B. Goodenough and Susan L. Gerhart. Towards a theory of test data selection. *IEEE Transactions On Software Engineering*, 1(2):156–173, June 1975.

[13] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[14] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, , Michael Politi, Rivi Sherman, Aharon Shtull-Truaring, and Mark Trakhenbrot. Statemate, a working environment for the development of complex reactive systems. *IEEE Transactions On Software Engineering*, 16:403–414, 1988.

[15] David Harel and Amnon Naamad. The Statemate semantics of statecharts. *IEEE Transactions On Software Engineering And Methodology*, 5(4):293–33, Oct 1996.

[16] Kathryn L. Heninger. Specifying software requirements for complex systems: New techniques and their applications. *IEEE Transactions on Software Engineering*, 6(1), 1980.

[17] R. M. Hierons. Testing from a finite state machine: Extending invertibility to sequences. *The Computer Journal*, 40(4):220–230, 1997.

[18] Hyong Seok Hong, Young Gon Him, Sun Deok Cha, Doo Hwan Bae, and Hasan Ural. A test sequence selection method for statecharts. *Software testing, verification and reliability*, 10:203–227, 2000.

[19] LINDO Systems Inc. Lingo. http://www.lindo.com, 2001.

[20] Information Processing Ltd., Bath, UK. *AdaTEST 95 User Manual*, June 1997.

[21] The International Organization for Standardization. *Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language. International Standard ISO/IEC 13817-1*, December 1996.

[22] J.Offut, A.Lee, G. Rothermel, R.H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on software engineering methodology*, 5(2):99–118, April 1996.

[23] J.Offut, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. *Proceedings of the 15th International Conference on Software Engineering*, pages 100–107, May 1993.

[24] D. Richard Kuhn. Fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering and Methodology*, 8(4):411–424, October 1999.

[25] E.S. Mresa and L. Bottaci. Efficiency of mutation operators and selective mutation strategies: an empirical study. *The Journal of Software Testing, Verification and Reliability*, 9(4):205–232, December 1999.

[26] L. Murray, D. Carrington, I. MacColl, J. McDonald, and P. Strooper. Formal derivation of finite state machines for class testing. Technical Report 98-03, Software Verification Research Centre, School of Information Technology, The University of Queensland, Brisbane 4072, Australia, February 1998.

[27] Martyn Ould. Testing - a challenge to method and tool developers. *Software Engineering Journal*, 39:59–64, March 1991.

[28] Krishan Sabnani and Anton Dahbura. A protocol test generation procedure. *Computer Networks And ISDN Systems*, 15:285–297, 1988.

[29] Harbhajan Singh, Mirko Conrad, Gottfried Egger, and Sadegh Sadeghipour. Test case design based on Z and the classification-tree method. *First IEEE International Conference on Formal Engineering Methods*, November 1997.

[30] Phil Stocks and David Carrington. Test template framework: A specification-based case study. *Proceedings Of The International Symposium On Software Testing And Analysis (ISSTA'93)*, pages 11–18, 1993.

[31] The MathWorks. Stateflow. http://www.mathworks.com/products/stateflow/.

[32] Ian Toyn. Formal reasoning in the Z notation using CADi$\mathbb{Z}$. *2nd International Workshop on User Interface Design for Theorem Proving Systems*, July 1996.

[33] Ian Toyn. A tactic language for reasoning about Z specifications. In *Proceedings of the Third Northern Formal Methods Workshop, Ilkley, UK*, September 1998.

[34] Ian Toyn. The CADi$\mathbb{Z}$ web site. http://www.cs.york.ac.uk/~ian/cadiz/, 2000. The latest version of CADi$\mathbb{Z}$ and its documentation may be downloaded from this site.

[35] International Specifications Union. Itu-t recommendations Z.100, specification and description language, 1994.

[36] Hasan Ural. Formal methods for test sequence generation. *Computer Communications*, 15(5):311–325, June 1992.

Classical search strategies for test case generation with Constraint Logic Programming[*]

Alexander Pretschner
Institut für Informatik, Technische Universität München, Germany
www4.in.tum.de/~pretschn

Abstract

Test case generation for concurrent reactive systems on the grounds of symbolic execution basically amounts to searching their state space. As in the case of model checkers, different search strategies (depth-first, breadth-first, best-first, tabu) together with different strategies for storing visited states have a significant impact on the performance of the generation algorithm. We present experimental data for the performance of different search strategies and discuss the results, taking into account counter examples as generated by model checkers.
Keywords. CASE, Model Checking, State Representations, Symbolic Execution.

1 Introduction

Usually, errors in early software development design phases result in disproportionately high costs if they have to be corrected. This is due to the increasing number of implications of each design decision: the earlier a decision is made, the more implications it involves.

One commonly accepted solution to this problem is an incremental approach to software/ hardware development. The idea is to get possibly executable pieces of software (or models of hardware) in early phases that can be used for validation by interacting with a respective customer, and for verification w.r.t. given specifications (properties).

The focus of this paper is on reactive (embedded) systems, and we thus concentrate on behavior models. In order to get executable behavior models of such a system, a suitable high-level, preferably graphical, specification formalism is needed. Finite state machines have been identified as a practically usable candidate. With adequate CASE tools, systems can be built, simulated, and verified. Simulation not only helps the developer in understanding the system and detecting errors, but can also be used for customer interaction (validation). Verification and validation are usually achieved by testing, e.g., checking if, given a certain input, the system's output corresponds to the desired one.

Testing is, however, incomplete. Besides semi-automatic proof systems, model checkers aim at *guaranteeing* that the system satisfies a certain property. Basically, they exhaustively search the system's state space. This necessarily requires finite state spaces (e.g., SMV or SPIN) or built-in abstractions (e.g., Uppaal or HyTech for a restricted class of hybrid systems). The problem with state space explosion is obvious.

Along side these practical considerations, the concept of model checkers is such that they focus on verifying properties of models. Not only in the area of embedded systems the system model is just one step in the development process: the system has to be implemented after the specification (or model, respectively) has been validated. The implementation process may introduce errors that have to be detected. One instance of *conformance testing* amounts to showing that the behaviors of the implementation constitute a subset of the behaviors of the specification.

With this article, we continue a series of papers [21, 22, 26, 27, 28, 29] on model based test case generation for reactive systems specified with the CASE tool AutoFocus. The idea is to use the system model (a network of hierarchic components with associated behaviors) for the

[*]Supported by the DLR (project MOBASIS) and the DFG (project KONDISK/IMMA; ref.no. Be 1055/7-2).

generation of test cases that later on, are fed into the actual implementation (hardware). Note that test cases can also be used for validating the system *model* itself, but this induces the lack of an instance that decides whether or not the behavior as exhibited by the test case is a correct one–when testing the implementation, the model plays the role of this instance.

Our tool prototype takes an AutoFocus system description, translates it into Constraint Logic Programming (CLP) and symbolically executes the resulting system. This sometimes involves, however, a guessing procedure for the next transition to be taken. It is this guessing procedure that makes up the difference between search strategies such as breadth-first, depth-first, or best-first. In this paper, we re-examine an example from earlier work, apply different search strategies for the test case generation procedure, and compare them quantitatively.

Its contribution consists of a number of statistics that indicate the superiority of best-first search. If no suitable fitness function can be defined, *random depth-first search with global state storage* when deployed in a competitive parallel setting is a good choice. Furthermore, the paper discusses some conceptual differences between search strategies for model checkers with strategies for test case generation.

Related work. Incremental SW development processes include the Rapid Prototyping, the spiral (meta) model, Extreme Programming/Modeling [3, 4], and, more geared towards reactive systems, the Cleanroom Reference model (CRM, [30]). The benefit of models is particularly acknowledged in the Rational Unified Process [20] and the CRM. The embedding of model based testing in incremental processes is discussed in [27]. A theory of formal testing is tackled in [16, 5]. They share the commonality of defining an observational congruence ("selection hypotheses") on systems. Similar relations are used in [32, 31] to compute whether or not a system (model) conforms to its specification. We differ from this approach in that we do not want to prove such a conformance relation but rather approximate its proof as done in traditional testing. (Constraint) Logic Programming for test case generation has been used in [24, 7, 23]; our approach differs (1) in the class of systems we consider, (2) in the input language with a concept of interface and a combined approach to behavior specifications with automata and functional definitions on transitions, and (3) in the thereby induced necessity for powerful constraint handlers on the grounds of Constraint Handling Rules (CHR, [15]). Lutess [12] is a tool for the generation of test cases for Lustre (as is Gatel [23], see above). The difference with our approach is the use of model checkers or random number generators for the generation of test cases as well as a restriction to boolean data types.

Code generation on the grounds of CLP is, for various non-modular [25] automata considered in [17, 14]. The relationship of Model Checking and (C)LP with possibly tabled resolution procedures is discussed (and used) in [10, 13, 9]. Bounded Model Checking with propositional solvers for test case generation is considered in [33].

Test case generation on the grounds of mutation analysis is, among others, treated in [2]. In the context of mutation testing, constraints for the generation of test cases for transformational systems are used in [11]. The idea is to formulate constraints that approximate criteria for killing mutants.

[6] uses a mixture of BDDs and Presburger constraints for the representation of sets of states in reactive systems. [1] uses linear constraints on real numbers for model checking hybrid systems. Clearly, the focus is on model checking. The difference with our approach is that (1) we are mixing enumerative and symbolic techniques rather than computing fixed points on sets of constraints and (2), again, use CHR with constraint solvers on arbitrary domains (e.g., FD) for allowing convenient interactions and user-defined specifications of test cases.

Overview. The remainder of this paper is organized as follows. In the next section, we give a brief overview of the specification concepts of the CASE tool AutoFocus and introduce an example, taken from [28], that is used for assessing the different search strategies: a smart card for inhouse access control.

In the main part of this paper, in Section 3, we then present our approach to test case generation on the grounds of CLP. Several search strategies and state storage algorithms are discussed and assessed. We also examine the relationship between search strategies for test case generation and for model checking. Section 4 concludes with an outlook on current and future work.

We assume familiarity with the basic concepts of CLP, finite state machines, and those of depth-first, breadth-first, and best-first search strategies.

Terminology. The terminology in this paper is that of [22]. A test sequence is a sequence of input/output event tuples. A test case describes a set of test sequences by imposing constraints on unbound variables in the I/O tuples. A test case specification is the formalization of a test purpose (e.g., "ensure coverage criterion C"); the aim of test case generation is to find test cases/sequences that satisfy a certain test case specification. Note that this may include the test of "unspecified" parts of the system's behavior.

2 AutoFocus

In this section, we briefly present the modeling concepts of the CASE tool AutoFocus (`http://autofocus.in.tum.de`, [18]) and present an example from earlier work that we will use as a basis for the generation of test cases with CLP.

AutoFocus. Similar to the UML-RT, systems are modularly specified by different views. The architectural view shows the system structure that consists of possibly hierarchic components interconnected by typed and directed channels. Typing is achieved by predefined or user-defined data types; possibly recursive types are defined by means of a Gofer-like functional language.

The bottom level component of each hierarchic component is equipped with an extended finite state machine, and each such machine can access the component's local variables. Transitions are equipped with a guard that accesses local variables as well as, by means of pattern matching, input channels of the respective component. The language of guards is the same functional language as mentioned above; it is thus possible not only to define functionality by means of state machines but also by means of possibly recursive functions. If the guard holds true, the transition may fire, resulting in an update of the component's local variables as well as the change of the current control state. Initial states are marked with a black dot; there is no such thing as an acceptance condition. If more than one transition can fire, one is selected non-deterministically; if none can, the system idles, i.e., remains in its current state.

Components are timed by a global clock, and they all perform their computations (firing transitions) simultaneously. Communication is synchronous (asynchronous communication is implemented by explicit buffer components).

In addition to the architecture, behavior, and data view, AutoFocus makes use of sequence charts. Even though we do not consider them further here, they play an important role in the development process: for specification, definition of test cases, and for the graphical representation of simulation results.

AutoFocus is equipped with code generators for Java, C, Prolog, and ADA. Furthermore, it is connected to a plethora of external tools. These include model checkers such as SMV or μcke as well as the propositional solver SATO, ATTOL/UnitTest for coverage measurements of test cases for generated ADA code, and DOORS for requirements tracing.

Example. Our example is the model of a smart card that may be used for inhouse access. After inserting the card and a personal identification number (PIN) into a terminal, the terminal may or may not, according to the rights a user owns, grant access to a particular door. This involves running authentication/identification protocols between card and terminal. A PIN may be blocked if the authentication process fails several times, and it can be unblocked if a corresponding personal unblocking key (PUK) is being entered. This example is part of an industrial case study we carried out in order to assess practical applicability of our tool prototype, and it is discussed in more detail in [28].

The card part of the system we consider consists of a single component that accepts commands on the input channel. These commands are provided by the terminal, and they include commands for authentication/identification, reading/writing data on the card, etc. The output is a signal

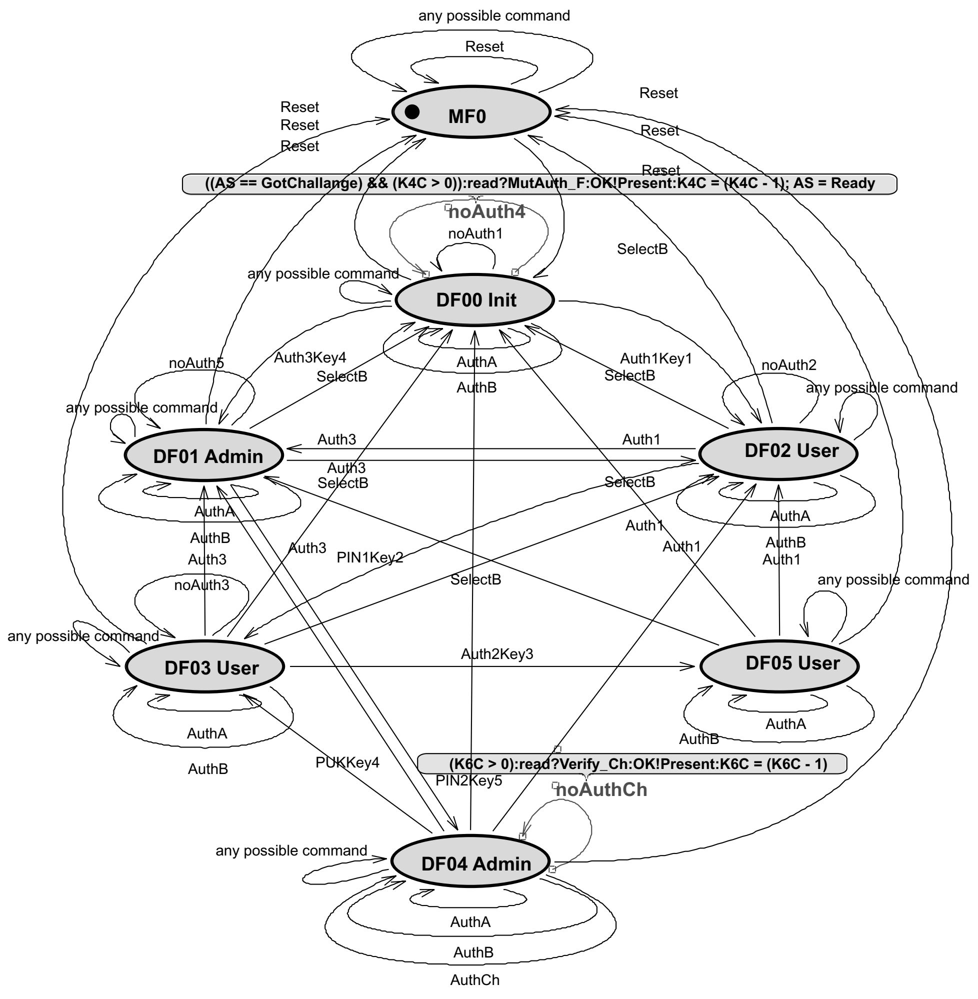


Figure 1: Inhouse Card [28]

that signifies whether or not the input command is a legal one. Its behavior is depicted in Fig. 1. Roughly speaking, the different states correspond to different access right levels to several parts of the card's data. The state is changed by an adequate authentication process. If this process fails, the respective counter (one for each state except for MF00) is decremented by one, and the card is locked if one of the counters reaches zero. By providing the corresponding PUK, it can be unlocked.

The test cases we will consider concern those that drive the card, for different counters, into the corresponding locked state. cnt_6 is the counter associated with state `DF04Admin` with initial value 15; cnt_4 is the counter associated with state `DF00Init` with initial value 14. The difficulty in finding a sequence that decrements cnt_4 to zero lies in the fact that between two decrements, a well-defined sequence of three transitions needs to be executed.

Note that this example contains a control state that is encoded by an internal variable. It stores the current status of the authentication process (three possible values). This reflects the experience that finite state machines alone quickly become too complicated.

3 Test case generation with CLP

In this section we first describe our algorithm for the computation of test cases based on symbolic execution with CLP. As mentioned before, this approach amounts to searching the system's state space. In our implementation, the part of the program that is concerned with implementing the search strategy is held rather orthogonal from its rest. We can thus modify the search strategy without altering the rest of the code that is automatically generated from an AutoFocus specification. A user interface for specifying the search strategy to be employed is the subject of current work; thus far, a strategy with interleaved choice of transitions is generated automatically (see below).

After the description of the principal ideas, we discuss and assess several search strategies: (1) bounded depth first with (1a) naive left-right choice of transitions, (1b) interleaved choice, (1c) global and (1d) local state storage, and (1e) best-first choice of transitions. We also briefly discuss (2) breadth first search. In addition, we take into account the possibility of storing sets of states by means of constraints.

Code generation for CLP is presented in more detail in [21]; interleaving transitions in our setting has first been described in [27]. The use of constraints is described in [22]; the advantages of using constraints are more thoroughly discussed in [26].

(Constraint) Logic Programming. In Logic Programming, problems are specified as sets of first order predicates (disjunctions with at most one positive literal—implications). Common LP languages such as Prolog then interpret ("solve") these predicates in a procedural manner (resolution); backtracking mechanisms are built-in. In our context, the possibility of function inversion plays a crucial role: Under certain circumstances, given the result of a function application, one can infer the function's arguments (or a set of them). This is achieved by binding free variables and backtracking until these desired arguments are found.

However, there are some pitfalls in LP. On the one hand, solutions of programs (models of logical formulae) are always based on the same carrier set, a term universe (the so-called minimal Herbrand model). On the other hand, in implementations of LP languages, there is a certain order in which predicates are evaluated (in the procedural sense, see above) which may result in infinite evaluations even though the succeeding predicate could prevent infinite backtracking by imposing constraints that its preceding predicate can only satisfy in a finite number of ways. This led to the idea of merging Constraint Languages with LP into Constraint Logic Programming (CLP) languages [19]. These languages allow for the formulation of constraints that are checked for satisfiability in every step of the evaluation of a set of logical formulae (expansion of a node in the resolution tree), and they hence necessitate mechanisms for delaying subexpressions. This yields the possibility of a priori cutting the evaluation tree of these formulae; the "generate and test" paradigm of LP languages is modified to "constrain and generate". On the other hand, with CLP, one can calculate in domains other than the Herbrand universe, for instance finite (integer) domains $\mathcal{FD}$, or rational or real numbers $\mathcal{Q}$ and $\mathcal{R}$ (one crucial point in the latter two domains is to calculate on finitely representable intervals.) One can, for instance, formulate Linear Programming Problems with a set of unknown variables, and if the CLP language is equipped with suitable constraint solvers (e.g., Simplex), the desired optimal results can be found by binding variables to the corresponding rational numbers or intervals. LP is an instance of CLP with constraints being equations over terms, or finite trees, respectively.

Even though there are many constraint solvers available, it turned out that sometimes people do not want to calculate on one specific domain but rather a mixture of different domains, and that there sometimes is a need to create new domains and constraint handlers. This led to the development of Constraint Handling Rules (CHR [15]), a meta language that allows for the definition of new constraint handlers (solvers) that, subsequently, can be translated into the corresponding target language, CLP in our case.

Idea. Test case generation is achieved by running CLP code automatically generated from specifications. Since CLP allows for unbound variables as arguments, we run an ordinary simulation, but we do not specify inputs at all moments of time. In fact, by not restricting the system at all, we get an enumeration of all possible system traces.

Each bottom level component (components with associated behaviors) is translated into a set of predicates each of which models one particular transition. The predicates' heads include thus not only source and destination states as well as the name of the transition, but also formal parameters for histories of local variables, and for those input and output channels that are connected to the component.

Bottom level components (or rather the predicates that implement them) are run by a driver predicate that corresponds to the component that contains those bottom level components. Channels between components are modeled by local variables (existential quantification). This simple translation scheme can be applied recursively, and the driver predicates thus reflect the hierarchy of AutoFocus components.

In terms of function and data type definitions used for guards and postconditions, they are translated into constraint handlers or Prolog code and integrated into the model [21, 22].

As mentioned above, test case generation is done by simulating the system w.r.t. constraints as imposed by the test case specification. These constraints may refer to structural–such as "all transitions should be fired at least once" or "all control states are to be visited once"–or functional criteria (such as "find a trace that makes output o_1 happen and then output o_2" or "find a trace that includes input i and exhibits nothing but outputs from a set O"). These constraints are formulated by appropriate constraint handlers for Booleans or enumerative types (finite domains). In practice, this is, however, not enough for an adequate formalization of test purposes. User-defined constraint handlers based on Constraint Handling Rules [15] are a powerful tool for specifying test cases with predefined as well as user-defined constraint handlers, and they are integrated into our system.

While the possibility of a-priori-pruning the search tree is one advantage of using constraints, we consider the possibility of restricting the model to be the key to scalability: Often, designers know that for a particular functional test case, certain parts of the model are irrelevant; constraints allow to ad-hoc slice the model without actually altering it [26]. This also leaves room for techniques widely used in program analysis.

A further advantage of using constraints is that they allow for reducing the number of traces. As an example, consider a transition with a guard $i(t) = c_1 \vee i(t) = c_2 \vee \ldots \vee i(t) = c_n$ for input channel i at time t and commands c_j (elements of a data type "Command"). In a naive flattening Prolog translation of this fragment, n transitions need to be tried: i has to be bound to each c_j. Using constraints, we are happy with one single member constraint: $\{i(t) \in \bigcup_{j=1}^{n} c_j\}$.

In this way, a computed test case may represent a set of test sequences (those with $i(t) \in \bigcup_{j=1}^{n} c_j$ which might have been strengthened in the further computation). When testing the actual implementation, we need fully instantiated traces; how to perform this instantiation intelligently is, however, not the focus of this paper.

Naive $\rightarrow$. The simplest possible search strategy is a naive left-right-choice of transitions. As stated above, transitions are encoded by predicates that contain, among other things, the source and the destination state of a transition in their head. Choosing a transition can then be left to the evaluation mechanism of CLP: the transition predicate that occurs first is tried first, then the second, and so on. However, this approach is rather inefficient for it (1) revisits states that have been visited before and (2) is likely to run into cycles (see below).

Interleaving. In previous papers [27, 26], we have described our implementation based on a bounded depth-first search with an interleaved choice of transitions. This means that when a state is revisited, its outgoing transitions are not tried in exactly the same order as before; the list of transitions is rather shifted by one. If $Tr_{s,i} =< T_1, T_2, \ldots, T_n >$ is the sequence of transitions that are tried in that order from state s for the ith time, then $Tr_{s,i+1} =< T_2, T_3, \ldots, T_n, T_1 >$ is the respective sequence when s is visited the $i + 1$st time. This simple mechanism, though imposing a little overhead, usually results in significant performance gains in terms of time since it decreases the likelihood of running into (control state) loops (Fig. 2 (a); with a naive left-to-right choice of transitions, the dashed ones are never taken.)

Storing states globally. One problem with naive bounded depth-first search is that states may be visited more than once. A state here refers to the cross product of tuples of data and control states for each component (one in the case of our example; the role of I/O as well as internal communication channels is discussed below). The obvious solution is to store each state once it has been visited. However, there is a price to pay: With this approach, the maximum

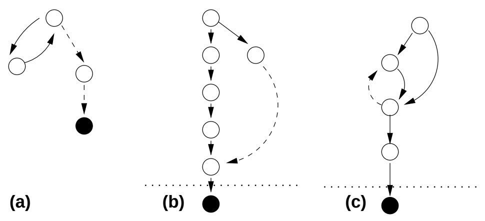


Figure 2: Search strategies

depth of the search tree becomes more important. In Fig. 2 (b), the black state that makes a trace satisfy the corresponding test case specification, is not reached (the horizontal line indicates the maximum depth). The reason is that the dashed transition is not taken for its destination state has already been visited and thus stored, indicating that it is not to be considered for further search. This is even though the black state could, on backtracking, be reached in fewer steps than the maximum depth. Note that for *testing* purposes, it is not even always desirable to exclude states from being visited twice; this reflects the strange kind of errors where something goes well n times, but not $n + 1$.

Storing states locally. This motivates another search strategy. States are prevented from being revisited only when the depth at the moment of their storage is below the depth of the new visit (Fig. 2 (c) where the right transition from the initial state may be taken even though the corresponding destination state has already been visited). The problem with this approach is that it necessitates the storage of a large number of additional states; its benefit is that it usually allows for detecting shorter traces than the ones detected by strategies that store states globally. However, for our examples of decremented counters, no performance gains could be achieved. Since the number of possible traces increases with this local storage algorithm, the performance is rather significantly reduced.

Note that for structural test purposes such as “find a transition tour” the approach of storing states is not applicable. In Fig. 2 (d) the dashed transitions result in unsuccessful traces since the initial state has already been visited. A transition tour cannot be computed in this way. Also note that it depends on both the test case specification and the system whether or not the stored states need to include information about input and output channels.

Storing sets of states. The use of CLP enables one to store sets of states by simply storing the constraints that describe this set. Deciding whether or not a stored state *entails* a potentially reachable one is then crucial for deciding whether or not the respective transition needs to be taken. This issue is the subject of future work; [10] contains a discussion in the setting of deductive model checking with CLP.

To illustrate the idea, we consider a naive first implementation. Rather than storing each state $s = (\mathit{ctl}, \mathit{auth\_stat}, c_1, \ldots, c_6)$ (control state, data state for authentication process, six counters) separately, we look if there is already a stored state s' that differs from s in exactly one component. Say that the difference occurs in the control component, i.e., $s' = (\mathit{ctl}', \mathit{auth\_stat}, c_1, \ldots, c_6)$ with $\mathit{ctl}' \neq \mathit{ctl}$. We now delete s from the database and replace it by $(\{\mathit{ctl}, \mathit{ctl}'\}, \mathit{auth\_stat}, c_1, \ldots, c_6)$. If later on, we run into a state s'' which again, differs from s and s' in nothing but the first component, we delete the corresponding entry from the database and replace it by $(\{\mathit{ctl}, \mathit{ctl}', \mathit{ctl}''\}, \mathit{auth\_stat}, c_1, \ldots, c_6)$, and so on. Doing so for all components, this simple strategy results in a reduced need for memory: all components of the vectors except for the first just need to be stored just once.

Since storing sets of states is not the central issue of this paper, we do not go into further detail here. Two remarks are, however, in order. It is easy to see how the simple strategy can be made more powerful: Rather than restricting the search for already visited states to one component, we could look for the projection onto two or even more of them and update the database accordingly. This does require some further intelligence though: While different entries in the database correspond to a disjunction of states, the components of a state vector correspond to conjunctions: Consider a stored vector $(\{\mathit{ctl}_1, \mathit{ctl}_2, \mathit{ctl}_3\}, \mathit{auth\_stat}, c_1, \ldots, c_6)$ and a newly

```
$SP = sorted\_shortestPaths(s)$
forall $p \in SP$
  $T_{p[1]} = trans(p[1], p[2])$
  for $i = 1..\|SP\|$
    if $SP[i] \neq p$ then $T_{p[1]} = T_{p[1]} \circ trans(p[1], (SP[i])[1])$
return $\bigcup_{p \in SP} T_{p[1]}$
```

Figure 3: Transition ordering for best-first

visited state $(ctl_1, auth\_stat', c_1, \ldots, c_6)$ with $auth\_stat' \neq auth\_stat$. Clearly, we cannot simply coalesce both entries into $(\{ctl_1, ctl_2, ctl_3\}, \{auth\_stat, auth\_stat'\}, c_1, \ldots, c_6)$. We can, on the other hand, combine sets of states when it is sound to do so, but this requires efficient comparisons of sets of sets with the above mentioned suitable definition of entailment—obviously, we are looking for generalizations of Binary Decision Diagrams to domains other than the Booleans.

The second remark aims in the same direction. We will later see how an implementation of the above idea does indeed result in considerable reduction of allocated memory. This implementation stores states in the above mentioned manner but does not use them for the computation of traces itself. This would enable one to *compute with sets of states* rather than single states, something that is referred to as the "collecting semantics" in the literature on abstract interpretation (e.g., [8]). Knowledge of this computed collecting semantics may turn out to be a good starting point for the definition of an abstract semantics that can, in turn, be used for an actual abstract interpretation.

Best-first. As we will undermine quantitatively in the following, the ordering in which transitions from a given (control) state are taken is crucial for the performance of the test case generation. This issue becomes increasingly important when there are many transitions from that state. In some cases, it is possible to define a fitness function that w.r.t. a test case specification computes the (approximately) best transition to be taken. In the case of control states to be reached or transitions to be taken, this fitness function is comparatively easy to define: From each control state $c \in S$, we compute the shortest path p_c to reach the desired destination state s. We then implement a best-first search by defining the transition ordering for state c as follows. Transitions that connect c with the second state in p_c are tried first. For the remaining transitions emanating from c, we choose those that lead to a state d satisfying the requirement that there is no d' where the length of $p_{d'}$ is strictly shorter than that of p_d.

Iterating this procedure leads to a transition ordering that first tries transitions from c that are on p_c. It then tries those transitions that lead to states with minimum shortest paths to s, then those that lead to states with the second minimum paths, etc. This algorithm works because each subpath of an optimal path is itself optimal.

More formally, in Fig. 3, SP with length $\|SP\|$ is the sequence of shortest paths (sequences of control states) from all states $s' \in S$ to s. It is assumed to be sorted with the shortest path occurring first (this could be the path from s to itself; thus modeling the idle transition). Furthermore, it does not contain duplicates. For a sequence q, $q[i]$ denotes the ith element of q, and for sequences p, q, $p \circ q$ denotes their concatenation. $T_{p[i]}$ is the transition sequence that should be taken in this order when state $p[i]$ is to be left. $trans(s_1, s_2)$ returns a sequence of all transitions from s_1 to s_2 in arbitrary order. The existence of idle transitions prevents $trans$ from returning the empty sequence in Fig. 3. Note that if there is no path from a state to s, it does not occur as the first element of a sequence in SP. This implies that transitions that lead to states that cannot reach s are (safely) ignored.

In our example, we are interested in optimal orderings w.r.t. reaching state DF00Init for decrementing cnt_4, and to reaching DF04Admin in order to decrement cnt_6. This is because transition noAuth4 is responsible for decrementing cnt_4, and transition noAuthCH is responsible for decrementing cnt_6. We thus use the algorithm for reaching states as one that is capable of finding good transition orderings for reaching transitions; as a further step, we move transitions noAuth4 and noAuthCH in front of the respective transition sequences.

Note that this simple heuristics is, in general, applicable only to control states for the number of data states may be too large. When the system contains data states, it is only a heuristic.

Table 1: $cnt_6 \to 0$

c_{max}		states	time	mem [MB]	len/succ
30	i	1982	14.8	3.0	30
	$\to$	3348	36.1	3.5	–
	r	934-2424-4160	3.9-23.7-457.3	2.6-3.1-3.8	1/10
	bf	21	0.0	2.1	21
40	i	1442	8.3	2.8	40
	$\to$	7137	157.6	5.0	–
	r	2149-5239-8399	18.0-91.3-209.5	3.0-4.2-5.4	4/10
	bf	21	0.0	2.1	21
50	i	1375	7.9	2.7	50
	n	10520	316.2	6.3	50
	r	1884-8183-14626	13.7-236.3-596.4	4.4-6.2-7.8	7/10
	bf	21	0.0	2.1	21
100	i	4789	65.7	4.1	100
	$\to$	1346	7.7	2.7	100
	r	731-7860-15571	2.8-233-595.8	2.5-5.2-8.2	10/10
	bf	21	0.0	2.1	21
150	i	197	0.2	2.3	150
	$\to$	1528	9.2	2.8	150
	r	446-7866-26714	1.2-285.8-1552.4	2.3-4.9-12.3	10/10
	bf	21	0.0	2.1	21
200	i	518	1.2	2.4	200
	$\to$	1468	8.6	2.8	200
	r	3426-21279-62565	39.7-1542.6-6821	3.4-10.3-21.3	10/10
	bf	21	0.0	2.1	21
250	i	535	0.7	2.4	250
	$\to$	2704	26.6	3.3	250
	r	359-13977-43397	0.4-744.6-3304.4	2.2-7.5-18.8	10/10
	bf	21	0.0	2.1	21
500	i	1518	4.6	2.8	281
	$\to$	705	1.1	2.5	500
	r	5171-43030-115563	65.6-5289.2-20452.1	4.1-20.5-46.5	10/10
	bf	21	0.0	2.1	21
1000	i	1884	6.9	2.9	281
	$\to$	900	0.9	2.6	739
	bf	21	0.0	2.1	21

This is the case in our example.

Breadth-first. The desire to get the shortest possible traces makes breadth-first search come into the game for it guarantees traces of minimum length. However, breadth first search severely suffers from memory explosion (at least if we do not employ symbolic representations such as BDDs). For the sake of a smaller memory allocation, we did not store the traces in the experiment. It is noteworthy that the architecture of our system does not necessitate the implementation of a naive meta interpreter (as usually done in breadth-first implementations in Prolog).

Experiments. We now give some experimental data, measured on a Pentium III with 850MHz and 256MB of memory. The CLP implementation we use is Eclipse (`www.icparc.ic.ac.uk/eclipse`) As mentioned above, we consider two functional test case specifications, namely decrement counters cnt_4 and cnt_6. Since a naive interleaving strategy without storing states as well as the strategy with local state storage resulted in prohibitive amounts of time needed, we omit mentioning the respective numbers here.

Concerning the test case that decrements cnt_6 to zero, Tab. 1 summarizes resource allocation for a strategy with globally storing states for interleaving (i), naive left-to-right ($\to$), random

Table 2: $cnt_4 \to 0$

c_{max}		states	time [s]	mem [MB]	len/succ
300	i	986	2.0	2.6	300
	$\to$	13138	420.2	7.3	300
	r	326-21488-56444	0.3-2146-7600.3	2.2-10.3-23.8	10/10
	bf	43	0.0	2.1	44
400	i	512	0.6	2.4	325
	$\to$	34715	3284.6	15.4	400
	r	328-1319-6286	0.3-10.9-86.0	2.3-2.7-4.6	10/10
	bf	43	0.0	2.1	44
500	i	512	0.5	2.4	325
	$\to$	22251	1386.4	10.7	500
	r	445-892-3397	0.5-3.5-27.4	2.4-2.6-3.5	10/10
	bf	43	0.0	2.1	44
1000	i	512	0.5	2.4	325
	$\to$	1188	1.8	2.7	846
	r	383-604-1176	0.4-782-2120	2.4-2.4-2.7	10/10
	bf	43	0.0	2.1	44

(r) and best-first (bf) choice of transitions. For random choice, ten simulations have been run; the respective entries consist of min-mean-max triples. The first column shows the number of (globally) stored states, the second the time needed to find the first sequence satisfying the test case specification, the third the required memory (including 2.1MB required for the runtime system and the code), and the fourth the length of the respective trace (or, in the case of random choice, the number of runs that led to a successful trace). Since we consider bounded depth-first search in this case, we give data for several values of the maximum depth, c_{max}. Tab.2 contains the respective data for cnt_4.

Tab. 3 contains some data on breadth-first-search. Since the considered test cases lead to sequences longer than the maximum depth of 12, we just show the cumulative amounts of time and memory needed to proceed up to the specified depth.

In conjunction with breadth-first search, an implementation of the above mentioned simple approach to storing sets of states allows us, under the given resources, to enumerate states up to a depth of 14 rather than 12 (for a depth of 15, the virtual memory of 750MB gets again exhausted).

The last experiment we conducted consisted of manually slicing the model by prohibiting the system of firing those transitions that lead to decrements of the other four counters. This is done by means of constraints rather than by altering the model itself. The results are almost identical to those of best-first-search. However, the approach of imposing additional constraints is, in a sense, more general than using a best-first search: It may not always be possible to define a suitable fitness function. Slicing by constraints does, on the other hand, necessitate detailed knowledge of the system under consideration. We also used the approach of interactive slicing in the original study [28].

Discussion. As mentioned above, local state storage increases performance, but in our examples, the necessary resources still were too high as to consider this strategy a serious candidate. Its advantage lies is in potentially detecting shorter sequences.

Not surprisingly, breadth-first search results in an explosion of memory requirements. It is, however, a serious candidate for deductive model checking when storing sets of states by means of constraints together with approximation procedures as proposed in [10] are taken into account.

The experiments with depth-first search with global storage exhibit the commonality of finding rather long sequences (which is usual for depth-first search). As suspected, the choice of the ordering of transitions is crucial w.r.t. performance of the algorithms. This becomes evident with the results on random choice of transitions where the minimum and maximum performance in terms of time differs as much as four orders of magnitude. Storing states may lead to unsuccessful

Table 3: Enumerating states

depth	time [s]	mem [KB]
6	0.0	57
7	0.2	160
8	0.7	529
9	2.3	1,879
10	8.9	6,870
11	33.7	23,476
12	127.8	86,792
13	>600	>750,000

runs for a given maximum search depth even though in principle, the test case specification could be satisfied by a trace shorter than this maximum depth.

According great importance to the transition ordering is consistent with the data on best-first search. This strategy is able to compute short sequences with negligible overhead. In fact, it was able to compute traces of minimum length; regardless of of c_{max}, resource allocation remained constant. A simpler version of the fitness function that simply consists of always trying transitions noAuth4 or noAuthCH, respectively, first, and not caring about the ordering of the remaining transitions, leads to sequences that are a little longer, but can be found with comparable resources. It is likely, however, that this finding does not generalize when systems with more control states are to be tested. In fact, if a second version [28] of the model that replaces a data state by a set of additional control states (for the authentication protocols) is used for test case generation, the differences between the two fitness functions result in performance gains of one order of magnitude for the fitness function that is based on shortest paths. Furthermore, the automaton in this paper is heavily interconnected which explains why depth-first search with interleaving comparably efficiently produces traces for large maximum depths. In these cases, the larger the maximum depth, the more likely it is to quickly find a trace. In [27] we noticed that the choice of c_{max} has an important influence on the performance. Good heuristics for finding suitable maximum depths remain to be found.

An interleaving choice of transitions is usually preferable to an arbitrary fixed ordering; the advantage becomes negligible with growing c_{max}. This is due to the nature of depth-first search in highly interconnected graphs when augmented by a strategy that stores states.

With these findings, the numbers for random choice of transitions with a stunningly high standard deviation are easily explained. In the optimal cases (minimum requirements), the random choice is such that it is favorable for finding the particular test sequence.

These results suggests that if it is easy to define a fitness function, one should consider using a best-first strategy. If it is not, one should instead try a random choice of transitions (states globally stored) and perform several computations simultaneously. An additional process should use an interleaving choice. The first process to terminate successfully then kills the others. In our examples, that multiplies the minimum requirements by 11 (plus a little overhead for process scheduling), but this seems to be reasonable when taking into account the rather small requirements for the optimal solutions.

Finally, while our example is not a concurrent one, the encoding into CLP that keeps different components separate, is likely to assure that the results also hold in a concurrent setting. This claim remains to be verified; we are planning a new study with smart card systems that can best be modeled by concurrent components.

Model Checking. The resemblance of many issues discussed in this paper with model checking is eye-catching. This is for several reasons. Firstly, using the capability of model checkers to produce counterexamples for test case generation is an old idea. In fact, at least when they are reasonably short, they are used for localizing errors in specifications. Second, the problem of identifying a suitable search strategy does constitute large parts of work done in areas such as non-symbolic on-the-fly model checkers like SPIN. Thirdly, and maybe less obviously, the main problem consists of computing (approximations of) fixed points or subsets thereof. We will briefly address these issues in the remainder of this section.

Model checkers, at least without built-in abstraction capabilities as found in Uppaal or HyTech, are inherently restricted to small finite systems. A CLP based approach to test case generation like ours is not. This is because we explicitly construct parts of the state space on-the-fly rather than first constructing and then computing fixed points on it as done in symbolic checkers. Constraints–in a sense, a generalization of boolean formulae encoding transition relations and encoded by BDDs–are useful in at least two ways. Firstly, they allow for storing possibly infinite sets of states (see also [10, 13], the latter of which notes that the boundary between traditional model checkers and CLP systems has become blurry, as in the case of HyTech). Secondly, powerful self-defined constraint handlers like CHRs can be used for interactively slicing the model [26]. Again, we consider this a key to scalability of our approach.

The problem of finding suitable search strategies is very similar in non-symbolic model checking. In fact, as bounded depth-first search has become an option in available tools, our approach is not really different. At least conceptually, there is a slight difference between model checking and generating test cases for the latter *always* tries to find a witness. Model checkers usually aim at proving a property of an entire system. This difference allows us to adopt the search strategy w.r.t. a given test case specification (note that this is also principally possible for on-the-fly model checkers, but this requires exact knowledge of the respective search algorithm). It is noteworthy that in testing, one is mainly interested in existentially path-quantified properties. The approach of first generating the entire state space may thus not be the prime solution in this context.

Again, the use of constraints can help in storing possibly infinite sets of states which, in conjunction with a suitable concept of entailment, may lead to more powerful search strategies. This claim remains to be verified empirically. Note that partial reduction, as used in SPIN, is unlikely to yield performance gains for AutoFocus systems are inherently synchronous. However, we consider it interesting to apply partial reduction to test case generation for discretized mixed discrete-continuous (or hybrid) systems.

Finally, CLP with suitable memoing techniques can be used directly for model checking [9, 10]. Results from the area of deductive databases (e.g., bottom-up evaluation with magic templates) may prove powerful also for model checking reactive systems. The idea is to generate fixed points or approximations thereof in a bottom-up manner. This results in deductive model checking procedures for possibly infinite systems (by means of widening/narrowing operators). While thus far we have concentrated on test case generation, our existing infrastructure directly provides an experimentation platform for deductive model checking (e.g., with the post-μ calculus and appropriate query logics).

To summarize, model checking and test case generation (on the grounds of CLP) are different w.r.t. their purpose, and it seems reasonable to advocate a complementary use of them. While they are quite similar in terms of problems related to search strategies, the use of constraints may prove a powerful tool to handle complex systems.

4 Conclusion

We have presented experimental data for several search strategies implemented in our CLP-based tool prototype for generating test cases from AutoFocus system specifications. An example from an earlier feasibility study was used to identify circumstances under which certain strategies are likely to perform better than others. Furthermore, we have discussed the relationship of model checking with our approach that is based on symbolic execution with constraints. The main differences are grounded on the handling of infinite (or large) state spaces as well as the opportunity of using constraints to naturally slice models (without actually having to alter them).

There is a plethora of future work to be done in this area. The need for research in the area of good fitness functions for different system topologies and/or properties to be tested is obvious. In addition, we consider efficient strategies for storing sets of states as a prerequisite for handling very large systems, too. In the following, we pick some additional aspects from a recent position paper [26].

One issue is the use of our approach to generating test cases for mixed discrete-continuous systems [29]. Storing sets of states as well as partial order reduction techniques seem promising candidates to successfully tackle this particularly difficult class of systems.

The exact relationship between a set of test cases and the system to be tested is not entirely clear. Rather than defining a congruence on traces and use this congruence to generate tests, we generate tests and are interested in how they relate to the system (or specification) to be tested. This is likely to result in the definition of a suitable approximation order on finite state machines (classical notions of refinement/abstraction with chaos completion pose problems when lifting traces to systems and being concerned with resulting input-enabled nondeterministic systems; the role of idling–or δ [32]–transitions makes such systems difficult to embed in a refinement context).

In addition, suitable input languages for test case specifications are needed. One might consider message or live sequence charts, but they would require an intuitive notation for and semantics of negation.

The problem of finding good heuristics for instantiating test cases into test sequences has been mentioned above. For numeric values, one can use limit analyses, e.g., instantiating an interval to three single values.

This example also leads to a notion of "bad test cases". Experience shows that specifications are incomplete (while a simulation semantics, in particular with idle transitions, suggests completeness), and that errors tend to occur where designers forgot to specify some special cases (e.g., a missing else-branch in an if-statement). Test case generators should be able to compute test cases that test these cases; this is related to (a) the role of idle transitions and (b) the definition of suitable coverage metrics. For state machines with functional definitions as allowed in AutoFocus, such metrics do not, to the author's knowledge, even exist.

Furthermore, generating large sets of test cases to satisfy some coverage metrics only makes sense if the generated test cases on the level of models can be lifted to the implementation level (i.e., generated code, or hand-written code when generators produce inadequate code) *by maintaining the respective coverage criterion.* On the implementation level, such test cases are used by certification authorities to verify conformance with a given standard (e.g., DO-178B for aircraft).

With a component-oriented specification language, we consider a compositional approach to test case generation promising: Generating test cases for (simple) components and combining them into test cases for larger systems. The problem is that a test case for one component may (and often will) be a forbidden behavior of the composed system.

Finally, a model-based development process and automatic test case generation needs to pay off. Some of our industrial partners in the area of safety-critical systems seriously consider implementing such a process which will provide an opportunity to assess benefits and problems.

Acknowledgment. The tool prototype is developed together with Heiko Lötzbeyer whose contributions are gratefully acknowledged. Oscar Slotosch built the original model [28].

References

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, February 1995.

[2] P. Ammann and P. Black. Test Generation and Recognition with Formal Methods. In *Proc. 1st Intl. workshop on Automated Program Analysis, Testing, and Verification (WAPATV'00)*, pages 64–67, 2000.

[3] K. Beck. *Extreme Programming Explained: Embrace Change.* Addison Wesley, 1999.

[4] M. Boger, T. Baier, F. Wienberg, and W. Lamersdorf. Extreme modeling. In *Proc. Extreme Programming and Flexible Processes in SW Engineering (XP'00)*, 2000.

[5] E. Brinksma. A theory for the derivation of tests. In *Proc. 8th Intl. Conf. on Protocol Specification, Testing, and Verification*, pages 63–74, 1988.

[6] T. Bultan. *Automated symbolic analysis of reactive systems.* PhD thesis, University of Maryland, 1998.

[7] A. Ciarlini and T. Frühwirth. Using Constraint Logic Programming for Software Validation. In *5th workshop on the German-Brazilian Bilateral Programme for Scientific and Technological Cooperation*, Königswinter, Germany, March 1999.

[8] P. Cousot and R. Cousot. Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM symp. on Principles of Programming Languages (POPL'77)*, pages 238–252, 1977.

[9] B. Cui, Y. Dong, X. Du, K. Narayan Kumar, C. Ramakrishnan, I. Ramakrishnan, A. Roychoudhury, S. Smolka, and D. Warren. Logic programming and model checking. In *Proc. PLILP/ALP*, Springer LNCS 1490, pages 1–20, 1998.

[10] G. Delzanno and A. Podelski. Model Checking in CLP. In *Proc. Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, pages 223–239, 1999.

[11] R. DeMillo and A. Offutt. Constraint-Based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.

[12] L. du Bousquet and N. Zuanon. An overview of lutess, a specification-based tool for testing synchronous software. In *Proc. 14th IEEE Intl. Conf. on Automated SW Engineering*, October 1999.

[13] L. Fribourg. Constraint logic programming applied to model checking. In *Proc. 9th Int. Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'99)*, LNCS 1817, Venice, 1999. Springer Verlag.

[14] L. Fribourg and M. Veloso-Peixoto. Automates Concurrents à Contraintes. *Technique et Science Informatiques*, 13(6):837–866, 1994.

[15] T. Frühwirth. Constraint Handling Rules. In *Constraint Programming: Basics and Trends (LNCS 910)*, pages 90–107. Springer Verlag, 1995.

[16] M. Gaudel. Testing can be formal, too. In *Proc. Intl. Conf. on Theory and Practice of Software Development (TAPSOFT'95)*, LNCS 915, pages 82–96, Aarhus, Denmark, May 1995.

[17] G. Gupta and E. Pontelli. A Constraint-based Approach to Specification and Verification of Real-time Systems. In *Proc. IEEE Real-time Symposium*, pages 230–239, San Francisco, December 1997.

[18] F. Huber, B. Schätz, and G. Einert. Consistent Graphical Specification of Distributed Systems. In *Industrial Applications and Strengthened Foundations of Formal Methods (FME'97)*, LNCS 1313, pages 122–141. Springer Verlag, 1997.

[19] J. Jaffar and M. Maher. Constraint Logic Programming: A Survey. *J. Logic Programming*, 20:503–581, 1994.

[20] P. Kruchten. *The Rational Unified Process - An Introduction.* Addison Wesley, 2000.

[21] H. Lötzbeyer and A. Pretschner. AutoFocus on Constraint Logic Programming. In *Proc. (Constraint) Logic Programming and Software Engineering (LPSE'2000)*, London, July 2000.

[22] H. Lötzbeyer and A. Pretschner. Testing Concurrent Reactive Systems with Constraint Logic Programming. In *Proc. 2nd workshop on Rule-Based Constraint Reasoning and Programming*, Singapore, September 2000.

[23] B. Marre and A. Arnould. Test Sequence Generation from Lustre Descriptions: GATEL. In *Proc. 15th IEEE Intl. Conf on Automated Software Engineering (ASE'00)*, Grenoble, 2000.

[24] C. Meudec. ATGen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution. In *Proc. 1st Intl. workshop on Automated Program Analysis, Testing, and Verification*, Limerick, 2000.

[25] O. Müller and T. Stauner. Modelling and verification using Linear Hybrid Automata. *Mathematical Computer Modeling of Dynamical Systems*, 6(1), March 2000.

[26] A. Pretschner and H. Lötzbeyer. Model Based Testing with Constraint Logic Programming: First Results and Challenges. In *2nd ICSE Intl. Workshop on Automated Program Analysis, Testing, and Verification (WAPATV'01)*, May 2001. To appear.

[27] A. Pretschner, H. Lötzbeyer, and J. Philipps. Model Based Testing in Evolutionary Software Development. In *Proc. 11th IEEE Intl. Workshop on Rapid System Prototyping (RSP'01)*, June 2001. To appear.

[28] A. Pretschner, O. Slotosch, H. Lötzbeyer, E. Aiglstorfer, and S. Kriebel. Model Based Testing for Real: The Inhouse Card Case Study, 2001. Submitted to FMICS'2001.

[29] A. Pretschner, O. Slotosch, and T. Stauner. Developing Correct Safety Critical, Hybrid, Embedded Systems. In *Proc. New Information Processing Techniques for Military Systems*, Istanbul, October 2000. NATO Research and Technology Organization.

[30] S. Prowell, C. Trammell, R. Linger, and J. Poore. *Cleanroom Software Engineering.* Addison Wesley, 1999.

[31] V. Rusu, L. du Bousquet, and T. Jéron. An Approach to Symbolic Test Generation. In *Proc. Integrated Formal Methods*, 2000.

[32] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software–Concepts and Tools*, 17(3):103–120, 1996.

[33] G. Wimmel, H. Lötzbeyer, A. Pretschner, and O. Slotosch. Specification Based Test Sequence Generation with Propositional Logic. *J. Software Testing, Validation, and Reliability*, 10(4):229–248, 2000.

Towards Formal Test Purposes

René G. de Vries and Jan Tretmans [*]
University of Twente
Formal Methods and Tools group, Department of Computer Science
P.O. Box 217, 7500 AE Enschede, The Netherlands
{rdevries, tretmans}@cs.utwente.nl

Abstract

This paper proposes a framework to formalize test purposes. It introduces the notion of exhibition and the concept of observation objectives. Observation objectives are related to test purposes, and are used for test selection. The framework gives means for reasoning about observation objectives, test suites and test results based on observation objectives. We instantiate the presented framework with the **ioco** conformance test theory and present an algorithm to derive an e-complete and sound test suite. Finally, we discuss some ideas on how to put observation objectives into practice.

1 Introduction

In this paper we consider conformance testing of reactive systems. A reactive system is a system that interacts with its environment by accepting inputs and producing outputs. Conformance testing is the activity of checking whether a system is correctly implemented according to its functional specification. Such a specification typically prescribes the interactions between the system and its environment. Testing performance, robustness, etc., although important, are outside our scope.

Testing consists of doing experiments with a system in a controlled way. Where the system normally interacts with its environment, the tester now pretends it is the environment. The tester stimulates and observes the system. The experiment carried out by the tester is defined by a test scenario, often referred to as a *test case*. Conformance testing involves black box testing, i.e., testing without knowing the internals of the system. We can only interact with the system by using the available, external interfaces.

Testing is laborious, error-prone and expensive, hence it is preferably automated as far as possible. This concerns both the generation of tests from a specification and the subsequent execution of these tests. Automation of test generation is only feasible if the specification is amenable to automatic processing by a test generation tool. This implies that it should be expressed in some formal language. Many theories have been developed for automatic derivation of tests from formal specifications, e.g., [LY96, Tre96].

A problem with automatic test generation is that it may result in large and unstructured test suites. Test generation algorithms can produce large, or even infinite numbers of test cases. The result is that the execution of such a test suite is not feasible within reasonable time limits – or even impossible, when it is infinite. Moreover, the relation between a test case and the specification requirement that it tests, i.e., its *test purpose*, gets lost. To overcome this problem we have to reduce the test suite by making a selection from the set of all possibly generated tests. Moreover, for each selected test its purpose should be clear. We refer to this activity as *test selection.*

[*]This research was supported by the Dutch Technology Foundation STW under project STW TIF.4111: *Côte de Resyste* – COnformance TEsting of REactive SYSTEms.

Test selection can be done at random, but the purpose of a randomly selected test case will be difficult to establish. The random approach is the one currently taken in the test tool TorX [BFV$^+$99]. A second approach is having criteria or heuristics added to the test generation algorithm so that it produces only a subset of all possible tests. A third approach to test selection is to have a test person specifying criteria, strategies or objectives to steer the generation of tests. Examples of such criteria are a particular coverage to be reached by the generated tests, or some particular behaviour, in terms of sequences of inputs and outputs of the system under test, that a test person would like to see during the testing process. This approach is taken in the test tools TGV [JM99] and Autolink [SEK$^+$98]; the test tool user has to specify a *test purpose* which is a sequence of possible interactions, for which the tools then derive a corresponding test case. In this paper we will pursue this third approach, and we refer to such criteria, strategies, purposes or objectives as *observation objectives*.

Observation objectives are very much related to *test purposes*. Different descriptions of what a test purpose is exist, both informally and formally. In the formal framework [ISO96] a test purpose is a set of models of implementations ($P \subseteq MODS$). In the informal setting of [ISO91] a test purpose is a prose description of the goal of what is being tested by a particular test case, referring to a particular conformance requirement. Other uses of the concept of a test purpose include automata [JM99], traces [SEK$^+$98] or syntactic restrictions on a specification [Hol99]. In [DRS$^+$00], which uses test purposes in the automata sense of TGV [JM99], it is concluded that a precise notion of a test purpose on the theoretical and methodological level is needed.

In this paper we propose such a precise notion of a test purpose. We try to formalize test purposes in the vein of [ISO96] and consistent with their use in tools like TGV and Autolink. Since, to our opinion, the word test purpose is already overloaded with many different meanings, we use the term *observation objective* instead.

The first point, when using observation objectives to steer the test generation process, is that we need a formal way of specifying an observation objective. By the nature of our black box testing approach, we can only make observations of the interactions of a system with its environment. Hence, the observations that we can make during testing will form the basis for defining an observation objective. Secondly, we need a clear and precise notion of how an observation objective is related to a test suite. This will be the basis to decide whether a particular test case will be selected. Thirdly, we must know how to interpret the results of test execution with respect to an observation objective, i.e., there must be a notion of satisfying, or reaching, an observation objective. The formal elaboration of these concepts in Section 3 constitutes the main part of this paper.

It is important to note the difference between conformance and observation objectives. Conformance expresses the notion of formal correctness: if during testing any non-conforming behaviour is observed, then this is sufficient to reject the implementation. It is a decision about correctness based on purely formal arguments. On the other hand, an observation objective describes desired behaviour which we wish to observe but which is not directly related to required behaviour or correctness. If this desired behaviour is actually observed then it may contribute to increased confidence in the implementation's correctness. If the observations described in the observation objective are not actually observed then no definite conclusion can be based solely on this information. Additional information from outside the realm of formal reasoning is necessary to interpret a missed observation objective.

Consider as an example a coffee machine which is specified to produce coffee after a coin has been inserted, but which, e.g., due to lack of cups, may also respond with a red light after a coin has been inserted. An example of an observation objective is that a test person wishes to see coffee being supplied after a coin has been inserted. If after several test runs the machine each time responds with the red light this observation objective is certainly not satisfied. However, there is also no reason to declare the machine non-conforming since the red light response constitutes valid behaviour according to the specification. In the terminology of [ISO91] this test would get the verdict *inconclusive*. The decision whether this implementation is acceptable, is outside the formal domain. It depends on the context, the test person and/or future users whether a machine which has never exhibited the ability to supply coffee can be put into operation. On the other hand, suppose the coffee machine would produce tea after insertion of the coin then a formal argument suffices to decide that the machine is non-conforming and should be rejected. In Section 3 we will formally elaborate this orthogonality between conformance and observation objectives. We will see that the coffee machine above **passes** the test but **misses** the observation objective.

As the basis for the formalization of observation objectives we take the framework *Formal Methods in Conformance Testing* [ISO96] and related frameworks [Tre99]. Section 2 recalls the main concepts of them. Then Section 3 presents the formalization of observation objectives as an extension of this framework. The concepts of exhibiting and revealing an observation objective, hitting an observation objective by a test case, the objective verdicts **hit** and **miss** and the relation between conforming and revealing implementations are discussed. Subsequently, we instantiate these concepts within the **ioco**-based, transition system conformance test theory of [Tre96]. First, Section 4 recalls the basics of this theory. Then Section 5 develops **ioco**-based observation objectives and **ioco**-based exhibition. This leads to a test derivation algorithm which derives, from a given specification and an observation objective, an **ioco**-sound test case with the ability to hit that observation objective. Finally, Section 6 gives some concluding remarks, including a brief discussion about putting the observation objective ideas into practice.

2 Formal Framework for Testing

This section presents the formal framework for conformance testing [BAL$^+$90, ISO96, Tre99] which forms the basis for the formalization of observation objectives in the next section. The presentation in this section is taken from [Tre99]. The presented framework can be used for testing of an implementation with respect to a formal specification of its functional behaviour. It introduces, at a high level of abstraction, the concepts used in a formal conformance testing process and it defines a structure which allows to reason about testing in a formal way. The most important part of this is to link the informal world of implementations, tests and experiments with the formal world of specifications and models. To this extent the framework introduces the concepts of conformance, i.e., functional correctness, testing, observations, sound and exhaustive test suites, and test derivation.

Conformance For talking about conformance we need implementations and specifications. The specifications are formal, so a universe of formal specifications denoted *SPECS* is assumed. Implementations are the systems that we are going to test, henceforth they will be called IUT, implementation under test, and the class of all IUT's is denoted by *IMPS*. So, conformance could be introduced by having a relation $\textbf{conforms}-\textbf{to} \subseteq \mathit{IMPS} \times \mathit{SPECS}$ with IUT $\textbf{conforms}-\textbf{to}\ s$ expressing that IUT is a correct implementation of specification s.

However, unlike specifications, implementations under test are real, physical objects, such as pieces of hardware or software; they are treated as black boxes exhibiting behaviour and interacting with their environment, but not amenable to formal reasoning. This makes it difficult to give a formal definition of **conforms−to** which should be our aim in a formal testing framework. In order to reason formally about implementations, we make the assumption that any real implementation $\text{IUT} \in \mathit{IMPS}$ can be modeled by a formal object $i_{\text{IUT}} \in \mathit{MODS}$, where *MODS* is referred to as the universe of models. This assumption is referred to as the *test hypothesis* [Ber91]. Note that the test hypothesis only assumes that a model i_{IUT} exists, but not that it is known a priori.

Thus the test hypothesis allows to reason about implementations as if they were formal objects, and, consequently, to express conformance by a formal relation between models of implementations and specifications. Such a relation is called an *implementation relation* $\textbf{imp} \subseteq \mathit{MODS} \times \mathit{SPECS}$ [BAL$^+$90, ISO96]. Implementation $\text{IUT} \in \mathit{IMPS}$ is said to be correct with respect to $s \in \mathit{SPECS}$, IUT $\textbf{conforms}-\textbf{to}\ s$, if and only if the model $i_{\text{IUT}} \in \mathit{MODS}$ of IUT is **imp**-related to s: $i_{\text{IUT}}\ \textbf{imp}\ s$.

Observation and testing The behaviour of an implementation under test is investigated by performing experiments on the implementation and observing the reactions that the implementation produces to these experiments. The specification of such an experiment is called a *test case*, and the process of applying a test to an implementation under test is called *test execution.*

Let test cases be formally expressed as elements of a domain TESTS. Then test execution requires an operational procedure to execute and apply a test case $t \in \mathit{TESTS}$ to an implementation under test IUT $\in \mathit{IMPS}$. This operational procedure is denoted by $\mathit{exec}(t, \mathrm{IUT})$. During test execution a number of observations will be made, e.g., occurring events will be logged, or the response of the implementation to a particular stimulus will be recorded. Let (the formal interpretation of) these observations be given in a domain of observations OBS, then test execution $\mathit{exec}(t, \mathrm{IUT})$ will lead to a subset of OBS. Note that *exec* is not a formal concept; it captures the action of "pushing the button" to let t run with IUT. Also note that $\mathit{exec}(t, \mathrm{IUT})$ may involve multiple runs of t and IUT, e.g., in case nondeterminism is involved.

Again, since $\mathit{exec}(t, \mathrm{IUT})$ corresponds to the physical execution of a test case, we have to model this process of test execution in our formal domain to allow formal reasoning about it. This is done by introducing an observation function $\mathit{obs} : \mathit{TESTS} \times \mathit{MODS} \to \mathcal{P}(\mathit{OBS})$. So, $\mathit{obs}(t, i_{\mathrm{IUT}})$ formally models the real test execution $\mathit{exec}(t, \mathrm{IUT})$.

In the context of an *observational framework* consisting of TESTS, OBS, *exec* and *obs*, it can now be stated more precisely what is meant by the test hypothesis:

$$\forall \mathrm{IUT} \in \mathit{IMPS}\ \exists i_{\mathrm{IUT}} \in \mathit{MODS}\ \forall t \in \mathit{TESTS} :\ \mathit{exec}(t, \mathrm{IUT}) = \mathit{obs}(t, i_{\mathrm{IUT}}) \tag{1}$$

This could be paraphrased as follows: for all real implementations that we are testing, it is assumed that there is a model, such that if we would put the IUT and the model in black boxes and would perform all possible experiments defined in TESTS, then we would not be able to distinguish between the real IUT and the model. Actually, this notion of testing is analogous to the ideas underlying testing equivalences [DNH84, DN87].

Usually, we like to interpret observations of test execution in terms of being right or wrong. So we introduce a family of *verdict functions* $\mathit{verd}_t : \mathcal{P}(\mathit{OBS}) \to \{\mathbf{fail}, \mathbf{pass}\}$ which allows to introduce the following abbreviation:

$$\mathrm{IUT}\ \mathbf{passes}\ t \quad =_{\mathrm{def}} \quad \mathit{verd}_t(\mathit{exec}(t, \mathrm{IUT})) = \mathbf{pass} \tag{2}$$

This is easily extended to a *test suite* $T \subseteq \mathit{TESTS}$: IUT **passes** $T \Leftrightarrow \forall t \in T$: IUT **passes** t. Moreover, an implementation fails test suite T if it does not pass: IUT **fails** $T \Leftrightarrow$ IUT $\mathbf{pas\not{s}es}$ T.

Conformance testing Conformance testing involves assessing, by means of testing, whether an implementation conforms, with respect to implementation relation **imp**, to its specification. Hence, the notions of conformance, expressed by **imp**, and of test execution, expressed by *exec*, have to be linked in such a way that from test execution an indication about conformance is obtained. So, ideally, we would like to have a test suite T_s such that for a given specification s

$$\mathrm{IUT}\ \mathbf{conforms\!-\!to}\ s \iff \mathrm{IUT}\ \mathbf{passes}\ T_s \tag{3}$$

A test suite with this property is called *complete*; it can distinguish exactly between all conforming and non-conforming implementations. Unfortunately, this is a very strong requirement for practical testing: complete test suites are usually infinite, and consequently not practically executable. Hence, usually a weaker requirement on test suites is posed: they should be *sound*, which means that all correct implementations, and possibly some incorrect implementations, will pass them; or, in other words, any detected erroneous implementation is indeed non-conforming, but not the other way around. Soundness corresponds to the left-to-right implication in (3). The right-to-left implication is called *exhaustiveness*; it means that all non-conforming implementations will be detected.

To show soundness (or exhaustiveness) for a particular test suite we have to use the formal models of implementations and test execution:

$$\forall i \in \mathit{MODS} :\ (\ i\ \mathbf{imp}\ s \iff \forall t \in T :\ \mathit{verd}_t(\mathit{obs}(t, i)) = \mathbf{pass}\ ) \tag{4}$$

Once (4) has been shown it follows that:

$$\begin{array}{ll}
 & \text{IUT } \mathbf{passes}\ T \\
\text{iff} & (*\ \text{definition } \mathbf{passes}\ T\ \ *) \\
 & \forall t \in T:\ \text{IUT } \mathbf{passes}\ t \\
\text{iff} & (*\ \text{definition } \mathbf{passes}\ t\ \ *) \\
 & \forall t \in T:\ \mathit{verd}_t(\mathit{exec}(t, \text{IUT})) = \mathbf{pass} \\
\text{iff} & (*\ \text{test hypothesis (1)}\ \ *) \\
 & \forall t \in T:\ \mathit{verd}_t(\mathit{obs}(t, i_{\text{IUT}})) = \mathbf{pass} \\
\text{iff} & (*\ \text{completeness on models (4) applied to } i_{\text{IUT}}\ \ *) \\
 & i_{\text{IUT}}\ \mathbf{imp}\ s \\
\text{iff} & (*\ \text{definition of conformance}\ \ *) \\
 & \text{IUT } \mathbf{conforms{-}to}\ s
\end{array}$$

So, if the completeness property has been proved on the level of models and if there is ground to assume that the test hypothesis holds, then conformance of an implementation with respect to its specification can be decided by means of a testing procedure.

3 Observation Objectives in the Formal Framework

Our aim in this section is to extend the framework of the previous section with *observation objectives.*

An observation objective describes the observations that we wish to see from the implementation during the test. Whether an implementation is able to show these observations is called *exhibition*: an implementation *exhibits* an observation objective if it has the possibility to show the observations described in the observation objective. In the next paragraph this concept of exhibition is elaborated completely analogous to the concept of conformance in the previous section.

The next task is to test for exhibition. Given an observation objective (and a specification), a test case should be derived such that it allows the implementation to show the observations described in the observation objective. Preferably, the derived test case should be such that all and only all implementations that have the possibility to show the observations will indeed do so. Analogous to completeness in the previous section this will be called *e-completeness.*

Finally, in the last paragraph, the orthogonality between conformance and exhibition is further elaborated.

Exhibition and observation First we introduce the formal notion of a *observation objective.* An observation objective is a formal specification of the behaviour that we would like to observe explicitly during testing. The universe of observation objectives is called TOBS. We introduce a relation that relates an observation objective with all implementations that are able to *exhibit* that observation objective. Exhibition of an observation objective by an implementation expresses that an implementation can possibly manifest the behaviour specified by the observation objective. When we execute a well chosen set of experiments, the implementation will show behaviour that is specified by the observation objective. The notion of showing the specified behaviour is determined by this relation. We call the relation $\mathbf{exhibits} \subseteq \mathit{IMPS} \times \mathit{TOBS}$.

To formally reason about exhibition, we have to link the informal, experimental world and the formal world. Analogous to the definition of **conforms−to**, we need a relation of **exhibits** in the formal domain in order to reason about exhibition of an observation objective and to decide whether experiments would be able to challenge an implementation to exhibit the requested observations. We make again the assumption of the test hypothesis. We call this relation the *reveal relation* $\mathbf{rev} \subseteq \mathit{MODS} \times \mathit{TOBS}$. So the implementation IUT $\in \mathit{IMPS}$ satisfies an observation objective $e \in \mathit{TOBS}$, if and only if the model $i_{\text{IUT}} \in \mathit{MODS}$ of IUT is **rev**-related to e: $i_{\text{IUT}}\ \mathbf{rev}\ e$.

To decide whether an implementation exhibits an observation objective, we use testing and interpret the gained observations obtained by the experiments. Recalling the introduced observation and testing framework and its formalisms, we use a verdict function that relates the observations with respect to an observation objective resulting in a verdict whether we have seen evidence of exhibition. We call such a function the **hit**-function $H_e : \mathcal{P}(OBS) \rightarrow \{\mathbf{hit}, \mathbf{miss}\}$. Here **hit** is the verdict that expresses that we have found such evidence during experimenting. We introduce the following abbreviation, where t_e is a test case related to an observation objective e, i.e. developed based on e.

$$IUT \ \mathbf{hits}\ e\ \mathbf{by}\ t_e \quad =_{\text{def}} \quad H_e(exec(t_e, IUT)) = \mathbf{hit}$$

This is extended to a test suite T_e and we abbreviate IUT **hits** e **by** T_e $=_{\text{def}}$ $H_e(\bigcup\{exec(t, IUT) \mid t \in T_e\}) = \mathbf{hit}$. Note that this abbreviation differs from the IUT **passes** T abbreviation. An implementation misses a test suite T_e if it does not hit: IUT **misses** e **by** T_e $=_{\text{def}}$ $\neg$ (IUT **hits** e **by** T_e).

Exhibition testing Exhibition testing is the process of deciding whether an implementation is **rev**-related to its observation objective, by means of experimentation. We need to relate the test execution and **rev**, aiming at getting a verdict from the test execution such that we obtain an indication about exhibition of the observation objective. Ideally we would like to have a test suite T_e such that:

$$\text{IUT } \mathbf{exhibits}\ e \iff \text{IUT } \mathbf{hits}\ e\ \mathbf{by}\ T_e \tag{5}$$

Analogous to conformance testing, we can introduce notions of completeness, exhaustiveness and soundness. We call a test suite of Equation 5 *e-complete*. Such a test suite can distinguish among all exhibiting and non-exhibiting implementations. A test suite is *e-exhaustive* when it only can detect non-exhibiting implementations (IUT **exhibits** e implies IUT **hits** e **by** T_e). A test suite is *e-sound* when it only can detect exhibiting implementations (IUT **exhibits** e if IUT **hits** e **by** T_e).

To reason whether a test suite is able to challenge an implementation to exhibit an observation objective and to detect all exhibiting implementations we have to show e-soundness and e-completeness of such a test suite. For this we use the formal models of implementation and test execution:

$$\forall i \in MODS : (\ \ i\ \mathbf{rev}\ e \quad \text{iff} \quad H_e(\bigcup\{obs(t_e, i) \mid t_e \in T_e\}) = \mathbf{hit}\ \ ) \tag{6}$$

Once (6) has been shown it follows that:

$$\begin{array}{ll}
 & \text{IUT } \mathbf{hits}\ e\ \mathbf{by}\ T_e \\
\text{iff} & (*\ \text{abbreviation } \mathbf{hits}\ e\ \mathbf{by}\ T_e\ *) \\
 & H_e(\bigcup\{exec(t_e, \text{IUT}) \mid t_e \in T_e\}) = \mathbf{hit} \\
\text{iff} & (*\ \text{test hypothesis (1)}\ *) \\
 & H_e(\bigcup\{obs(t_e, i_{\text{IUT}}) \mid t_e \in T_e\}) = \mathbf{hit} \\
\text{iff} & (*\ \text{e-completeness on models (6)}\ *) \\
 & i_{\text{IUT}}\ \mathbf{rev}\ e \\
\text{iff} & (*\ \text{definition of exhibition}\ *) \\
 & \text{IUT } \mathbf{exhibits}\ e
\end{array}$$

We can now decide whether an implementation exhibits an observation objective by means of testing, provided that this completeness property has been proven and it is plausible that the test hypothesis holds.

Combining conformance and exhibition We return to the test selection problem, where we combine exhibition and conformance for the derivation of a test suite. The observation objective is used as a criterion for the selection: we aim at obtaining a test suite that is e-complete and sound. We want e-soundness so that we can conclude that an implementation exhibits, based on the test results. Furthermore we require e-exhaustiveness, because we want to be able to find all implementations that are able to exhibit. The soundness property provides us with the ability to detect erroneous implementations. So the test derivation process can be reformulated as given an observation objective e and a specification s, find a test suite $T_{s,e}$ such that $T_{s,e}$ is sound with respect to **imp** and $T_{s,e}$ is e-complete with respect to **rev**.

When we apply observation objectives for conformance testing, it appears that not every observation objective is feasible. As said, we use the test results of exhibition to support the confidence of correctness of an implementation. Ideally, we try to detect by experimentation all implementations IUT that **pass** and **hit**, i.e. for some $T_{s,e}$: $\{\text{IUT} \in \mathit{IMPS} \mid \text{IUT } \textbf{passes } T_{s,e}$ and IUT $\textbf{hits } e \textbf{ by } T_{s,e}\}$. This gives some restrictions on the specification of an observation objective. We study this in the domain of formal models of the implementations.

Consider Figure 1. Let i **imp** s be the set with all conforming models of implementations, and i **passes** T, the models that pass a sound test suite of T. The set i **rev** e is the set of models that exhibit observation objective e. Notice that this set corresponds to the set of implementations IUT **hits** e **by** T_e with T_e being an e-complete test suite. We distinguish four different scenarios of intersection of i **rev** e and i **passes** T. These are visualized in Figure 1a-d.

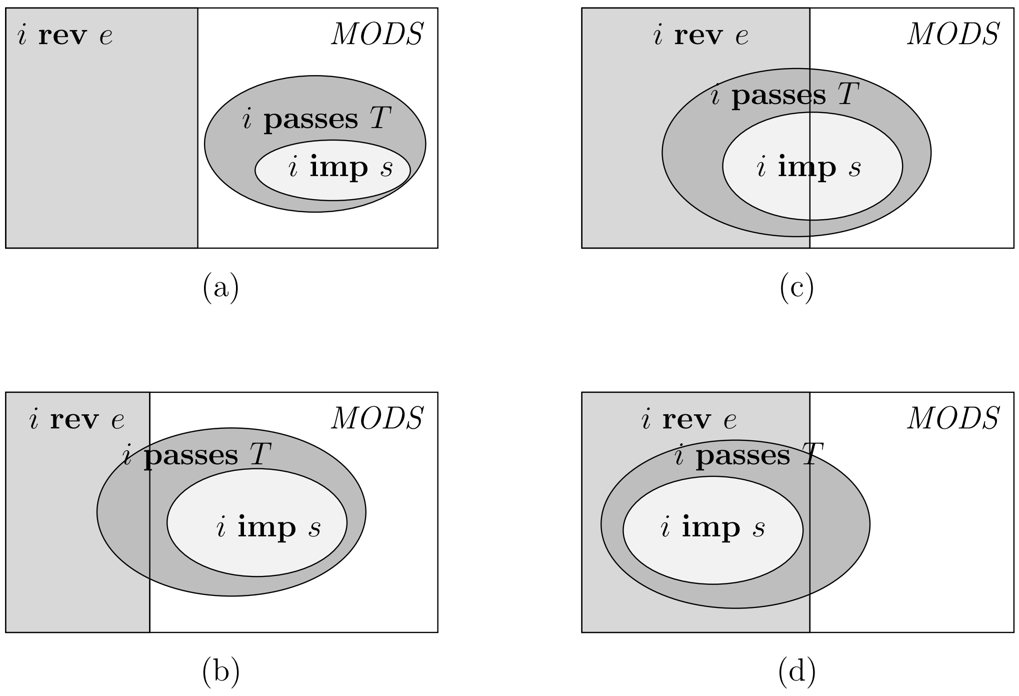


Figure 1: Scenarios of choosing an observation objective

Figure 1(a) visualizes the empty intersection. This means that there exists no implementation that exhibits and that will be passed as a correct implementation. A test suite for this situation is infeasible, since for all exhibiting implementations, we will get a **fail** verdict during execution of the test suite. In Figure 1(b), we have chosen a test suite where we detect (following a (**pass**, **hit**) verdict) implementations that exhibit but that are incorrect (not in i **imp** s). This situation is not desirable. Fortunately, we can check during test derivation whether these situations (a) and (b) occur (based on our formal models and formal relations), and decide not to derive such test suites.

The ideal situation is depicted in Figure 1(d) (this might also include the situation i **rev** $e \supseteq i$ **passes** T). Here all correct implementations also exhibit. However, this would in general make it very complicated to specify an observation objective. It would imply that observation objectives would be restricted to the ones with the property $\{i \in \mathit{MODS} \mid i \ \mathbf{rev}\ e\} \supseteq \{i \in \mathit{MODS} \mid i \ \mathbf{imp}\ s\}$. This is hard to prove taking all nondeterminism and complexity of a specification into account. So we loosen such a requirement and in practice we try to develop a test suite that is visualized in Figure 1(c). We have the most confidence in the correctness of an implementation that conforms and exhibits (verdict $(\mathbf{pass}, \mathbf{hit})$). However, we still might find an implementation that is correct, but does not exhibit (which corresponds to the notion of the **inconclusive** verdict [ISO91]), or an implementation that is incorrect and exhibits.

Summarizing all, we can formulate our practical approach to conformance testing using exhibition. We choose an observation objective e such that:

$$\{i \mid i \ \mathbf{rev}\ e\} \cap \{i \mid i \ \mathbf{imp}\ s\} \neq \emptyset \tag{7}$$

From such an observation objective and formal specification we derive a test suite $T_{s,e}$ that is e-complete and sound. After execution of $T_{s,e}$ we reject an implementation that gives a **fail** verdict. We have an increased confidence in the correctness when the test results evaluate to a $(\mathbf{pass}, \mathbf{hit})$ verdict. An experiment that evaluates to a $(\mathbf{pass}, \mathbf{miss})$ verdict has not found any evidence of non conformance of the implementation, but has also not detected any behaviour that supports our confidence in the correctness of the implementation.

4 Testing with ioco

We will instantiate the discussed framework with the **ioco** based conformance testing theory, c.f. [Tre96]. In this section we recall the basics of the **ioco** theory and its related notation.

Labelled transition systems Labelled transition systems provide a formalism to specify, model, analyse and reason about system behaviour. A labelled transition system description is defined in terms of states and labelled transitions between states.

Definition 4.1
A *labelled transition system* is a 4-tuple $\langle S, L, T, s_0 \rangle$ where

- S is a non-empty set of *states*;
- L is a finite set of *labels*;
- $T \subseteq S \times (L \cup \{\tau\}) \times S$ is a set of triples, the *transition relation*;
- $s_0 \in S$ is the *initial state*.

$\square$

The labels in L represent the observable interactions of a system. The special label $\tau \notin L$ represents an unobservable, internal action. We denote the class of all labelled transition systems over L by $\mathcal{LTS}(L)$. Transition systems without infinite paths of transitions with only internal actions are called *strongly converging*. For technical reasons we restrict $\mathcal{LTS}(L)$ to strongly converging transition systems.

A *trace* is a finite sequence of observable actions. The set of all traces over L is denoted by L^*, with ϵ denoting the empty sequence. If $\sigma_1, \sigma_2 \in L^*$, then $\sigma_1 \cdot \sigma_2$ is the concatenation of σ_1 and σ_2. Some additional notations and properties are introduced in Definitions 4.2 and 4.3.

Definition 4.2
Let $p = \langle S, L, T, s_0 \rangle$ be a labelled transition system with $s, s' \in S$, and let $\mu_{(i)} \in L \cup \{\tau\}$, $a_{(i)} \in L$,

$\sigma \in L^*$, and $n \geq 0$.

$$\begin{array}{lll}
s \xrightarrow{\epsilon} s' & =_{\text{def}} & s = s' \\
s \xrightarrow{\mu} s' & =_{\text{def}} & (s, \mu, s') \in T \\
s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} s' & =_{\text{def}} & \exists s_0, \ldots, s_n : \ s = s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} s_n = s' \\
s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} & =_{\text{def}} & \exists s' : \ s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} s' \\
s \not\xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} & =_{\text{def}} & \text{not } \exists s' : \ s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} s' \\
s \overset{\epsilon}{\Longrightarrow} s' & =_{\text{def}} & s \xrightarrow{\tau \cdot \ldots \cdot \tau} s' \\
s \overset{a}{\Longrightarrow} s' & =_{\text{def}} & \exists s_1, s_2 : \ s \overset{\epsilon}{\Longrightarrow} s_1 \xrightarrow{a} s_2 \overset{\epsilon}{\Longrightarrow} s' \\
s \xRightarrow{a_1 \cdot \ldots \cdot a_n} s' & =_{\text{def}} & \exists s_0 \ldots s_n : \ s = s_0 \overset{a_1}{\Longrightarrow} s_1 \overset{a_2}{\Longrightarrow} \ldots \overset{a_n}{\Longrightarrow} s_n = s' \\
s \overset{\sigma}{\Longrightarrow} & =_{\text{def}} & \exists s' : \ s \overset{\sigma}{\Longrightarrow} s' \\
s \overset{\sigma}{\not\Longrightarrow} & =_{\text{def}} & \neg\ \exists s' : s \overset{\sigma}{\Longrightarrow} s'
\end{array}$$

□

We will not always distinguish between a labelled transition system and its initial state: if $p = \langle S, L, T, s_0 \rangle$, then we will identify the labelled transition system p with its initial state s_0, and we write, for example, $p \overset{\sigma}{\Longrightarrow}$ instead of $s_0 \overset{\sigma}{\Longrightarrow}$.

Definition 4.3
Let p be a (state of a) labelled transition system and let P be a set of states.

1. $\mathit{init}(p) \ =_{\text{def}} \ \{ \mu \in L \cup \{\tau\} \mid p \xrightarrow{\mu} \ \}$
2. $\mathit{init}(P) \ =_{\text{def}} \ \bigcup \ \{ \mathit{init}(p) \mid p \in P \ \}$
3. $\mathit{traces}(p) \ =_{\text{def}} \ \{ \sigma \in L^* \mid p \overset{\sigma}{\Longrightarrow} \ \}$
4. $p \,\textbf{after}\, \sigma \ =_{\text{def}} \ \{ p' \mid p \overset{\sigma}{\Longrightarrow} p' \ \}$
5. $P \,\textbf{after}\, \sigma \ =_{\text{def}} \ \bigcup \ \{ \ p \,\textbf{after}\, \sigma \ \mid p \in P \ \}$

□

A labelled transition system can also be denoted by a process-algebraic behaviour expression. We introduce a (for this paper) limited set with syntax:

$$B \quad =_{\text{def}} \quad \textbf{stop} \mid \alpha; B \mid B + B \mid \Sigma \mathcal{B}$$

Here $\alpha \in L$ and $\mathcal{B}$ is a countable set of behaviour expressions. The semantics are defined by the following inference rules and axioms:

$$\begin{array}{rcl}
 & \vdash & \textbf{stop} \not\xrightarrow{\alpha}, \alpha \in L \cup \{\tau\} \\
 & \vdash & \alpha; B \xrightarrow{\alpha} B \\
B_1 \xrightarrow{\alpha} B_1', \alpha \in L & \vdash & B_1 + B_2 \xrightarrow{\alpha} B_1' \\
B_2 \xrightarrow{\alpha} B_2', \alpha \in L & \vdash & B_1 + B_2 \xrightarrow{\alpha} B_2' \\
B \xrightarrow{\alpha} B', B \in \mathcal{B}, \alpha \in L & \vdash & \Sigma \mathcal{B} \xrightarrow{\alpha} B'
\end{array}$$

Input-output transition systems We assume that the label set L can be partitioned into input actions L_I and output actions L_U: $L = L_I \cup L_U$ and $L_I \cap L_U = \emptyset$. Moreover, we consider systems which always accept any input. In terms of transition systems: all inputs, i.e., all actions in L_I, are enabled in any reachable state of the transition system. Such transition systems are called *input-output transition systems.* In input-output transition systems, inputs of one system communicate with the outputs of the other system, and vice versa, (cf. IOA [LT89]).

Definition 4.4
An *input-output transition system* p is a labelled transition system in which the set of actions L is partitioned into input actions L_I and output actions L_U ($L_I \cup L_U = L, L_I \cap L_U = \emptyset$), and for which all inputs are enabled in any reachable state:

$$\text{whenever} \quad p \overset{\sigma}{\Longrightarrow} p' \quad \text{then} \quad \forall a \in L_I : \ p' \overset{a}{\Longrightarrow}$$

The class of input-output transition systems with input actions in L_I and output actions in L_U is denoted by $\mathcal{IOTS}(L_I, L_U) \subseteq \mathcal{LTS}(L_I \cup L_U)$.

□

Implementation relation We will use the relation **ioco** as implementation relation. This relation assumes that the specification is expressed as a labelled transition system in which inputs and outputs can be distinguished (not necessarily $\mathcal{IOTS}$), and that the implementation behaves as, i.e., can be modeled by, an input-output transition system (cf. *test hypothesis* Section 2): $\mathbf{ioco} \subseteq \mathcal{IOTS}(L_I, L_U) \times \mathcal{LTS}(L_I \cup L_U)$.

An implementation $i \in \mathcal{IOTS}(L_I, L_U)$ is **ioco**-correct with respect to the specification $s \in \mathcal{LTS}(L_I \cup L_U)$ if i can never produce an output which could not have been produced by s in the same situation, i.e., after the same specification trace. Moreover, i may only stay silent, i.e., produce no output at all, if s can do so. The absence of outputs is called *quiescence* and is denoted by a special label δ ($\delta \notin L \cup \{\tau\}$), cf. [Vaa91].

To formalize this notion of conformance **ioco** we first define quiescence as the absence of outputs. Then we extend traces of actions with the special action δ. Occurrence of δ in a state p, denoted by $p \xrightarrow{\delta}$, expresses that state p cannot produce any output. Since no 'normal' action in p is executed in that case, p cannot move to another state, so always $p \xrightarrow{\delta} p'$ implies $p' = p$. Traces in which both normal actions in L and the special action δ may occur are called *suspension traces*. To denote suspension traces the notations $\xrightarrow{\mu}$ and $\overset{\sigma}{\Longrightarrow}$ (Definitions 4.1, 4.2 and 4.3) are extended to $L_\delta = L \cup \{\delta\}$, and to traces in L_δ^*, respectively. Note that this overlapping of notation does not introduce conflicts.

Definition 4.5
Let $p \in \mathcal{LTS}(L_I \cup L_U)$.

1. A state q of p is *quiescent*, denoted by $q \xrightarrow{\delta} q$, if $\forall \mu \in L_U \cup \{\tau\} : q \overset{\mu}{\not\longrightarrow}$
2. The *suspension traces* of $p \in \mathcal{LTS}(L_I \cup L_U)$ are:
   $Straces(p) =_{\text{def}} \{ \sigma \in (L \cup \{\delta\})^* \mid p \overset{\sigma}{\Longrightarrow} \}$,
   where $\overset{\sigma}{\Longrightarrow}$ is as in Definition 4.2 extended with δ-transitions $q \xrightarrow{\delta} q$.

□

We can now define the possible outputs $out(\, p \,\mathbf{after}\, \sigma\,)$ of a process p after a suspension trace σ. The action δ may occur in $out(\, p \,\mathbf{after}\, \sigma\,)$ as a special action indicating that after σ it is possible to observe no outputs at all, i.e., quiescence. Using $out(\, p \,\mathbf{after}\, \sigma\,)$ the definition of **ioco** is now straightforward by requiring that after any suspension trace of the specification any possible output of the implementation should be a possible output of the specification.

Definition 4.6
Let p be a (state of a) labelled transition system, and let P be a set of states; let $i \in \mathcal{IOTS}(L_I, L_U)$ and $s \in \mathcal{LTS}(L_I \cup L_U)$, then

1. $out(p) =_{\text{def}} \{ x \in L_U \cup \{\delta\} \mid p \xrightarrow{x} \}$
2. $out(P) =_{\text{def}} \bigcup \{ out(p) \mid p \in P \}$
3. $i \;\mathbf{ioco}\; s =_{\text{def}} \forall \sigma \in Straces(s) : out(\, i \,\mathbf{after}\, \sigma\,) \subseteq out(\, s \,\mathbf{after}\, \sigma\,)$

□

Test generation In order to present the test derivation algorithm we first define a test case and a test suite. We use the special label $\theta \notin L_I \cup L_U$ to represent the observation of quiescence. Furthermore $L_\theta = L \cup \{\theta\}$. The notation $\overline{\sigma}$ represents replacement of δ-actions by θ-actions and vice versa, i.e. the action that models quiescence is replaced by the action that observes quiescence.

Definition 4.7

1. A *test case* t is a labelled transition system $\langle S, L_I \cup L_U \cup \{\theta\}, T, s_0 \rangle$ such that
   - t is deterministic and has finite behaviour;
   - S contains the terminal states **pass** and **fail**, with $init(\mathbf{pass}) = init(\mathbf{fail}) = \emptyset$;
   - for any state $t' \in S$ of t, $t' \neq \mathbf{pass}, \mathbf{fail}$: $init(t') = \{a\}$ for some $a \in L_I$ or $init(t') = L_U \cup \{\theta\}$.

   The universe of all test cases of L_I and L_U is denoted as $TEST(L_U, L_I)$.
2. A *test suite* T is a set of test cases: $T \subseteq TEST(L_U, L_I)$.

□

Algorithm 4.8
Let s be a specification with initial state s_0. Let S be a non-empty set of states, with initially $S = s_0$ **after** ϵ. Then a test case t is obtained from S by a finite number of recursive applications of one of the following three cases:

1. (∗ terminate the test case ∗)
   t := **pass**
2. (∗ give a next input to the implementation ∗)
   $t := a\,;\,t'$, if $a \in \mathit{init}(S)$
   where $a \in L_I$, and t' is obtained by recursively applying the algorithm for $S' := S$ **after** a.
3. (∗ check the next output of the implementation ∗)
   $t := \quad \Sigma\{\overline{\alpha};\textbf{fail} \mid \alpha \in L_U \cup \{\delta\} \text{ and } \alpha \notin \mathit{out}(S)\}$
   $\quad + \quad \Sigma\{\overline{\alpha}; t_\alpha \mid \alpha \in \mathit{out}(S)\}$
   where t_α is obtained by recursively applying the algorithm for $S' := S$ **after** α.

□

This algorithm was proved in [Tre96] to produce only sound test cases, i.e., test cases which never produce **fail** while testing an **ioco**-conforming implementation. Moreover, it was shown that any non-conforming implementation can always be detected by a test case generated with this algorithm. For more details about the relation **ioco**, for a rationale for its use, and for more generic definitions we refer to [Tre96].

5 Observation Objectives in IOTS

In this section we instantiate the framework presented in Section 3 with the **ioco** theory of Section 4. We start with an example to clarify the introduced concepts.

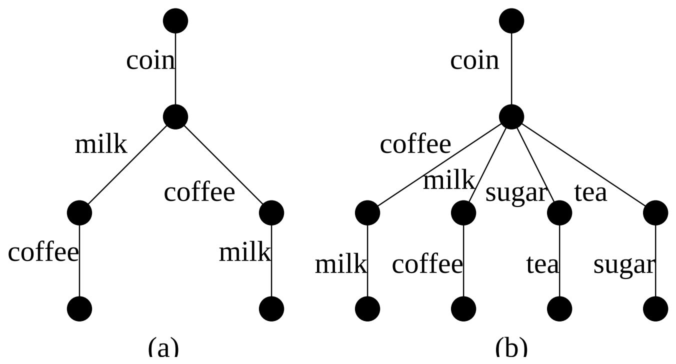


Figure 2: Coffee machines

Example 5.1
Consider the specification s of a simple coffee machine in Figure 2(a), with $L_I = \{\text{coin}\}$ and $L_U = \{\text{milk}, \text{coffee}\}$. This coffee machine provides a user with coffee with milk after the insertion of a coin. We want to test an implementation for conformance. However, we will not do this exhaustively, but we are interested in only those experiments that will challenge the system to provide coffee. If we find during testing no evidence of non conformance and we will observe coffee, than we assume that the implementation is correct. The specification of the coffee machine describes two traces of interactions that let the machine provide coffee. These two traces are captured in the observation objective, since during the experiments we might observe these sequences of interactions. So we choose the formal observation objective e={coin·coffee·milk, coin·milk·coffee}. Furthermore we define the **hit**-function as $H_e(O) = \textbf{hit}$ iff $O \cap e \neq \emptyset$; if we observe any test run that is included in the observation objective, we conclude that the system exhibits. Note that here

TOBS is the power set of a set of traces of the specification with label set $L = \{\text{coffee,coin,milk}\}$ ($TOBS = \mathcal{P}(L^*)$). Now let $T_{s,e}$ be a sound en e-complete test suite for s and e. When during testing using test suite $T_{s,e}$, we observe either coin·coffee·milk or coin·milk·coffee as a test run we conclude that IUT **exhibits** e.

□

Reveal relations We instantiate the presented framework of Section 3 with the **ioco** theory. We inherit $TESTS = TEST(L_U, L_I)$, $OBS = (L_I \cup L_U \cup \{\theta\})^*$ and $MODS = \mathcal{IOTS}(L_I, L_U)$. We first consider *singular observation objectives.* A singular observation objective can be exhibited by one observation of a test case execution. The used observation objective in Example 5.1 is a singular observation objective. For singular observation objectives it would be sufficient to specify it by a set of traces which consists of all potential observations that will lead to exhibition during testing. We take as the specification of the observation objective a set of traces from L_δ^*, i.e. the counterpart of a subset of *OBS* (δ versus θ). So $TOBS = \mathcal{P}(L_\delta^*)$ and we instantiate $\mathbf{rev} \subseteq \mathcal{IOTS}(L_I, L_U) \times \mathcal{P}((L_I \cup L_U \cup \{\delta\})^*)$. We define the *reveal input output singular* relation **rios** for the singular case. This relation relates formally all models of implementations that are potentially able to exhibit a singular test objective to that observation objective. A model of an implementation exhibits the singular test objective if one of its suspension traces is an element of the observation objective. So a **hit**-function H_e for a singular observation objective evaluates to **hit** if one of the prefixes of an observation (test run, i.e. trace from L_θ^*) is included in the observation objective.

Definition 5.2
Let $i \in \mathcal{IOTS}(L_I, L_U)$ be an implementation, $O \subseteq L_\theta^*$ a set of observations and $e \subseteq L_\delta^*$ a singular observation objective, then

1. $\mathbf{prefix}(O) = \{\sigma_1 \mid \sigma_1 \cdot \sigma_2 \in O \text{ with } \sigma_2 \in L_\theta^*\}$
2. $i\ \mathbf{rios}\ e \quad =_{\text{def}} \quad Straces(i) \cap e \neq \emptyset$
3. $H_e^{\mathbf{rios}}(O) \ = \ \mathbf{hit} \quad =_{\text{def}} \quad \mathbf{prefix}(O) \cap e \neq \emptyset$

□

For example, reconsider Figure 2(a) as an implementation i after making it input complete. Then i **rios** { coin · milk, coin · coffee · milk } holds. This observation objective insists on the observation of milk.

With a singular observation objective we can only require the exhibition of one trace from that observation objective. This is a limited expressiveness of an observation objective. We want to specify more than one trace that should be exhibited during the experiment. We can do this with *plural observation objectives.* A plural observation objective is composed out of singular observation objectives, which all should be individually exhibited during the execution of the test suite in order to satisfy the composed (plural) observation objective. The following example shows an application of this.

Example 5.3
Consider the coffee machine in Figure 2(b), with $L_I = \{\text{coin}\}$ and $L_U = \{\text{milk, coffee, tea, sugar}\}$. During testing we want to observe evidence that the machine can deliver coffee with milk and tea with sugar. We specify this by a plural observation objective and choose the set of singular observation objects $E = \{\{$ coin · coffee · milk, coin · milk · coffee $\}$, $\{$ coin · tea · sugar, coin · sugar · tea $\}\}$. The **hit**-function is defined by $H_E(O) = \mathbf{hit}$ iff $\forall e \in E : O \cap e \neq \emptyset$.

□

In **ioco** terms a plural observation objective is a set which is an element of $TOBS = \mathcal{P}(\mathcal{P}(L_\delta^*))$ and the reveal relation $\mathbf{rev} \subseteq \mathcal{IOTS}(L_I, L_U) \times \mathcal{P}(\mathcal{P}(L_\delta^*))$. The exhibition of a plural test objective E requires in general more than one observation during testing, since seldom $\bigcap\{e \mid e \in E\} \neq \emptyset$, which can be satisfied by just one observation. We can, analogously to, the singular case define the *reveal input out plural relation* **riop** and the **hit**-function $H_E^{\mathbf{riop}}(O)$.

Definition 5.4
Let $i \in \mathcal{IOTS}(L_I, L_U)$ be an implementation, $O \subseteq L_\theta^*$ a set of observations and $E \subseteq \mathcal{P}(L_\delta^*)$ a plural observation objective then:

1. i **riop** $E \quad =_{\text{def}} \quad \forall e \in E : i$ **rios** e
2. $H_E^{\textbf{riop}}(O) \;=\;$ **hit** iff $\;\forall e \in E : H_e^{\textbf{rios}}(O)$

□

For example, consider Figure 2b as an implementation i after making it input complete. Then i **riop** {{ coin · milk, coin · coffee · milk }, { coin · tea, coin · sugar · tea }} holds. This plural observation objective insists on the observation of milk and tea.

Test generation We first consider the singular case. We have a formal definition of the **rev** relation and of the **imp**-relation, which are respectively **rios** and **ioco**. We want to generate a test suite that is sound and e-complete. Furthermore we restrict our observation objective such that $\{i \in \mathit{MODS} \mid i$ **rios** $e\} \;\cap\; \{i \in \mathit{MODS} \mid i$ **ioco** $s\} \neq \emptyset$ (c.f. Section 3 Equation 7). Proposition 5.5 shows that it is sufficient to choose the test objective as a subset of the suspension traces of the specification to satisfy this constraint.

Proposition 5.5
Let $s \in \mathcal{LTS}(L_I \cup L_U)$ be a specification and let $e \in \mathcal{P}(L_\delta^*)$ be an observation objective.

$$\text{if } e \subseteq \mathit{Straces}(s) \text{ then } \{i \in \mathcal{IOTS}(L_I, L_U) \mid i \;\textbf{rios}\; e\} \cap \{i \in \mathcal{IOTS}(L_I, L_U) \mid i \;\textbf{ioco}\; s\} \neq \emptyset$$

□

The restriction of an observation objective to a subset of the suspension traces of the specification is a sufficient condition to guarantee a non empty intersection of conforming and exhibiting implementations. This restriction excludes observation objectives that challenge implementations to react on inputs that are not specified by the specification. Since our goal is to test for conforming implementations it makes no sense to check for exhibition of non-specified behaviour. From now on we choose singular and plural observation objectives such that $e \subseteq \mathit{Straces}(s)$, and $E \subseteq \mathcal{P}(\mathit{Straces}(s))$, respectively. From practical point of view, a test engineer might want to specify an observation objective in a less strict way, i.e. including traces which are not in the specification. Then we take as semantics for the observation objective, the intersection of $\mathit{Straces}(s)$ and the specified traces.

We present a simple algorithm to derive a test suite from an observation objective and a specification.

Algorithm 5.6
Let $s \in \mathcal{LTS}(L_I \cup L_U)$ be a specification, and let $e \subseteq \mathit{Straces}(s) \subseteq L_\delta^*$ be a singular observation objective. Then a test suite $T \subseteq \mathit{TEST}(L_U, L_I)$ can be generated, by adding a test case $t_{\epsilon,\sigma}$ to T for every trace σ of e, i.e. $T = \{t_{\epsilon,\sigma} \mid \sigma \in e\}$. The test case $t_{\epsilon,\sigma}$ is obtained by application of the following rules:

1. $t_{\rho,\epsilon} :=$ **pass**
2. $t_{\rho,a\cdot\sigma'} := a; t_{\rho\cdot a,\sigma'}$ with $a \in L_I$
3. $t_{\rho,x\cdot\sigma'} := \quad \Sigma\{\overline{\alpha}; \textbf{fail} \mid \alpha \in L_U \cup \{\delta\} \text{ and } \alpha \notin \mathit{out}(\, s \textbf{ after } \rho\,)\}$
   $\quad + \;\; \Sigma\{\overline{\alpha}; \textbf{pass} \mid \alpha \in \mathit{out}(\, s \textbf{ after } \rho\,)\backslash\{x\}\}$
   $\quad + \;\; \overline{x}; t_{\rho\cdot x,\sigma'}$, with $x \in L_U \cup \{\delta\}$

□

Proposition 5.7

1. A test suite obtained with Algorithm 5.6 is sound with respect to s and **ioco**.
2. A test suite obtained with Algorithm 5.6 is e-sound with respect to **rios** and e.
3. A test suite obtained with Algorithm 5.6 is e-exhaustive with respect to **rios** and e.

□

As a consequence of Proposition 5.7 the obtained test suite is e-complete. The next example shows a resulting test suite produced by the algorithm.

Example 5.8
The test cases depicted in Figure 3 will be generated when we apply Algorithm 5.6 for the specification s in Figure 2a, using the singular observation objective { coin · milk · coffee, coin · coffee }, expressing that we wish to see evidence of delivery of coffee. Note that now $\mathit{TOBS} = \mathcal{P}((\{\text{coin,coffee,milk},\delta\})^*)$. For clarity we marked the states where the observation objective is exhibited with **hit**. Notice that this can only be done with singular observation objectives, since in general we need more than one observation before we can conclude that a plural observation objective exhibits.

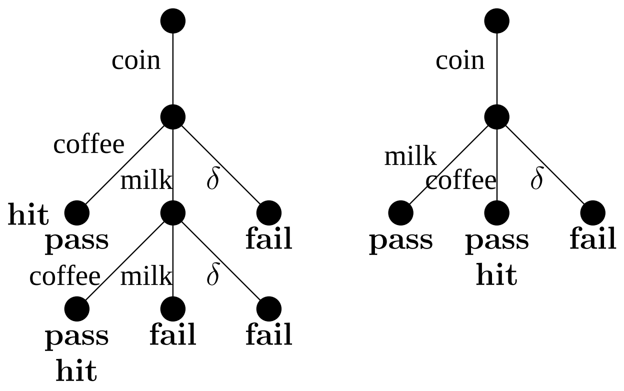


Figure 3: Test suite for a singular observation objective

□

In order to judge if we have found evidence whether an implementation exhibits the observation objective, we gather all the observations during testing, i.e. a set of test runs. By applying the **hit**-function $H_e^{\mathbf{rios}}$ we get the exhibition verdict.

The test generation of test suites for plural observation objectives is straightforward. We flatten the set of singular observation objectives to one set and apply Algorithm 5.6. So, let $s \in \mathcal{LTS}(L_I \cup L_U)$ be a specification and $E \subseteq \mathcal{P}(\mathit{Straces}(s))$ be a plural observation objective. Then a test suite $T \in \mathit{TEST}(L_U, L_I)$ for testing can be obtained by applying Algorithm 5.6 with $e = \bigcup E$. To obtain a verdict of exhibition of a plural observation objective, we evaluate $H_E^{\mathbf{riop}}$ for the obtained set of test runs during execution of the test suite.

The presented algorithm is not optimized for efficiency. For instance, one might combine several traces of a singular observation objective to generate one test case instead of the generation of one test case for every trace. In the case of Figure 3 we can reduce the test suite to only the leftmost test case.

During testing, it is possible that we do not detect that an implementation exhibits a plural observation objective, but only some of the composing singular observation objectives are exhibited. This can be due to many reasons like nondeterministic behaviour of the implementation, limited execution time etc. It is possible to define a measure to see how much of a plural observation objective has been exhibited during the test execution. This can be done in a straightforward way by counting how many elements of the observation objective have been hit. We call this coverage *exhibition coverage.*

Definition 5.9
Let $E \subseteq \mathcal{P}(L_\delta^*)$ be a plural observation objective and let $O \subseteq L_\theta^*$ be a set of observations. Then the *exhibition coverage* γ_E is defined as:

$$\gamma_E \quad =_{\text{def}} \quad \frac{|\{e \in E \mid H_e^{\mathbf{rios}}(O) = \mathbf{hit}\}|}{|E|}$$

$\square$

6 Concluding Remarks

In this paper we have proposed a formal framework for observation objectives and we have shown how to use it as means for test selection in the field of conformance testing. The major contribution of this framework is that we have a clear and precise notion of what an observation objective is, when it is satisfied and how test results based on observation objectives should be interpreted. Furthermore we have instantiated the framework with **ioco**-based testing theory and we have presented an e-complete and sound test derivation algorithm.

One of the major questions is how to represent and specify an observation objective. As we showed by our examples, a test engineer can specify a test object by explicitly writing down a set of suspension traces. However, in practice we would like to put a level above that and use observation objectives as a semantic object for traditional test purposes. Given a specification and a TGV like test purpose, i.e. an automaton, or a regular expression over traces (Autolink), we would like to automatically translate this to an observation objective. Another idea is to represent a test purpose by a property of a specification in a logic, e.g. LTL, and define the semantics in terms of observation objectives.

However, observation objectives give not only means to represent test purposes that somehow are related to the specification, but can also be used to specify strategies. Examples of these are maximum execution length in terms of number of observations, or specifying an output eager observation strategy. The latter might be expressed by a regular expression over traces: $((L_I{\cdot}(L_U^*){\cdot}\delta)^*)$.

Another issue is the combination of several observation objectives. As an example one might think of a combination of a test purpose together with a pragmatic (often applied) strategy. Having the formal framework, with the formal semantics, we can define a logic over observation objectives to facilitate these combinations.

Currently we have already implemented the basics of the observation objectives approach in the on-the-fly test derivation and execution tool TorX [BFV+99]. An observation objective is represented by a labelled transition system, which is again represented by a process algebra (LOTOS) and offers a functionality to specify a strategy for the test derivation instead of only randomness. The first experiments are promising, in the sense that we indeed succeed in steering the test derivation process.

Acknowledgement

The authors would like to acknowledge Lex Heerink, Axel Belinfante, Jan Feenstra and the anonymous reviewers for their constructive comments.

References

[BAL+90] E. Brinksma, R. Alderden, R. Langerak, J. van de Lagemaat, and J. Tretmans. A formal approach to conformance testing. In J. de Meer, L. Mackert, and W. Effelsberg, editors, *Second Int. Workshop on Protocol Test Systems*, pages 349–363. North-Holland, 1990. Also: Memorandum INF-89-45, University of Twente, The Netherlands.

[Ber91] G. Bernot. Testing against formal specifications: A theoretical view. In S. Abramsky and T.S.E. Maibaum, editors, *TAPSOFT'91, Volume 2*, pages 99–119. Lecture Notes in Computer Science 494, Springer-Verlag, 1991.

[BFV$^+$99] A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, 12^{th} *Int. Workshop on Testing of Communicating Systems*. Kluwer Academic Publishers, 1999.

[DN87] R. De Nicola. Extensional equivalences for transition systems. *Acta Informatica*, 24:211–237, 1987.

[DNH84] R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[DRS$^+$00] L. Du Bousquet, S. Ramangalahy, S. Simon, C. Viho, A. Belinfante, and R.G. de Vries. Formal test automation: The conference protocol with TGV/Torx. In H. Ural, R.L. Probert, and G. v. Bochmann, editors, *IFIP* 13^{th} *Int. Conference on Testing of Communicating Systems(TestCom 2000)*. Kluwer Academic Publishers, 2000.

[Hol99] M. Hollenberg. Test templates for test generation. In K. Tarnay G. Csopaki, S. Dibuz, editor, 12^{th} *Int. Workshop on Testing of Communicating Systems*, pages 167–178. Kluwer Academic Publishers, 1999.

[ISO91] ISO. *Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework.* International Standard IS-9646. ISO, Geneve, 1991. Also: CCITT X.290–X.294.

[ISO96] ISO/IEC JTC1/SC21 WG7, ITU-T SG 10/Q.8. *Information Retrieval, Transfer and Management for OSI; Framework: Formal Methods in Conformance Testing.* Committee Draft CD 13245-1, ITU-T proposed recommendation Z.500. ISO – ITU-T, Geneve, 1996.

[JM99] T. Jéron and P. Morel. Test generation derived from model-checking. In *Computer Aided Verification CAV'99*. Lecture Notes in Computer Science, Springer-Verlag, 1999.

[LT89] N.A. Lynch and M.R. Tuttle. An introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989. Also: Technical Report MIT/LCS/TM-373 (TM-351 revised), Massachusetts Institute of Technology, Cambridge, U.S.A., 1988.

[LY96] D. Lee and M. Yannakakis. Principles and methods for testing finite state machines – a survey. *The Proceedings of the IEEE*, 84(8):1090–1123., August 1996.

[SEK$^+$98] M. Schmitt, A. Ek, B. Koch, J. Grabowski, and D. Hogrefe. – AUTOLINK – Putting SDL-based Test Generation into Practice. In A. Petrenko and N. Yevtushenko, editors, 11^{th} *Int. Workshop on Testing of Communicating Systems*, pages 227–243. Kluwer Academic Publishers, 1998.

[Tre96] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996. Also: Technical Report No. 96-26, Centre for Telematics and Information Technology, University of Twente, The Netherlands.

[Tre99] J. Tretmans. Testing Concurrent Systems: A Formal Approach. In J.C.M. Baeten and S. Mauw, editors, *CONCUR'99 –* 10^{th} *Int. Conference on Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 46–65. Springer-Verlag, 1999.

[Vaa91] F. Vaandrager. On the relationship between process algebra and Input/Output Automata. In *Logic in Computer Science*, pages 387–398. Sixth Annual IEEE Symposium, IEEE Computer Society Press, 1991.

Complexity Issues of Connectivity Testing

Jens Chr. Godskesen
The IT University of Copenhagen

Abstract

This paper deals with complexity issues of *connectivity testing*, an approach put forward for testing embedded systems. Instead of testing the conformance of a system against its specification, which often turns out to be infeasible, in connectivity testing only the composition of the software and the hardware in which the software is embedded is tested. The testing framework is based on the notion of a *fault model*, that is a model which formally captures errors in the interface between the hardware and the software. A *complete* test suite for a fault model is a test suite that detects the faults of the model with 100 % coverage. We prove the problem of computing a smallest complete test suite to be NP-hard. Therefore we devise approximative polynomial time algorithms computing *minimal complete* test suites.

1 Introduction

Connectivity testing is a testing framework centered around testing of embedded systems, it has previously been presented in the papers [10, 7, 9, 8]. *Embedded systems* are dedicated systems like mobile phones, hi-fi equipment, remote controls etc., with a very limited user interface, for instance a few push buttons, and a simple display, where the software, typically a finite state machine (FSM), is embedded in the hardware.

Testing of embedded systems has become more and more difficult because the advances in processor speed and memory size at a low cost have made the manufacturing of sophisticated embedded systems feasible. Also, because products are often manufactured in large scale testing should have high focus in order to avoid a cumbersome an expensive updating of errors in the products.

Traditionally the problem of testing a system correct with respect to a specification is often referred to as *conformance testing.* The goal is to derive, hopefully automatically, a set of tests, a *test suite*, from the specification that may help to ensure the conformance of the system behaviour against the specification. That is, the goal is to check whether the behaviour of the implementation is the one of its specification. The test suite should at least enjoy the property that if the system does not pass all the tests in the test suite then it is not in conformance with its specification. A formal framework for conformance testing has been outlined in [3, 17].

The problem with conformance testing in general however is the size of the test suites, which may turn out to be infinite. Therefore the outcome of applying a finite number of tests in a test suite may only approximate a full conformance test. As a consequence conformance testing may be practically infeasible.

For specifications being FSM's it has been shown although that test suites need not be infinite. In [5] it was proven that conformance between a specification S and its implementation I can be tested by a test suite that is polynomial in the size of the number of states and the number of input symbols in S, given that I is an FSM with the same input alphabet and the same number of states as S, and that S can be reset to its initial state at any time. [1] A polynomial time algorithm for constructing the test suite was outlined in [5].Unfortunately, if$I$ has more states than S then the size of the test suite is exponential. Occasionally even stronger assumptions about S may be taken and in that case other methods can be applied, for instance the Rural Chinese Postman Tour [2] (for an overview see e.g. [11] and [13]).

For industrial sized examples it may often be however that S contains so many states and input symbols that even if S and I are of equal size the conformance testing problem is intractable, either because the test generation procedure is too time consuming or because the test suite is too large.

Connectivity testing is an alternative, although still formally rooted, approach for testing embedded systems. The idea being that the focus is on testing the *composition* of the two system components: the embedded software and the hardware in which the software is embedded. Hence, in contrast to conformance testing, our goal is *not* to test conformance between the behaviour of the system specification and the behaviour of its implementation. Instead we want to test for errors that may be detected as faults in the interface between the two composed components. We define that kind of errors by means of *fault models*.

A test suite that detects the faults defined by a fault model with 100 % coverage is said to be *complete*. In turns out however that the problem of generating a smallest complete test suite is NP-hard. We therefore provide approximative and greedy polynomial time algorithms for computing *minimal complete* test suites. A minimal complete test suite is complete, albeit not necessarily a smallest one, but no test may be removed from it in order to maintain it complete.

[1]Recently this approach has been carried out in a timed setting in [16] and [4]. The results in [16] is mainly of theoretical interest because of its double exponential test generation algorithm, and the ideas in [4] has been shown useful for specification with a little more than 100 states.

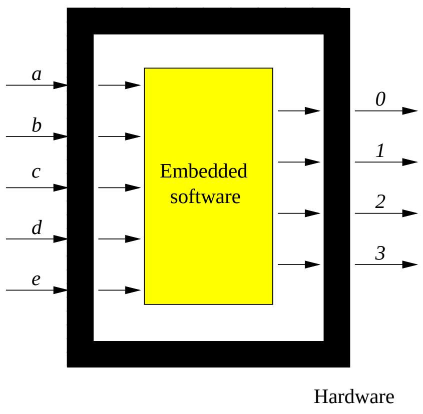


Figure 1: Embedded systems.

The rest of the paper is organized as follows. In Sec. 2 we motivate our approach. In Sec. 3 we define notions and terminology related to FSM's. Fault models are introduced in Sec. 4 and test suites in Sec. 5. The existence of minimal complete test suites as well as the NP-hardness of computing a smallest such one is addressed in Sec. 6. In Sec. 7 we show that minimal complete test suites may be computed in polynomial time. Sec. 8 is the conclusion. Proofs can be found in the appendix.

2 Motivation

Conceptually an embedded system may be regarded as depicted in Fig. 1. That is, an embedded system consists of embedded software encapsulated by hardware. Notably all communications between the system environment and the embedded software pass through the hardware. In Fig. 1 this is visualized by letting the inputs from the system environment to the software (a, b, c, d, e) pass through the hardware towards the software via *connections* (the unlabeled arrows). Similarly, the outputs $(0, 1, 2, 3)$ generated by the software have to pass via connections through the hardware in order to emerge at the system environment.

Each connection is assumed to be related to precisely one input or output. This assumption im-

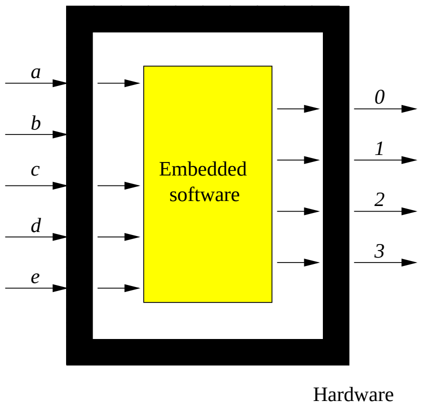


Figure 2: A faulty embedded systems.

plicitly implies that there is a one to one correspondence between external inputs to the system and the inputs to the embedded software, likewise there is a one to one correspondence between the outputs from the software and the outputs from the system. Connections are considered to be abstract notions, they may have no direct physical counterpart.

Ideally it should be ascertained that the specification of the software component is correct. For instance, it may have been verified by some FSM verification technique. Then exploiting the ability to automatically generate executable code from specifications and assuming a careful construction of such compilers it would be reasonable to expect the generated code to be correct with respect to the specification, that is the two perform the same FSM behaviour.

In the composition of the two system components it then follows that the hardware (and probably drivers managing the interaction between the hardware and the software) may be the only error prone part. Therefore in order to manage the multitude of potential errors we shall make an abstraction and regard the hardware (and the drivers) as a black box interfacing the embedded software through the connections. As a consequence system errors may now only be referred to in terms of the connections.

Given these assumptions and in order to conclude that the system is correct it should be tested that there are no errors manifested as faults in the connections between the hardware component and the embedded software. In the system in Fig. 1 a fault could for instance be that one of the connections is missing as shown in Fig. 2 where the b-input is *disconnected.* In the physical world, say a mobile phone, this may correspond to the situation where some button of the system is not connected such that the software will never receive the input, and therefore the pressing of the button will cause no effect. In order to make sure the faults are testable they are assumed to be *permanent*, that is no connection may alter between being connected and disconnected.

Testing in order to detect the kind of faults addressed in this paper is a matter of providing sequences of inputs that will reveal the missing connections. If say the b-input connection in Fig. 1 is missing this may be revealed by an input sequence containing b. The fault may be revealed directly if the system on input b immediately is expected to return some output, because the erroneous system displays no output. The fault may be exposed indirectly if the system after input b eventually is expected to return some output and it turns out that this output is distinct from the one actually shown by the faulty system.

Following the ideas outlined above we outline in the remaining part of this paper a framework for testing embedded systems by means of fault models. It is assumed that embedded software behaves as FSM's. We introduce one type of fault models for *disconnected inputs.*

The notion of a fault model has appeared elsewhere in the literature. In particular, fault models has been used for test generation for embedded systems in [15, 14]. However, although similar their notion of a fault model is in the setting of conformance testing. Another significant difference is that they consider the embedded component to be the erroneous part. In the hardware community *structural fault models*, like for instance the "stuck at" model, has for decades been used for circuit testing (see e.g. [1]).

3 Finite state machines

We consider a variant of deterministic finite state machines (FSM's) where a set of outputs (possibly the empty one) is produced on each input.

Definition 1 *An FSM, M, with input type $\mathcal{E}$ is a five tuple $(\mathcal{S}, \mathcal{E}, \Omega, \tau, s_M)$ where $\mathcal{S}$ is a finite set of states, $\mathcal{E}$ is a finite set of inputs, Ω is a finite set of outputs, and τ is a transition function, $\tau : \mathcal{S} \times \mathcal{E} \to \mathcal{S} \times 2^{\Omega}$. $s_M \in \mathcal{S}$ is the initial state.*

In the remaining part of this paper we presuppose for any FSM, that the sets $\mathcal{S}$, $\mathcal{E}$ and Ω are ranged over by s, α, and ω respectively. Also we let o range over 2^{Ω} and we let σ range over $\mathcal{E}^*$, including the empty sequence ϵ. If σ' is a prefix of σ we write $\sigma' \preceq \sigma$. $|\sigma|$ is the length of σ. The notation $\sigma{\downarrow}\alpha$ denotes σ where all occurrences of α are removed, that is

$$\sigma{\downarrow}\alpha \quad = \quad \begin{cases} \epsilon & \text{if } \sigma = \epsilon \\ \sigma'{\downarrow}\alpha & \text{if } \sigma = \sigma'\alpha \\ (\sigma'{\downarrow}\alpha)\alpha' & \text{if } \sigma = \sigma'\alpha',\ \alpha \neq \alpha' \end{cases}$$

For a transition function τ we write $s \xrightarrow{\alpha/o} s'$ and s^{α} for s' if $\tau(s, \alpha) = (s', o)$. If $s_{i+1} = {s_i}^{\alpha_i}$ for $i = 1, \ldots, n$ we write s^{σ} for s_{n+1} where $\sigma = \alpha_1 \ldots \alpha_n$. Whenever $s = {s_M}^{\sigma}$ for some σ we say that s is *reachable*. Often we shall regard a state s as a function $s : \mathcal{E}^* \to 2^{\Omega}$ defined by

$$s(\sigma) \quad = \quad \begin{cases} \emptyset \text{ if } \sigma = \epsilon \\ o \text{ if } \sigma = \sigma'\alpha,\ s^{\sigma'} \xrightarrow{\alpha/o} s^{\sigma} \end{cases}$$

We generalize this notion to machines and write $M(\sigma)$ for $s_M(\sigma)$.

Definition 2 *Let M and M' be two (not necessarily distinct) FSM's with the same input type and let s and s' be states in M and M' respectively. Then $s \approx_i s'$ if $s(\sigma) = s'(\sigma)$ for all σ with $|\sigma| \leq i$. $s \approx s'$ if $s \approx_i s'$ for all i. $M \approx M'$ if $s_M \approx s_{M'}$.*

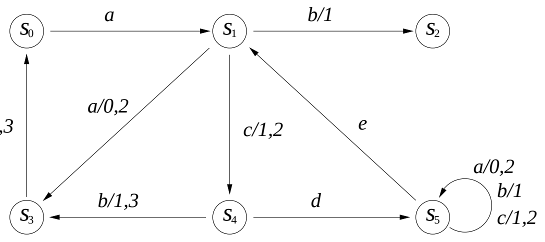


Figure 3: An FSM with initial state s_0.[2]

Suppose in the following two FSM's M and M' with the same input type $\mathcal{E}$ and with disjoint state sets $\mathcal{S}$ and $\mathcal{S}'$ respectively. Similarly to the proof of Theorem 10-2 in [12] on may prove that:

Lemma 3 *Let $m = |\mathcal{S}| + |\mathcal{S}'|$. Then $s \not\approx s'$ iff $s \not\approx_i s'$ for some $i < m$.*

Let $\mathcal{A}_{\alpha} \subseteq \mathcal{S} \times \mathcal{S}'$ be defined by

$$\mathcal{A}_{\alpha} = \{(s, s') \,|\, s(\alpha) \neq s'(\alpha)\}$$

For any $i = 1, 2, \ldots$ define the set $\mathcal{A}_i \subseteq \mathcal{S} \times \mathcal{S}'$ inductively by

$$\begin{aligned} \mathcal{A}_1 \quad &= \quad \bigcup_{\alpha \in \mathcal{E}} \mathcal{A}_{\alpha} \\ \mathcal{A}_{i+1} \quad &= \quad \{(s_1, s_2) |\, \exists \alpha.({s_1}^{\alpha}, {s_2}^{\alpha}) \in \mathcal{A}_i\} \end{aligned}$$

Let $\mathcal{B}_i = \mathcal{A}_i \cup {\mathcal{A}_i}^{-1}$ for $i = 1, 2, \ldots$. Then the proof of inequivalence being characterized as described by the following lemmas follows directly from the definition of $\mathcal{B}^i$.

Lemma 4 *$s \not\approx_i s'$ iff $(s, s') \in \mathcal{B}^i$.*

In order to help defining fault models in Sec. 4 we introduce a syntactic convention for defining *mutations* of FSM's.

Definition 5 *$M[\alpha]$ is M where any transition $s \xrightarrow{\alpha/o} s'$ in M is replaced by $s \xrightarrow{\alpha/\emptyset} s$.*

[2]Empty outputs and transitions to the same state with empty output are left out.

Letting M denote the machine in Fig. 3, $M[b]$ is the machine in Fig. 4. One interpretation of $M[\alpha]$ is that the input α has no longer any effect in $M[\alpha]$ because the machine remains in its state and generates no output. This intuition is represented by the lemma and corollary below:

Lemma 6 *$M[\alpha](\sigma\alpha') = M(\sigma\alpha'{\downarrow}\alpha)$ if $\alpha \neq \alpha'$.*

Corollary 7 *$M(\sigma) = M[\alpha](\sigma)$ if $\sigma = \sigma{\downarrow}\alpha$.*

We end this section with a characterization of when an FSM mutation, $M[\alpha]$, is inequivalent to its origin M. First however, some terminology is introduced.

Definition 8 *s is α-*active *if $s(\alpha) \neq \emptyset$. s is α-distinguishable (of degree i) if $s \not\approx_i s^\alpha$. M is α-active (α-distinguishable) if it contains a reachable α-active (α-distinguishable) state. M is α-sensitive if it is α-active or α-distinguishable.*

From the definition above it follows that M is α-active if $M(\sigma\alpha) \neq \emptyset$ for some σ and M is α-distinguishable if $M(\sigma_1\alpha\sigma_2) \neq M(\sigma_1\sigma_2)$ for some σ_1 and some *distinguishing* sequence σ_2. As an example, in Fig. 3 it turns out that s_1 and s_5 are the only a-active states, s_4 is d-distinguishable of degree 1, and s_5 is e-distinguishable of degree 2.

The characterization as to whether a mutation is equivalent or not to its origin are stated by the lemma below.

Lemma 9 *$M \not\approx M[\alpha]$ iff M is α-sensitive.*

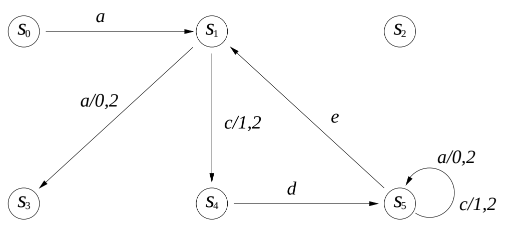


Figure 4: The FSM in Fig. 3 with input b removed.

4 Fault models

As already touched upon in Sec. 2 the purpose of a fault model is to capture the errors that are manifested as disconnections between the hardware and the software components. For instance, we would like that the fault shown in Fig. 2 where the b-input is disconnected can be handled by a fault model. Importantly, a fault model should be the formal basis for the generation of test suites.

The fault in Fig. 2 implies that the input b never will occur as input to the embedded software. Recalling that we stipulated the embedded software to be correct relative to its FSM specification, it then follows that at the specification level we may choose to model a fault as a modification of the original specification. If for instance the specification of the embedded software is the FSM in Fig. 3 then the error prone version of that FSM where input b has no effect is the mutation in Fig. 4. We therefore may choose to let a fault relative to some M be any mutation of M, say M', such that $M \not\approx M'$ and we may let a fault model be a set of such modified FSM's. In general we define a fault model as follows

Definition 10 *A fault model for M is a set of FSM's $\mathcal{M}^M$ each with the same input type as M such that $M \not\approx M'$ for all $M' \in \mathcal{M}^M$.*

The fault models for *disconnected inputs* are expected to capture faults as the one described at the beginning of this section where the b-input is disconnected, that is they should cover the faults that arise when input connections are disrupted. A fault like this occurs for instance when the pressing of a button on a mobile phone does not cause any effect because the input corresponding to the button is disconnected.

A single disconnection fault can be modeled by the mutation $M[\alpha]$ because it specifies that the input α has no behavioural effect. The fault models for disconnected inputs consists of sets of such specifications. Let $\mathcal{E}' \subseteq \mathcal{E}$ where $\mathcal{E}$ is the input type for some M. Then, if M is α-sensitive for any $\alpha \in \mathcal{E}'$, a fault model for disconnected input for M may be defined by

$$\mathcal{M}^{M}_{\mathcal{E}'} = \{M[\alpha] \,|\, \alpha \in \mathcal{E}'\}$$

This fault model represents any one implementation in which precisely a single input α is disconnected for some $\alpha \in \mathcal{E}'$, but otherwise the implementation behaves as specified by M. If we for instance let M denote the FSM in Fig. 3 then

$$\mathcal{M}^{M}_{\{a,b,c,d,e\}} \tag{1}$$

is an input fault model for M containing a mutation of M for each of its inputs. For instance it contains the FSM defined in Fig. 4.

We let $\mathcal{M}^{M}_{\mathcal{E}}$ range over the set of fault models for M where $\mathcal{E}$ is a subset of the input type of M. Note that any fault model $\mathcal{M}^{M}_{\mathcal{E}}$ is finite because the input type is finite for any M.

5 Tests

The system to be tested obviously is a nonformal object. Hence often when developing a formal testing framework some assumption that allows for regarding the system as a formal object is taken. This step is traditionally referred to as the *testing hypothesis.* [3]

Recall the conceptualization of embedded systems put forward in Sec. 2 and in particular the one to one correspondences between external and internal inputs and outputs. In our testing framework we shall adopt the convention that the names for the external inputs to (outputs from) the embedded system are identical to the names for the inputs to (outputs from) the embedded software. This assumption, together with the overall assumption of embedded software being correctly compiled from some FSM specification, facilitates that the embedded system to be tested can be considered an FSM with the same input type as its software specification.

[3]See for instance [17, 4].

Taking advantage of this testing hypothesis we may now formally define the notions related to tests and how to apply a test to a system.

Definition 11 *Let $\mathcal{E}$ be a set of inputs. A test is a finite sequence of inputs $\sigma \in \mathcal{E}^*$. A* test suite *$T \subseteq \mathcal{E}^*$ is a finite set of tests.*

Given a set of tests T we would like to filter out those tests in T with the maximal respectively minimal length, so let

$$max(T) = \{\sigma \in T \,|\, \forall \sigma' \in T.\, |\sigma| \geq |\sigma'|\}$$

and let

$$min(T) = \{\sigma \in T \,|\, \forall \sigma' \in T.\, |\sigma| \leq |\sigma'|\}$$

We let the size of a set of tests T be defined by $|T| = \Sigma_{\sigma \in T} |\sigma|$.

For a family of sets of tests $\langle T_i \rangle_{i \in I}$ we let, when it is clear from the context, $\langle T_i \rangle_{i \in I}$ denote the set of set of tests such that $T \in \langle T_i \rangle_{i \in I}$ if $T = \{\sigma_i \,|\, i \in I\}$ with $\sigma_i \in T_i$ for all $i \in I$. That is, T contains a test from each member of the family $\langle T_i \rangle_{i \in I}$ and any test in T belongs to a member of the family.

Definition 12 *Let $\mathcal{E}$ be a set of inputs. Let $\sigma \in \mathcal{E}$, $T \subseteq \mathcal{E}^*$, and let $T_i \subseteq \mathcal{E}^*$ for all $i \in I$. The cover of σ with respect to a family $\langle T_i \rangle_{i \in I}$, $\sigma(\langle T_i \rangle_{i \in I})$, is defined by*

$$\sigma(\langle T_i \rangle_{i \in I}) = \{i \in I \,|\, \exists \sigma' \preceq \sigma.\, \sigma' \in T_i\}$$

The cover of T with respect to $\langle T_i \rangle_{i \in I}$ is $T(\langle T_i \rangle_{i \in I}) = \bigcup_{\sigma \in T} \sigma(\langle T_i \rangle_{i \in I})$.

That is, a test covers a family member of $\langle T_i \rangle_{i \in I}$ if it contains a prefix that belongs to the member. A set of tests covers a family member if one of its tests does. For instance, it is obvious that $T(\langle T_i \rangle_{i \in I}) = I$ for all $T \in \langle T_i \rangle_{i \in I}$.

Let M and M' be two FSM's with the same input type $\mathcal{E}$. [4] The application of a test $\sigma \in \mathcal{E}^*$ and

[4]E.g. M may be the specification of the software in the embedded system M'.

a set of tests $T \subseteq \mathcal{E}^*$ respectively to M' with respect to M is defined by

$$apply_M(M',\sigma) = \begin{cases} pass & \text{if } \forall \sigma' \preceq \sigma.\ M(\sigma') = M'(\sigma') \\ fail & \text{otherwise} \end{cases}$$

$$apply_M(M',T) = \begin{cases} pass & \text{if } \forall \sigma \in T.\ apply_M(M',\sigma) = pass \\ fail & \text{otherwise} \end{cases}$$

Knowing how to apply tests we introduce the notions of *sound*, *exhaustive*, and *complete* for sets of tests.

Definition 13 *T is sound for $\mathcal{M}_{\mathcal{E}}^M$ if for any $\sigma \in T$ there exists some $M' \in \mathcal{M}_{\mathcal{E}}^M$ such that $apply_M(M',\sigma) = fail$. T is exhaustive for $\mathcal{M}_{\mathcal{E}}^M$ if for all $M' \in \mathcal{M}_{\mathcal{E}}^M$, $apply_M(M',T) = fail$. T is complete for $\mathcal{M}_{\mathcal{E}}^M$ if it is sound and exhaustive for $\mathcal{M}_{\mathcal{E}}^M$.*

That is, a set of tests T is sound for a fault model if any $\sigma \in T$ is guaranteed to detect some fault of the model, and T is exhaustive for a fault model if any fault of the model is detectable by some $\sigma \in T$.

Complete test suites for the fault models of disconnected inputs can be obtained by means of sets of tests that characterize the faults in the models. A characterization of $M[\alpha]$ can be defined by $T_\alpha^M = T_1 \cup T_2$ where

$$T_1 = \{\sigma\alpha \,|\, M(\sigma\alpha) \neq \emptyset\}$$

and

$$T_2 = \{\sigma\alpha' \,|\, M(\sigma\alpha') \neq M(\sigma\alpha' \downarrow \alpha),\ \alpha' \neq \alpha\}$$

Hence a test for $M[\alpha]$ may be $\sigma\alpha$ for some σ where σ takes s_M to an α-active state. Or, a test for $M[\alpha]$ may be $\sigma\alpha'$ for some σ and α' where $\alpha' \neq \alpha$ such that M does not produce the same output on $\sigma\alpha'$ and the test constructed by removing all α's from $\sigma\alpha'$. For instance, if M is the FSM in Fig. 3, then $\{aa, ab, ac\} \subseteq T_a^M$, $\{ab, aab, acdb\} \subseteq T_b^M$, $\{ac, acdc\} \subseteq T_c^M$, $\{acda, acdc\} \subseteq T_d^M$, and $\{acdeaa, acdebb\} \subseteq T_e^M$.

Notice, that a characterizing set may in general be infinite. Actually all the characterizing sets mentioned in the examples above are infinite.

In order to prove that these characterizations can be applied for the construction of complete test suites we use the following:

Lemma 14 *Let $\mathcal{E}$ be the input type of M. Then for all $\alpha \in \mathcal{E}$, $apply_M(M[\alpha],\sigma) = fail$ iff $\exists \sigma' \in T_\alpha^M.\ \sigma' \preceq \sigma$.*

Corollary 15 *If $T \in \langle T_\alpha^M \rangle_{\alpha\in\mathcal{E}'}$ for some $\mathcal{E}' \subseteq \mathcal{E}$ then T is sound for $\mathcal{M}_{\mathcal{E}}^M$.*

Corollary 16 *The cover of T wrt. $\langle T_\alpha^M \rangle_{\alpha\in\mathcal{E}}$ is $\mathcal{E}$ iff T is exhaustive for $\mathcal{M}_{\mathcal{E}}^M$.*

From Corol. 15 and 16 it follows that a complete test suite can be defined as shown below.

Theorem 17 *If $T \in \langle T_\alpha^M \rangle_{\alpha\in\mathcal{E}}$, $T' \subseteq T$, and the cover of T' wrt. $\langle T_\alpha^M \rangle_{\alpha\in\mathcal{E}}$ is $\mathcal{E}$ then T' is complete for $\mathcal{M}_{\mathcal{E}}^M$.*

Notice, no complete test suite for a fault model $\mathcal{M}_I^M$ needs to contain more tests than the number of faults in $\mathcal{M}_I^M$. As an example, letting $\mathcal{M}_{\mathcal{E}}^M$ be as defined by Eqn. 1 then $T_3 = \{aa, ab, ac, acda, acdeaa\}$ and $T_4 = \{ab, ac, acda, acdeaa\}$ are complete for $\mathcal{M}_{\mathcal{E}}^M$ because they belong to $\langle T_\alpha \rangle_{\alpha\in\mathcal{E}}$. However, also the subset $\{ab, acdeaa\}$ is complete for $\mathcal{M}_{\mathcal{E}}^M$ because its cover with respect to $\langle T_\alpha^M \rangle_{\alpha\in\mathcal{E}}$ is $\{a, b, c, d, e\}$.

6 Minimal complete test suites

As indicated at the end of the previous section some complete test suites for a fault model contain fewer tests than other complete test suites for the same fault model. Clearly, in practice one would be interested in complete test suites that are as small as possible. Ideally, for some fault model $\mathcal{M}_{\mathcal{E}}^{M}$, it would be preferable to construct a *smallest complete* test suite T for $\mathcal{M}_{\mathcal{E}}^{M}$. That is, a test suite T where $|T| \leq |T'|$ for any test suite T' that is complete for $\mathcal{M}_{\mathcal{E}}^{M}$. However,

Theorem 18 *The problem of computing a smalles complete test suite for some fault model $\mathcal{M}_{\mathcal{E}}^{M}$ is NP-hard.*

Hence we settle with an easier problem. The problem we want to solve is to find a test suite that is complete for a fault model and which does not contain genuine subsets that are also complete for the same fault model. Such a test suite we call *minimal complete.*

Definition 19 *A test suite T is minimal complete for $\mathcal{M}_{\mathcal{E}}^{M}$ if T is complete for $\mathcal{M}_{\mathcal{E}}^{M}$ and if no $T' \subset T$ is complete for $\mathcal{M}_{\mathcal{E}}^{M}$.*

Because any test in a minimal complete test suite T for a fault model $\mathcal{M}_{\mathcal{E}}^{M}$ detects at least one fault in $\mathcal{M}_{\mathcal{E}}^{M}$ it holds that T contains no more tests that the number of faults in $\mathcal{M}_{\mathcal{E}}^{M}$.

Given a fault model $\mathcal{M}_{\mathcal{E}}^{M}$ and given a test suite $T \in \langle T_{\alpha}^{M} \rangle_{\alpha \in \mathcal{E}}$, the problem of determining a minimal complete test suite can, due to Th. 17, be rephrased as a *set-covering problem.* That is, we have to cover the set $\mathcal{E}$ by sets in

$$\{\sigma(\langle T_{\alpha}^{M} \rangle_{\alpha \in \mathcal{E}}) \,|\, \sigma \in T\}$$

and in particular we have to cover $\mathcal{E}$ by as few of these sets as possible. In other words, we have to find a smallest possible subset of T with a cover $\mathcal{E}$. The set-covering problem however is known to be NP-complete, although approximative methods for computing set-covers exists (see e.g. [6]). Our contribution in the remaining part of this section may be regarded as an approximative and greedy method for computing set-covers in that we outline a general procedure for obtaining minimal complete test suites. In Sec. 7 the procedure is shown to have polynomial time complexity. First however we need some terminology.

Definition 20 *Let $\langle T_i \rangle_{i \in I}$ be a family of sets of tests. Then $\mu\langle T_i \rangle_{i \in I}$ is the smallest set of sets of tests satisfying that $T \in \mu\langle T_i \rangle_{i \in I}$ if $T \in \langle T_i \rangle_{i \in I}$ and if*

$$\forall \sigma \in T.\ \exists i \in I.\ T \cap min(T_i) = \{\sigma\} \qquad (2)$$

That is, $T \in \mu\langle T_i \rangle_{i \in I}$ if any T_i has a test in T and if any test in T uniquely among the tests in T belongs to the smallest tests in some T_i. If $T \in \mu\langle T_i \rangle_{i \in I}$ we say that T is *minimalized* with respect to the family $\langle T_i \rangle_{i \in I}$.

For a family of non-empty set of tests $\langle T_i \rangle_{i \in I}$ a minimalized set of tests may be constructed by first selecting a set $T \in \langle min(T_i) \rangle_{i \in I}$. Then tests in T not satisfying Eq. 2 are iteratively removed from T one at a time until the remaining tests in T satisfy the equation. For instance, let M be the FSM defined in Fig. 3 and let T be the test suite $\{aa, ab, ac, acda, acdeaa\}$ then T belongs to $\langle T_{\alpha}^{M} \rangle_{\alpha \in \{a,b,c,d,e\}}$, but T does not belong to $\mu\langle T_{\alpha}^{M} \rangle_{\alpha \in \{a,b,c,d,e\}}$ because $\{aa, ab, ac\} \in min(T_{a}^{M})$ and therefore Eq. 2 is not satisfied by aa. It turns out that $\{ab, ac, acda, acdeaa\}$ is minimalized with respect to $\langle T_{\alpha}^{M} \rangle_{\alpha \in \{a,b,c,d,e\}}$.

If $T \in \langle min(T_i) \rangle_{i \in I}$ we let $\mu_{\langle T_i \rangle_{i \in I}}(T)$ denote a minimalized subset of T with respect to $\langle T_i \rangle_{i \in I}$. $\mu_{\langle T_i \rangle_{i \in I}}$ can be considered a function

$$\mu_{\langle T_i \rangle_{i \in I}} : \langle min(T_i) \rangle_{i \in I} \to \mu\langle T_i \rangle_{i \in I}$$

because the resulting set of tests can be constructed deterministically as outlined above if we presuppose some lexicographical ordering on the set of tests according to which tests are considered for removal. [5]

For any family of set of tests $\langle T_i \rangle_{i \in I}$ we next define the functions $\mathcal{H}_{\langle T_i \rangle_{i \in I}}$, $\mathcal{G}_{\langle T_i \rangle_{i \in I}}$, and $\mathcal{F}_{\langle T_i \rangle_{i \in I}}$ that are used for computing minimal complete test suites.

[5]In this paper we do not consider the problem of computing a smallest minimalized subset of $T \in \langle min(T_i) \rangle_{i \in I}$ with respect to $\langle T_i \rangle_{i \in I}$.

Definition 21 *Define for $T_i \subseteq \mathcal{E}^*$, $i \in I$ the function $\mathcal{H}_{\langle T_i \rangle_{i\in I}} : \langle min(T_i)\rangle_{i\in I} \to 2^{\mathcal{E}^*}$ by*

$$\mathcal{H}_{\langle T_i \rangle_{i\in I}}(T) = \begin{cases} \emptyset & \text{if } T = \emptyset \\ T' & \text{otherwise} \end{cases}$$

where $T' = \mu_{\langle T_i \rangle_{i\in I'}}(max(T))$ and where $I' = \{i \in I \,|\, \exists \sigma \in max(T).\ \sigma \in min(T_i)\}$.

That is, $\mathcal{H}_{\langle T_i \rangle_{i\in I}}(T)$ returns a minimalized set of tests with respect to the family to which the largest tests in T belong. For instance, if M is the FSM defined in Fig. 3 and if T is the test suite $\{aa, ab, ac\}$ that belongs to the family $\langle min(T^M_\alpha)\rangle_{\alpha\in\{a,b,c\}}$ then $\mathcal{H}_{\langle T_\alpha \rangle_{\alpha\in\{a,b,c\}}}(T)$ equals $\{ab, ac\}$. If T is $\{aa, ab, ac, acda, acdeaa\}$ that belongs to $\langle min(T^M_\alpha)\rangle_{\alpha\in\{a,b,c,d,e\}}$ then $\mathcal{H}_{\langle T_\alpha \rangle_{\alpha\in\{a,b,c,d,e\}}}(T)$ is $\{acdeaa\}$.

Definition 22 *Define for $T_i \subseteq \mathcal{E}^*$, $i \in I$ the function $\mathcal{G}_{\langle T_i \rangle_{i\in I}} : \langle min(T_i)\rangle_{i\in I} \to 2^{\mathcal{E}^*}$ by*

$$\mathcal{G}_{\langle T_i \rangle_{i\in I}}(T) = \begin{cases} \emptyset & \text{if } T = \emptyset \\ T'' & \text{otherwise} \end{cases}$$

where $T = \{\sigma_i \mid i \in I\}$, $T' = \mathcal{H}_{\langle T_i \rangle_{i\in I}}(T)$, and $T'' = \{\sigma_i \,|\, i \in I \setminus T'(\langle T_i\rangle_{i\in I})\}$.

That is, on input $\{\sigma_i \mid i \in I\} \in \langle min(T_i)\rangle_{i\in I}$ the function $\mathcal{G}_{\langle T_i \rangle_{i\in I}}$ returns a subset of this set. The tests not returned are those that belong only to family members which are covered by T'. T' is a minimalized set with respect to the family to which the largest tests in $\{\sigma_i \mid i \in I\}$ belongs. As an example, letting M be the FSM defined in Fig. 3, if T is the test suite $\{aa, ab, ac, acde, acdeaa\}$ that belongs to $\langle min(T^M_\alpha)\rangle_{\alpha\in\{a,b,c,d,e\}}$ then $\mathcal{G}_{\langle T_\alpha \rangle_{\alpha\in\{a,b,c,d,e\}}}(T)$ equals $\{ab\}$. The reason why is, as shown above, that $\mathcal{H}_{\langle T_\alpha \rangle_{\alpha\in\{a,b,c,d,e\}}}(T)$ is $\{acdeaa\}$ and because the cover of *acdeaa* with respect to $\langle T_\alpha\rangle_{\alpha\in\{a,b,c,d,e\}}$ is $\{a, c, d, e\}$ as shown in the previous section.

The following lemma ensures that the function $\mathcal{F}_{\langle T_i \rangle_{i\in I}}$ defined below is well defined.

Lemma 23 *$\mathcal{G}_{\langle T_i \rangle_{i\in I}}(T) \in \langle min(T_i)\rangle_{i\in I'}$ where $I' = I \setminus (\mathcal{H}_{\langle T_i \rangle_{i\in I}}(T))(\langle T_i\rangle_{i\in I})$.*

Definition 24 *Define for $T_i \subseteq \mathcal{E}^*$, $i \in I$ the function $\mathcal{F}_{\langle T_i \rangle_{i\in I}} : \langle min(T_i)\rangle_{i\in I} \to 2^{\mathcal{E}^*}$ inductively by*

$$\mathcal{F}_{\langle T_i \rangle_{i\in I}}(T) = \begin{cases} \emptyset & \text{if } T = \emptyset \\ T' \cup \mathcal{F}_{\langle T_i \rangle_{i\in I'}}(T'') & \text{otherwise} \end{cases}$$

where $T' = \mathcal{H}_{\langle T_i \rangle_{i\in I}}(T)$, $T'' = \mathcal{G}_{\langle T_i \rangle_{i\in I}}(T)$, and $I' = I \setminus T'(\langle T_i\rangle_{i\in I})$.

The functions may as results give minimal complete test suites.

Theorem 25 *If $T \in \langle min(T^M_\alpha)\rangle_{\alpha\in\mathcal{E}}$ then $\mathcal{F}_{\langle T^M_\alpha \rangle_{\alpha\in\mathcal{E}}}(T)$ is minimal complete for $\mathcal{M}^M_{\mathcal{E}}$.*

Intuitively, a minimal complete test suite for a fault model is constructed by first selecting a set of tests T containing a smallest test for each fault in the fault model. The test for a fault is selected from the set of tests that characterizes the fault. Secondly, all the longest tests in T are collected and among those some tests may be eliminated resulting in a minimalized set of tests T' that will be contained in the resulting test suite. All faults in the model that may be detected by at least one of the tests in T' are disregarded and not considered further in the computation. The process is repeated for the faults still to be dealt with until no fault in the model needs consideration.

For instance, letting M be as defined in Fig. 3 and let T be $\{aa, ab, ac, acda, acdeaa\}$, then whenever $I = \{a, b, c, d, e\}$

$$\begin{aligned} & \mathcal{F}_{\langle T^M_i \rangle_{i\in I}}(T) \\ & \quad = \mathcal{H}_{\langle T^M_i \rangle_{i\in I}}(T) \cup \mathcal{F}_{\langle T^M_i \rangle_{i\in\{b\}}}(\mathcal{G}_{\langle T^M_i \rangle_{i\in I}}(T)) \\ & \quad = \{acdeaa\} \cup \mathcal{F}_{\langle T^M_\alpha \rangle_{\alpha\in\{b\}}}(\{ab\}) \\ & \quad = \{ab, acdeaa\} \end{aligned}$$

that is, first *acdeaa* is selected as the longest test and since the cover of *acdeaa* is $\{a, c, d, e\}$ the recursive call only has to consider the fault $M[b]$.

```
( 1) For each $s \in \mathcal{S}$; associate $(\sigma_s, |\sigma_s|)$ to $s$ where $\sigma_s$
     is a shortest test such that $s = {s_M}^{\sigma_s}$.
     The unreachable states are removed from $\mathcal{S}$.
( 2) For all $\alpha \in \mathcal{E}'$ and $s \in \mathcal{S}$; if $s$ is $\alpha$-active then
     associate $(\sigma_s\alpha, |\sigma_s\alpha|)$ to $\alpha$.
( 3) For each $\alpha \in \mathcal{E}'$; walk through the pairs associated
     to $\alpha$, select one $(\sigma_\alpha, i_\alpha)$ where $i_\alpha \leq i$ for any pair
     $(\sigma, i)$ associated to $\alpha$. If for some $\alpha \in \mathcal{E}'$ no pairs
     are associated to $\alpha$ let $(\sigma_\alpha, i_\alpha)$ be $(\epsilon, \infty)$. Let $\tilde{T}$
     be $\{(\sigma_\alpha, i_\alpha, \alpha) \mid \alpha \in \mathcal{E}'\}$.
( 4) Compute $\mathcal{B}^1$. For each $(s, s') \in \mathcal{B}^1$ associate a pair
     $(\alpha, 1)$ such that $s(\alpha) \neq s'(\alpha)$.
( 5) For $i = 2, \ldots, 2|\mathcal{S}| - 1$ compute $\mathcal{B}^i$. If $(s_1, s_2) \in \mathcal{B}^i$
     and no pair is associated to $(s_1, s_2)$ then associate
     $(\sigma\alpha, i)$ to it for some $\alpha$ where $(\sigma, i-1)$ is associated
     to $({s_1}^\alpha, {s_2}^\alpha) \in \mathcal{B}^{i-1}$.
( 6) For all $\alpha \in \mathcal{E}'$ and $s \in \mathcal{S}$; if some $(\sigma, i)$ is associated
     to $(s, s^\alpha)$ and if $|\sigma_s| + 1 + i < i'$ where $(\sigma', i', \alpha) \in \tilde{T}$
     then replace $(\sigma', i', \alpha)$ by $(\sigma_s\alpha\sigma, |\sigma_s| + 1 + i, \alpha)$.
( 7) If $i = \infty$ for some $(\sigma, i, \alpha) \in \tilde{T}$ then terminate with
     an error.
( 8) Compute a partition $\tilde{T}_{l_1}, \ldots, \tilde{T}_{l_n}$ of $\tilde{T}$ where $\tilde{T}_{l_i}$ is
     $\{(\sigma, l_i, \alpha) \mid (\sigma, l_i, \alpha) \in \tilde{T}\}$ and $l_1 < l_2 < \ldots < l_n$.
(9) For $i = n, \ldots, 1$; compute
     (a) $\mathcal{E}_i = \{\alpha \mid \exists\sigma.\ (\sigma, l_i, \alpha) \in \tilde{T}_{l_i}\}$, and
     (b) $T_{l_i} = \mu_{\langle T_\alpha^M \rangle_{\alpha \in \mathcal{E}_i}}(\{\sigma \mid \exists\alpha.\ (\sigma, l_i, \alpha) \in \tilde{T}_{l_i}\})$,
     (c) and remove from each $\tilde{T}_{l_j}$ where $l_j < l_i$ all
         $(\sigma, j, \alpha)$ if $\alpha \in T_{l_i}(\langle T_\alpha^M \rangle_{\alpha \in \mathcal{E}'})$.
(10) Return $T_{l_1} \cup \ldots \cup T_{l_n}$.
```

Figure 5: A polynomial time algorithm for computing minimal complete test suites.

7 Complexity

In this section we argue that the problem of computing a minimal complete test suite has polynomial time complexity. The correctness of the algorithms is due to Theorem 25, although the family of sets of tests that characterizes fault in the fault model is never explicitly computed, only a smallest test in each family member is searched for. Recall that in general it would be impossible to compute a set that characterizes a fault because the set may be infinite.

The algorithm is presented in Fig. 5. Its input is an FSM $M = (\mathcal{S}, \mathcal{E}, \Omega, \tau, s_M)$ and a set of inputs $\mathcal{E}' \subseteq \mathcal{E}$. In the first part on the algorithm (line 1-6) it computes a set $\tilde{T}$ such that whenever M is α-sensitive for all $\alpha \in \mathcal{E}'$ then

$$T = \{\sigma \mid \exists\alpha, i.\ (\sigma, i, \alpha) \in \tilde{T}\}$$

belongs to $\langle min(T_\alpha^M) \rangle_{\alpha \in \mathcal{E}'}$. If M is not α-sensitive for some $\alpha \in \mathcal{E}'$ then an error is reported in line 7 and the algorithm terminates, otherwise (line 8-10) $\mathcal{F}_{\langle T_\alpha^M \rangle_{\alpha \in \mathcal{E}'}}(T)$ is computed.

The reason why $T \in \langle min(T_\alpha^M) \rangle_{\alpha \in \mathcal{E}'}$ if M is α-sensitive for all $\alpha \in \mathcal{E}'$ is as follows. For all $\alpha \in \mathcal{E}'$, if there exists α-active states in $\mathcal{S}$ then after line 1-3 $\tilde{T}$ contains $(\sigma\alpha, |\sigma\alpha|, \alpha)$ where σ is a smallest test from s_M to an α-active state. Then in line 4-5, all inequivalent pairs of states are associated with a shortest test that distinguishes them. It is obvious that if $(s_1, s_2) \in \mathcal{B}^i$ for some i and if no pair is associated to (s_1, s_2) then $(s_1, s_2) \notin \mathcal{B}^j$ for all $j < i$. Hence $s_1 \approx_j s_2$ for all $j < i$ due to Lem. 4, so the test associated to (s_1, s_2) indeed is shortest possible. Note that according to Lem. 3 at most $2|\mathcal{S}| - 1$ iterations are needed in order to find a shortest test that distinguishes any two inequivalent states.

The information computed in line 4-5 is used in line 6 where $\tilde{T}$ may be updated. Updating takes place whenever $(\sigma', i', \alpha) \in \tilde{T}$ but for some s, (s, s^α) is associated with a test σ such that $|\sigma_s| + 1 + |\sigma| < i'$. Hence updating takes place in case there exists an α-distinguishable state s where s and s^α are distinguishable by some test σ such that $\sigma_s\alpha\sigma$ is shorter that the current test kept with α in $\tilde{T}$. If updating takes place then the last event in $\sigma_s\alpha\sigma$ cannot be α. [6]

Formally, in order to conclude that $\sigma \in min(T_\alpha^M)$ for any $(\sigma, i, \alpha) \in \tilde{T}$ we take advantage of the following lemma.

Lemma 26 *$min(T_1) = min(T_2)$ whenever*

$$T_1 = \{\sigma\alpha' \mid M(\sigma\alpha') \neq M(\sigma\alpha'{\downarrow}\alpha),\ \alpha' \neq \alpha\}$$

and

$$T_2 = \{\sigma_1\alpha\sigma_2\alpha' \mid M(\sigma_1\alpha\sigma_2\alpha') \neq M(\sigma_1\sigma_2\alpha'),\ \alpha' \neq \alpha\}$$

[6]Updating takes place only if $M(\sigma_s\alpha\sigma) \neq M(\sigma_s\sigma)$. Suppose in order to obtain a contradiction that $\sigma_s\alpha\sigma = \sigma_s\alpha\sigma'\alpha$ for some σ'. Then either ${s_M}^{\sigma_s\alpha\sigma'}$ or ${s_M}^{\sigma_s\sigma'}$ is an α-active state. However then $\sigma_s\alpha\sigma$ would not be shorter than the current test kept with α in $\tilde{T}$.

From Lem. 26 it then follows that $min(T_\alpha^M)$ may be defined as $min(T_1 \cup T_2)$ where $T_1 = \{\sigma\alpha \mid M(\sigma\alpha) \neq \emptyset\}$ and T_2 is $\{\sigma_1\alpha\sigma_2\alpha' \mid M(\sigma_1\alpha\sigma_2\alpha') \neq M(\sigma_1\sigma_2\alpha'),\ \alpha' \neq \alpha\}$. That is, any $\sigma \in min(T_\alpha^M)$ either is a shortest test to an α-active state to which α is appended (as computed in line 1-3) or a concatenation $\sigma\alpha\sigma'$ where σ is a shortest test from s_M to an α-distinguishable state s and where σ' is a shortest test, not ending with α, that distinguishes s and s^α (as computed in line 4-6).

The total time complexity of the algorithm is $O(|\mathcal{S}|^3|\mathcal{E}| + |\mathcal{S}|^2|\mathcal{E}||\Omega| + |\mathcal{S}||\mathcal{E}'|^2|\Omega|)$ (see Appendix A.8).

8 Conclusion

In this paper we have outlined the framework, connectivity testing, for the testing of embedded systems. In contrast to for instance conformance testing the approach is focused on detecting errors manifested as faults in the interface between the two system components: the hardware and the embedded software. Since we assume the software to be fault free the errors we address belong to the hardware, or probably software drivers not being part of the software model. Formally we choose to model errors by means of a fault model for disconnected inputs.

A test suite is complete if it detects all faults in the fault model, and if no test in the suite is useless. We have proven that the problem of computing a smallest possible complete test suite is NP-hard. Therefore we devised approximative greedy polynomial time algorithms that computes minimal complete test suites. A test suite is minimal complete if all its tests are needed in order to make it complete. Importantly, the test suites contain no more tests than the number of faults in the fault model and the length of any test in a test suites is bounded by twice the number of states in the FSM specification of the software (in practice the length may be much smaller of course as demonstrated by the experiments carried out).

Other types of fault models than the ones reported in this paper may be of interest. For instance, the single-fault fault models presented here may be generalized to multi-fault fault models. This kind of fault model are dealt with in [7], although no algorithms for test generation is provided there. Fault models and algorithms dealing with redirected inputs and outputs are presented in [9] and [8]. Mixing input and output faults is yet another research topic.

A Appendix

A.1 Proof of Lem. 6

Let M be an FSM with state set $\mathcal{S}$. Let $\mathcal{S}'$ be the set of states in $M[\alpha]$. By definition of $M[\alpha]$ there exists a $1:1$ mapping $f : \mathcal{S} \to \mathcal{S}'$, such that $f(s_M) = s_{M[\alpha]}$ and whenever $\alpha \neq \alpha'$ then $s \xrightarrow{\alpha'/o} s'$ is a transition in M only if $f(s) \xrightarrow{\alpha'/o} f(s')$ is a transition in $M[\alpha]$.

We prove that if $\alpha \neq \alpha'$ then $s(\sigma\alpha'{\downarrow}\alpha) = f(s)(\sigma\alpha')$ for all $s \in \mathcal{S}$. Hence in particular $s_M(\sigma\alpha'{\downarrow}\alpha) = s_{M[\alpha]}(\sigma\alpha')$ if $\alpha \neq \alpha'$ and therefore $M(\sigma\alpha'{\downarrow}\alpha) = M[\alpha](\sigma\alpha')$ if $\alpha \neq \alpha'$. The proof is by induction in the length of σ.

Let α' be some event such that $\alpha \neq \alpha'$ and let $s \in \mathcal{S}$.

Basis: $(\sigma = \epsilon)$ Consider $s \xrightarrow{\alpha'/o} s'$. Then $f(s) \xrightarrow{\alpha'/o} f(s')$ so $s(\alpha') = f(s)(\alpha')$ and since $\alpha'{\downarrow}\alpha = \alpha'$ it follows that $s(\alpha'{\downarrow}\alpha) = f(s)(\alpha')$.

Step: Suppose that $s(\sigma\alpha'{\downarrow}\alpha) = f(s)(\sigma\alpha')$ for all σ with $|\sigma| \leq n$ (IH). Let $\sigma = \alpha''\sigma'$ for some α'' and some σ' with $|\sigma'| = n$.

If $\alpha'' = \alpha$ then $\sigma\alpha'{\downarrow}\alpha = \sigma'\alpha'{\downarrow}\alpha$ so $s(\sigma\alpha'{\downarrow}\alpha) = s(\sigma'\alpha'{\downarrow}\alpha)$. Then by IH we obtain $s(\sigma'\alpha'{\downarrow}\alpha) = f(s)(\sigma'\alpha')$. Moreover, since $f(s)^{\alpha''} = f(s)$ by definition of $M[\alpha]$ it follows that $f(s)(\sigma\alpha') = f(s)(\sigma'\alpha')$ and therefore $s(\sigma\alpha'{\downarrow}\alpha) = f(s)(\sigma\alpha')$.

If $\alpha'' \neq \alpha$ then consider $s^{\alpha''}$. Due to the property of f, $f(s)^{\alpha''} = f(s^{\alpha''})$. Then by IH, $s'(\sigma'\alpha'{\downarrow}\alpha) = f(s')(\sigma'\alpha')$ and therefore $s(\sigma\alpha'{\downarrow}\alpha) = f(s)(\sigma\alpha')$. ∎

A.2 Proof of Lem. 9

(If) Suppose M is α-sensitive. If M is α-active then $M(\sigma\alpha) \neq \emptyset$ for some σ. However, since $M[\alpha](\sigma\alpha) = \emptyset$ then $M \not\approx M[\alpha]$. If M is α-distinguishable then for some σ, ${s_M}^{\sigma} \not\approx {s_M}^{\sigma\alpha}$. Suppose, in order to obtain a contradiction, that $M \approx M[\alpha]$. Then $s_M \approx s_{M[\alpha]}$, so ${s_M}^{\sigma} \approx {s_{M[\alpha]}}^{\sigma}$, and ${s_M}^{\sigma\alpha} \approx {s_{M[\alpha]}}^{\sigma\alpha}$. By definition of $M[\alpha]$, ${s_{M[\alpha]}}^{\sigma} = {s_{M[\alpha]}}^{\sigma\alpha}$ and then, since $\approx$ is an equivalence relation, ${s_M}^{\sigma} \approx {s_M}^{\sigma\alpha}$. Hence we obtain a contradiction.

(Only if) Suppose M is not α-sensitive. That is, M is neither α-active nor α-distinguishable. Let f be the mapping introduced in Sec. A.1. It is sufficient to prove for all $s \in \mathcal{S}$ that $s(\sigma) = f(s)(\sigma)$ for all σ. The proof is by induction in $|\sigma| \geq 1$.

Basis: ($|\sigma| = 1$) Let s be any state and let $\sigma = \alpha'$ for some α'. If $\alpha' \neq \alpha$ then due to the properties of f, $s(\alpha') = f(s)(\alpha')$. If $\alpha' = \alpha$ then because M is not α-active it follows that $s(\alpha') = \emptyset$. From the definition of $M[\alpha]$ we conclude that also $f(s)(\alpha') = \emptyset$.

Step: Assume for all s and for all σ with $|\sigma| < n + 1$ that $s(\sigma) = f(s)(\sigma)$ (IH). Let $s \in \mathcal{S}$ and let $\sigma = \alpha'\sigma'$ for some α' and some σ' with $|\sigma'| = n$. If $\alpha' \neq \alpha$ then due to the property of f, $f(s)^{\alpha'} = f(s^{\alpha'})$. Hence, since $s(\sigma) = s^{\alpha'}(\sigma')$ and since by IH $s^{\alpha'}(\sigma') = f(s^{\alpha'})(\sigma')$, it follows that $s(\sigma) = f(s)(\sigma)$. If $\alpha' = \alpha$ then because M is not α-distinguishable it must be that $s \approx s^{\alpha'}$. From the definition of $M[\alpha]$ we infer that $f(s) = f(s)^{\alpha'}$. It then follows that $s(\sigma) = s(\sigma')$ and that $f(s)(\sigma) = f(s)(\sigma')$. Hence, since by IH $s(\sigma') = f(s)(\sigma')$, we conclude that $s(\sigma) = f(s)(\sigma)$. ∎

A.3 Proof of Lem. 14

Suppose $apply_M(M[\alpha], \sigma) = fail$. Then for some $\sigma' \preceq \sigma$, $M(\sigma') \neq M[\alpha](\sigma')$. Hence from Corollary 7 it follows that $\sigma' \neq \sigma'{\downarrow}\alpha$, so σ' must contain at least one α. Therefore σ' can be partitioned into $\sigma_1\alpha\sigma_2$ for some σ_1 and σ_2 where $\sigma_2 = \sigma_2{\downarrow}\alpha$. Then, if $\sigma_2 = \epsilon$ it must be that $M(\sigma_1\alpha) \neq \emptyset$ because $M[\alpha](\sigma_1\alpha) = \emptyset$. Hence, $\sigma' \in T^M_\alpha$. If $\sigma_2 \neq \epsilon$ then since $M[\alpha](\sigma_1\alpha\sigma_2) = M(\sigma_1\alpha\sigma_2{\downarrow}\alpha)$, due to Lem. 6, it follows that $\sigma_1\alpha\sigma_2 \in T^M_\alpha$.

Suppose $\sigma' \preceq \sigma$ for some $\sigma' \in T^M_\alpha$. If $\sigma' = \sigma_1\alpha$ and $M(\sigma_1\alpha) \neq \emptyset$ then clearly $M(\sigma') \neq M[\alpha](\sigma')$, so $apply_M(M[\alpha], \sigma) = fail$. Otherwise, $\sigma' = \sigma_1\alpha'$ and $\alpha \neq \alpha'$ such that $M(\sigma') \neq M(\sigma'{\downarrow}\alpha)$, but then since $M[\alpha](\sigma') = M(\sigma'{\downarrow}\alpha)$, due to Lem. 6, it follows that $apply_M(M[\alpha], \sigma) = fail$. ∎

A.4 Proof of Lem. 23

Follows from the definitions of the functions $\mathcal{G}_{\langle T_i\rangle_{i\in I}}$ and $\mathcal{H}_{\langle T_i\rangle_{i\in I}}$, and the definition of a cover of a set of tests with respect to a family $\langle T_i\rangle_{i\in I}$. ∎

A.5 Proof of Th. 25

Let $\mathcal{M}^M_{\mathcal{E}'}$ be a fault model and let T belong to $\langle min(T^M_\alpha)\rangle_{\alpha\in\mathcal{E}'}$. The proof is by induction in the size of $\mathcal{E}'$.

Basis: If $\mathcal{E}' = \emptyset$ the theorem holds vacuously.

Step: Let $\mathcal{E}' \neq \emptyset$. Assume for all $\mathcal{E}'' \subset \mathcal{E}'$ that whenever $T' \in \langle min(T^M_\alpha)\rangle_{\alpha\in\mathcal{E}''}$ then $\mathcal{F}_{\langle T^M_\alpha\rangle_{\alpha\in\mathcal{E}''}}(T')$ is minimal complete for $\mathcal{M}^M_{\mathcal{E}''}$ (Induction Hypothesis).

Soundness is preserved by subsets. Hence soundness of $\mathcal{F}_{\langle T^M_\alpha\rangle_{\alpha\in\mathcal{E}'}}(T)$ follows because T is sound due to Theorem 17 and because $\mathcal{F}_{\langle T^M_\alpha\rangle_{\alpha\in\mathcal{E}'}}(T) \subseteq T$.

In the rest of the proof we let $T' = \mathcal{H}_{\langle T^M_\alpha\rangle_{\alpha\in\mathcal{E}'}}(T)$, $T'' = \mathcal{G}_{\langle T^M_\alpha\rangle_{\alpha\in\mathcal{E}'}}(T)$, and $\mathcal{E}'' =$

$\mathcal{E}' \setminus T'(\langle T_\alpha^M \rangle_{\alpha\in\mathcal{E}'})$. Due to Lem. 23 if follows that $T'' \in \langle min(T_\alpha^M)\rangle_{\alpha\in\mathcal{E}''}$.

In order to prove exhaustiveness let $M[\alpha] \in \mathcal{M}_{\mathcal{E}'}^M$. Hence $\alpha \in \mathcal{E}'$. Then either $\alpha \in \mathcal{E}''$ or $\alpha \in T'(\langle T_\alpha^M \rangle_{\alpha\in\mathcal{E}'})$. In the latter case, for some $\sigma \in T'$ there exists $\sigma' \preceq \sigma$ with $\sigma' \in T_\alpha^M$, so $apply_M(M[\alpha],\sigma) = fail$ for some $\sigma \in T'$ due to Lem. 14. In the former case $M[\alpha] \in \mathcal{M}_{\mathcal{E}''}^M$ and because $T'' \in \langle min(T_\alpha^M)\rangle_{\alpha\in\mathcal{E}''}$ it follows by induction that $\mathcal{F}_{\langle T_\alpha^M\rangle_{\alpha\in\mathcal{E}''}}(T'')$ is exhaustive, hence for some $\sigma \in \mathcal{F}_{\langle T_\alpha^M\rangle_{\alpha\in\mathcal{E}''}}(T'')$ it must be that $apply_M(M[\alpha],\sigma) = fail$

Finally, we prove minimality. The proof is by contradiction.

Assume for some $T_0 \subset \mathcal{F}_{\langle T_\alpha^M\rangle_{\alpha\in\mathcal{E}'}}(T)$ that T_0 is complete for $\mathcal{M}_{\mathcal{E}'}^M$. Moreover, let $\sigma_0 \in \mathcal{F}_{\langle T_\alpha^M\rangle_{\alpha\in\mathcal{E}'}}(T) \setminus T_0$. Since $\sigma_0 \in \mathcal{F}_{\langle T_\alpha^M\rangle_{\alpha\in\mathcal{E}'}}(T)$ it follows by the definition of $\mathcal{F}_{\langle T_\alpha^M\rangle_{\alpha\in\mathcal{E}'}}$ that either $\sigma_0 \in T'$ or $\sigma_0 \in \mathcal{F}_{\langle T_\alpha^M\rangle_{\alpha\in\mathcal{E}''}}(T'')$. We prove that both cases leads to a contradiction.

Suppose $\sigma_0 \in T'$. Then because $T' \in \mu\langle T_\alpha^M\rangle_{\alpha\in\mathcal{E}_1}$, where $\mathcal{E}_1$ is

$$\{\alpha \in \mathcal{E}' \mid \exists\sigma \in max(T).\ \sigma \in min(T_\alpha^M)\}$$

it follows that $T' \cap min(T_\alpha^M) = \{\sigma_0\}$ for some $\alpha \in \mathcal{E}_1$. Since T_0 is assumed to be complete there must exists $\sigma_1 \in T_0$ such that $apply_M(M[\alpha],\sigma_1) = fail$. Hence for some $\sigma_2 \preceq \sigma_1$, $\sigma_2 \in T_\alpha^M$. Then $|\sigma_0| \leq |\sigma_2|$ because $\sigma_0 \in min(T_\alpha^M)$, but also $|\sigma_0| \geq |\sigma_1|$ because $\sigma_1 \in T$ and $\sigma_0 \in max(T)$. Therefore $\sigma_1 = \sigma_2$ and $|\sigma_0| = |\sigma_1|$. Since $T_0 \subset \mathcal{F}_{\langle T_\alpha^M\rangle_{\alpha\in\mathcal{E}'}}(T)$ it must be that either $\sigma_1 \in T'$ or $\sigma_1 \in \mathcal{F}_{\langle T_\alpha^M\rangle_{\alpha\in\mathcal{E}''}}(T'')$. However, $\sigma_1 \notin T'$ because $T' \cap min(T_\alpha^M) = \{\sigma_0\}$. Also, $\sigma_1 \notin \mathcal{F}_{\langle T_\alpha^M\rangle_{\alpha\in\mathcal{E}''}}(T'')$ because $\sigma_1 \in max(T)$ and $max(T) \cap \mathcal{F}_{\langle T_\alpha^M\rangle_{\alpha\in\mathcal{E}''}}(T'') = \emptyset$. Hence we obtain a contradiction.

If $\sigma_0 \in \mathcal{F}_{\langle T_\alpha^M\rangle_{\alpha\in\mathcal{E}''}}(T'')$, let $\mathcal{M}$ be

$$\{M' \in \mathcal{M}_{\mathcal{E}''}^M \mid apply_M(M',\sigma_0) = fail\}$$

Then in order for T_0 to be exhaustive, it must be that for all $M' \in \mathcal{M}$, there exists $\sigma_{M'} \in T_0$ such that $apply_M(M',\sigma_{M'}) = fail$. Because of Lem. 14 and the definition of $\mathcal{E}''$ if follows that $\sigma_{M'} \notin T'$ for all $M' \in \mathcal{M}$. Hence, since T_0 is assumed to be exhaustive, it must be that for all $M' \in \mathcal{M}$, $\sigma_{M'} \in \mathcal{F}_{\langle T_\alpha^M\rangle_{\alpha\in\mathcal{E}''}}(T'')$. But then we have a contradiction since by induction $\mathcal{F}_{\langle T_\alpha^M\rangle_{\alpha\in\mathcal{E}''}}(T'')$ is minimal complete since $T'' \in \langle min(T_\alpha^M)\rangle_{\alpha\in\mathcal{E}''}$. ∎

A.6 Proof of Lem. 26

Let $T_1 = \{\sigma\alpha' \mid M(\sigma\alpha') \neq M(\sigma\alpha'{\downarrow}\alpha),\ \alpha' \neq \alpha\}$ and let

$$T_2 = \{\sigma_1\alpha\sigma_2\alpha' \mid M(\sigma_1\alpha\sigma_2\alpha') \neq M(\sigma_1\sigma_2\alpha'),\ \alpha' \neq \alpha\}$$

It is sufficient to prove that $min(T_1) \subseteq T_2$ and that $min(T_2) \subseteq T_1$. [7]

We first prove that $min(T_1) \subseteq T_2$. If $T_1 = \emptyset$ we are done so suppose $T_1 \neq \emptyset$. Let $\sigma \in min(T_1)$. Then $\sigma = \sigma'\alpha'$ for some σ' and α' where $\alpha \neq \alpha'$. Also, $\sigma = \sigma_1\alpha\sigma_2\alpha'$ for some σ_1 and σ_2 because $\sigma \neq \sigma{\downarrow}\alpha$ since $M(\sigma) \neq M(\sigma{\downarrow}\alpha)$. Suppose, in order to obtain a contradiction, that $\sigma \notin T_2$. Then $M(\sigma) = M(\sigma_1\sigma_2\alpha')$ for all σ_1 and σ_2 where $\sigma = \sigma_1\alpha\sigma_2\alpha'$. Now, let σ_1 and σ_2 be such that $\sigma = \sigma_1\alpha\sigma_2\alpha'$. Obviously, $\sigma{\downarrow}\alpha = \sigma_1\sigma_2\alpha'{\downarrow}\alpha$ so $M(\sigma{\downarrow}\alpha) = M(\sigma_1\sigma_2\alpha'{\downarrow}\alpha)$. But then, since $M(\sigma) \neq M(\sigma{\downarrow}\alpha)$, it follows that $M(\sigma_1\sigma_2\alpha') \neq M(\sigma_1\sigma_2\alpha'{\downarrow}\alpha)$. However, then $\sigma_1\sigma_2\alpha' \in T_1$ and we obtain a contradiction because $|\sigma_1\sigma_2\alpha'| < |\sigma|$.

Next we prove that $min(T_2) \subseteq T_1$. If $T_2 = \emptyset$ we are done so suppose $T_2 \neq \emptyset$. Let $\sigma \in min(T_2)$. Then $\sigma = \sigma_1\alpha\sigma_2\alpha'$ for some σ_1, σ_2, and α' where $\alpha \neq \alpha'$. If $\sigma{\downarrow}\alpha = \sigma_1\sigma_2\alpha'$ it follows by the definition of T_2 that $M(\sigma) \neq M(\sigma{\downarrow}\alpha)$ and hence $\sigma \in T_1$. Suppose instead that $\sigma{\downarrow}\alpha \neq \sigma_1\sigma_2\alpha'$. Let

[7] That for any two sets of tests T and T', $min(T) \subseteq T'$ and $min(T') \subseteq T$ implies $min(T) = min(T)$ can be shown as follows. Let $min(T) \subseteq T'$ and let $min(T') \subseteq T$. Suppose, $min(T) \neq min(T)$. Hence, without loss of generality, there exists $\sigma \in min(T)$ such that $\sigma \notin min(T')$. However, since $\sigma \in T'$ it must be that there exists $\sigma' \in min(T')$ such $|\sigma'| < |\sigma|$. But, then since $\sigma' \in T$ and because $\sigma \in min(T)$ we obtain a contradiction.

$\sigma_1 = \sigma_{11}\alpha\sigma_{12}\alpha\ldots\alpha\sigma_{1m}$ for some $\sigma_{11},\ldots,\sigma_{1m}$, where $\sigma_{1i} = \sigma_{1i}{\downarrow}\alpha$ for $i = 1,\ldots,m$. Let $\sigma_2 = \sigma_{21}\alpha\sigma_{22}\alpha\ldots\alpha\sigma_{2n}$ for some $\sigma_{21},\ldots,\sigma_{2n}$, where $\sigma_{2i} = \sigma_{2i}{\downarrow}\alpha$ for $i = 1,\ldots,n$. Since either $\sigma_1 \neq \sigma_1{\downarrow}\alpha$ or $\sigma_2 \neq \sigma_2{\downarrow}\alpha$ it must be that either $m > 1$ or $n > 1$. Next, let

$$\begin{aligned}
\sigma'_0 &= \sigma_{11}\sigma_{12}\ldots\sigma_{1m}\sigma_{21}\sigma_{22}\ldots\sigma_{2n}\alpha' \\
\sigma'_1 &= \sigma_{11}\alpha\sigma_{12}\ldots\sigma_{1n}\sigma_{21}\sigma_{22}\ldots\sigma_{2n}\alpha' \\
&\vdots \\
\sigma'_{n+m-2} &= \sigma_{11}\alpha\sigma_{12}\alpha\ldots \\
&\quad\ldots\alpha\sigma_{1n}\sigma_{21}\alpha\sigma_{22}\alpha\ldots\alpha\sigma_{2n}\alpha'
\end{aligned}$$

that is, σ'_i is $\sigma_1\sigma_2\alpha'$ where the last $n+m-i-2$ α's have been removed. Note that $\sigma'_0 = \sigma{\downarrow}\alpha$ and that $\sigma'_{n+m-2} = \sigma_1\sigma_2\alpha'$. Since $|\sigma'_i| < |\sigma|$ and because $\sigma \in min(T_2)$ it must be that $M(\sigma'_i) = M(\sigma'_{i-1})$ for all $i = 1,\ldots,n+m-2$. Hence $M(\sigma'_0) = M(\sigma'_{n+m-2})$ and consequently $M(\sigma{\downarrow}\alpha) = M(\sigma_1\sigma_2\alpha')$. Therefore, since $M(\sigma) \neq M(\sigma_1\sigma_2\alpha')$, $M(\sigma) \neq M(\sigma{\downarrow}\alpha)$ and hence $\sigma \in T_1$. ■

A.7 Proof of Th. 18

The proof consists of establishing a polynomial deterministic time reduction from the NP-complete Directed Hamiltonian-Cycle problem (see e.g. [6]). Actually, the reduction is to the decision problem: *Does $\mathcal{M}^M_{\mathcal{E}}$ have a smallest complete test suite of size k?* However, since if it is easy to compute a smallest complete test suite then it is also easy to determine the size of such a test suite, and hence to decide the decision problem. Or the other way around, if the decision problem is NP-hard then so is the problem of determining a smallest complete test suite.

Let $G = (V, E)$ be a directed graph. Let $\mathcal{E} = \{\alpha_v \mid v \in V\}$ such that if $u \neq v$ then $\alpha_u \neq \alpha_v$. Construct G' by modifying G such that all edges (u, v) are labelled α_v. Hence any edge entering a node v is labelled by α_v. Also, for any vertices u and v if $(u, v) \notin E$ then add an edge from u to u labelled by α_v. Choose some vertex $v_{G''} \in V$ and let $\{e_1,\ldots,e_n\}$ be the edges in G' that enter $v_{G''}$ and that are labelled by $\alpha_{v_{G''}}$. Let $\Omega = \{\omega_1,\ldots,\omega_n\}$. Let G'' be G' where $e_1,\ldots,e_n$ are labelled by $\{\omega_1\},\ldots,\{\omega_n\}$ respectively and where all other edges are labelled by the empty set. Observe that $G'' = (V, E'')$ may be considered an FSM $M_{G''} = (V, \mathcal{E}, \Omega, E'', v_{G''})$.

Clearly, G'' can be constructed from G in polynomial deterministic time. The correctness of the reduction follows from Lem. 27.

Lemma 27 *G contains a Hamilton-cycle iff there exists a smallest complete test suite for $\mathcal{M}^{M_{G''}}_{\mathcal{E}}$ of size $|V|$.*

Proof: Suppose G contains a Hamilton-cycle cycle $v_1, v_2, \ldots, v_{|V|}, v_1$. Assume (without loss of generality) that $v_1 = v_{G''}$. Then in G'' (leaving out the output labels)

$$v_1 \xrightarrow{\alpha'_1} v_2 \xrightarrow{\alpha'_2} \ldots v_{|V|} \xrightarrow{\alpha'_{|V|}} v_1$$

for some events $\alpha'_1, \alpha'_2, \ldots, \alpha'_{|V|}$. Due to the construction of G'', $\mathcal{E} = \{\alpha'_1, \ldots, \alpha'_{|V|}\}$. Let $\sigma = \alpha'_1\alpha'_2\ldots\alpha'_{|V|}$. Then $\sigma \in T^{M_{G''}}_{\alpha'_{|V|}}$ because $v_{G''}(\sigma) \neq \emptyset$ since $\alpha'_{|V|} = \alpha'_{v_{G''}}$. Also, because edges in G'' that are labelled by $\alpha_{v_{G''}}$ all are labelled by distinct outputs or the empty output it turns out that for all $\alpha \in \mathcal{E}$, if $\alpha \neq \alpha'_{|V|}$ then $v_{G''}(\sigma) \neq v_{G''}(\sigma{\downarrow}\alpha)$. Hence σ cover $\mathcal{E}$ wrt. $\langle T^{M_{G''}}_{\alpha}\rangle_{\alpha\in\mathcal{E}}$ so according to Lem. 16 $\mathcal{M}^{M_{G''}}_{\mathcal{E}}$ is a fault model. From Theorem 17 it follows that $\{\sigma\}$ is complete for $\mathcal{M}^{M_{G''}}_{\mathcal{E}}$. Clearly, $\{\sigma\}$ is a smallest complete test suite because all $\alpha \in \mathcal{E}$ must occur in a test in order for a test suite to be complete.

Suppose $\mathcal{M}^{M_{G''}}_{\mathcal{E}}$ is a fault model with a smallest test suite T of size $|V|$. Then T covers $\mathcal{E}$ wrt. $\langle T^{M_{G''}}_{\alpha}\rangle_{\alpha\in\mathcal{E}}$ due to Lem. 16. Hence for all $\alpha \in \mathcal{E}$ there is a test in T containing α. Then since $|T| = |\mathcal{E}|$ and because only the event $\alpha_{v_{G''}}$ may cause output T contains only one test σ. Finally, in order for σ to contain all $\alpha \in \mathcal{E}$ it must be that σ visits all vertices in G'', so therefore G contains a Hamilton-cycle. ■

A.8 Complexity

Consider the algorithm in Fig. 5. Line 1 is $O((|\mathcal{S}| + |\tau|) \lg |\mathcal{S}|)$ if a modified version of Dijkstra's single-source shortest paths algorithm (see e.g. [6]) is applied. Recall the functionality of τ, then the complexity of line 1 can be rewritten as $O(|\mathcal{S}||\mathcal{E}| \lg |\mathcal{S}|)$. Line 2 is $O(|\mathcal{S}||\mathcal{E}'||\Omega|)$ and both line 3 and 6 are $O(|\mathcal{S}||\mathcal{E}'|)$. Line 4 is $O(|\mathcal{S}|^2|\mathcal{E}||\Omega|)$ because $s(\alpha) \neq s'(\alpha)$ is $O(|\Omega|)$. For $i > 2$, computing $\mathcal{B}^i$ is $O(|\mathcal{S}|^2|\mathcal{E}|)$, hence line 5 is $O(|\mathcal{S}|^3|\mathcal{E}|)$. Clearly, line 7, 8 and 10 are $O(|\mathcal{E}'|)$. The minimalizations in line 9 can be performed as follows: For each $(\sigma, l_i, \alpha) \in \tilde{T}_{l_i}$ in turn, remove it if $\sigma' \in T_\alpha^M$ for some other triple $(\sigma', l_i, \alpha') \in \tilde{T}_{l_i}$. Checking $\sigma' \in T_\alpha^M$ is $O(l_i + |\Omega|)$ because we have to check if $M(\sigma') \neq \emptyset$ and $\sigma' = \sigma''\alpha$ for some σ'' or if $\sigma' \neq \sigma''\alpha$ for some σ'' and $M(\sigma') \neq M(\sigma'{\downarrow}\alpha)$. Hence the complexity of all the minimalizations is $O(\Sigma_{i=1}^n(|T_{l_i}|^2(l_i+|\Omega|)))$ which is $O(\Sigma_{i=1}^n(|T_{l_i}|^2(|\mathcal{S}| + |\Omega|)))$ becase l_i is $O(|\mathcal{S}|)$. Then because $O(\Sigma_{i=1}^n(|T_{l_i}|^2)) = O(|\mathcal{E}'|^2)$ it turns out that the complexity of the minimalizations is $O(|\mathcal{E}'|^2(|\mathcal{S}| + |\Omega|))$. The cover of T_{l_i} is $O(l_i|T_{l_i}||\mathcal{E}'||\Omega|)$: for each $\sigma \in T_{l_i}$ and $\alpha \in \mathcal{E}'$ we must check for all σ' where $\sigma'\alpha \preceq \sigma$ whether $M(\sigma'\alpha) \neq \emptyset$ and we must check for all σ' where $\sigma'\alpha' \preceq \sigma$ and $\alpha \neq \alpha'$ whether $M(\sigma'\alpha') \neq M(\sigma'\alpha'{\downarrow}\alpha)$. Hence the complexity of computing all the covers is $O(\Sigma_{i=1}^n l_i|T_{l_i}||\mathcal{E}'||\Omega|)$ which is $O(|\mathcal{S}||\mathcal{E}'|^2|\Omega|)$. All the removals are $O(\Sigma_{i=1}^n\Sigma_{j=1}^{i-1}|\tilde{T}_{l_j}|)$ which is $O(|\mathcal{E}'|^2)$. Hence the complexity of line 9 is $O(|\mathcal{S}||\mathcal{E}'|^2|\Omega|)$. The total complexity of the algorithm thus is $O(|\mathcal{S}|^3|\mathcal{E}| + |\mathcal{S}|^2|\mathcal{E}||\Omega| + |\mathcal{S}||\mathcal{E}'|^2|\Omega|)$. ∎

References

[1] M. Abramovici, M.A. Breuer, and A.D. Friedman. *Digitial System Design and Testable Design.* Computer Science Press, 1990.

[2] A.V. Aho, A.T. Dahbura, D. Lee, and M.U Uyar. An optimization technique for protocol conformance test generation based on uio sequences and rural chinese postman tours. *IEEE Trans. on Comm.*, 39(11):1604–1615, Nov. 1991.

[3] E. Brinksma, R. Alderen, R. Langerak, J. van de Lagemaat, and J. Tretmans. A formal approach to conformance testing. *2th Int. Workshop on Protocol Test Systems*, pages 349–363. Elsevier Science Publishers B.V., 1990.

[4] R. Cardell-Oliver and T. Glover. A practical and complete algorithm for testing real-time systems. *FTRTFT'98*, volume 1486 of *LNCS*, Lyngby, Denmark, Sep. 1998. Springer–Verlag.

[5] T.S Chow. Testing software desing models by finite-state machines. *IEEE Transactions on Software Engeneering*, 4(3):178–187, 1978.

[6] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, 1997.

[7] J.C. Godskesen. Fault models for embedded systems. *CHARME'99*, volume 1703 of *LNCS*, Bad Herrenalb, Sep. 1999. Springer–Verlag.

[8] J.C. Godskesen. Test generation for embedded systems with redirected inputs. *Proc. of WS-EST'99*, NIST, Gaithersburg, USA, Nov. 1999.

[9] J.C. Godskesen. Test generation for embedded systems with redirected outputs. In *Proceedings of HLDVT'99*, San Diego, November 1999.

[10] J.C. Godskesen. Two algorithms for generating tests for embedded systems. In *Proceedings of ISCIS'99*, Kusadasi, October 1999.

[11] G.J. Holzmann. *Design and Validation of Computer Protocols.* Series in Computer Science. Prentice–Hall, Englewood Cliffs, NJ, 1991.

[12] Z. Kohavi. *Switching and Finite Automata Theory.* McGraw-Hill, 1978.

[13] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines– a survey. *Proceedings of the IEEE*, 84(8), August 1996.

[14] A. Petrenko and N. Yevtushenko. Fault detection in embedded components. *10th Int. Workshop on Testing of Comm. Systems*, Cheju Island, Korea, Sep. 1997. Chapman & Hall.

[15] A. Petrenko, N. Yevtushenko, G. v. Bochmann. Fault models for testing in context. *FORTE/PSTV'96*, Univ. of Kaiserslautern, Dept. of Inform., Oct. 1996. Chapman & Hall.

[16] J.G. Springintveld, F.W. Vaandrager, P.R. D'Argenio. Testing timed automata. Tech. Rep. CSI-R9712, Univ. of Nijmegen, Aug. 1997.

[17] J. Tretmans. *A Formal Approach to Conformance Testing.* PhD thesis, University of Twente, 1992.

A Simple Approach to Testing Timed Systems

Sébastien Salva Eric Petitjean Hacène Fouchal

LERI-*RESYCOM*
Université de Reims Champagne-Ardenne,
UFR Sciences Exactes et Naturelles,
Moulin de la Housse, BP 1039,
51687 Reims Cedex 2, France
e-mail:{Sebastien.Salva, Eric.Petitjean, Hacene.Fouchal}@univ-reims.fr

Abstract

Conformance testing of timed systems is still a new field even it could help many designer since a long time. This study presents an new approach to testing complex systems having timing constraints. This technique aims to prove if any timed system implementation respects some properties described by timed test purposes. The region graph model (generated from timed automata model) is used for the specification of timed systems. The kernel of this method is the extraction of executable timed test cases from the user timed test purposes. Then, these test cases will be submitted to the implementation, by means of a specific architecture, and the implementation reactions indicate us whether the implementation achieves the timed test purpose or not.

Key-words : Timed automata, region graph, conformance testing, real-time systems, test purpose, test execution.

1 Introduction

The design of complex and critical systems becomes more and more difficult and very expensive. Before industrial development, systems need to be tested. A large part of the development effort is spent in this last step. For the near future, the main challenge is to provide a solid theoretical framework as well as a useful, practical and automated tools dedicated to testing systems (especially with their behavioral and temporal aspects). In the last decade, some studies and tools have been used in practice [BFG$^+$00, CH96], but most of them are only used for academic experiments [CKM92, CGPT95].

This paper copes with the conformance testing of complex timed systems. Contrary to many other studies on timed testing, we use the region graph model [AD94]. The main disadvantage is the generation of a region graph which is a hard task (exponential complexity on the number of clocks). But we prefer to use it since it has a high expression power for timed systems and can be used in practice if the number of clocks is low. We believe that real systems can be specified by a reduced number of clocks [DY96]. Then, we extract executable timed test cases from timed test purposes. A timed test purpose is a sequence of events required by a designer with his own timing constraints. The extraction of timed test cases is performed by a synchronous product between the test purpose and the specification. The main aim of this product is to select the periods of the test purpose which can be found on the specification for the execution of an action. We show the execution of the timed test cases on the implementation. This step is performed by building an execution tree on which we highlight the set of actions to execute, the set of actions to observe and their appropriate sets of moments. Our main care is to cover the entire region where an action has to be submitted to the implementation, then the action will be submitted at the starting moment of the region graph and at the ending moment of the region. All these steps (except the execution) are being implemented in a prototype timed testing tool.

This paper is structured as follows: the main works about the timed testing field are cited in Section 2. In this section, we present also the usual models on timed systems. Section 3 details all the main aspects about the derivation of timed test cases. Section 4 is dedicated to the execution of test cases on the implementation. The complexity of this methodology is expressed in section 5. Section 6, gives the conclusion and some ideas about future works.

2 Background

The conformance testing on timed systems is still a new field. We will give a brief overview on some studies of this area. We will talk about two models for timed systems used since a decade.

2.1 Related work

There are many works dedicated to the verification of timed automata [ACH94, DOY94, DY95]. Some tools [DOTY95, BLL$^+$98] have been developed for this purpose. But some other studies proposed various testing techniques for timed systems. [Kon95] deals with an adaptation of the canonical tester for timed testing and it has been extended in [LC97]. In [CL97], the authors derive test cases from specifications described in the form of a constraint graph. They only consider the minimum and the maximum allowable delays between input/output events. [SVD01] gives a general outline and a theoretical framework for timed testing based of the adaptation of the classical testing techniques to timed systems. [ENDKE98] presents an adaptation of the Wp-method [FBK$^+$91] to timed systems to a reduced model of timed automata (grid automata). [COG98] presents a specific testing technique which suggests a practical algorithm for test generation. [RNHW98] gives a particular method for the derivation of the more relevant inputs of the systems. [PF99a] suggests a technique for translating a region graph into a graph where timing constraints are expressed by specific labels using clock zones. The last study [NS01], suggests a selection technique of timed tests from a restricted class of dense timed automata specifications. It is based on the well known testing theory proposed by Hennessy in [DNH84]. As we can notice, there are different ways to tackle the problem of timed testing. All of these studies focuss on reducing the specification formalism in order to be able to derive test cases feasible in practice. In contrast to these studies, we use thw region graph model but we don't extract all possible test cases we only consider test cases derived from test purposes given by the user.

2.2 Timed system models

2.2.1 Timed automata model

Timed automata [AD94] are graphs representing timed systems during their executions. To represent time in a timed system, a set of clocks is associated to the automaton. Each clock is represented by a real value (dense time representation) and grows strictly monotonically. All clocks are set to 0 in the initial state. Clocks can be reset on any transition. To execute a transition, all the clocks of the system must satisfy the transition constraints.

Timed Input Output Automata (TIOA) are extended timed automata where actions are divided into input ones and output ones.

A transition $S_i \xrightarrow{?x} S_j$, labeled by the input action "?x" models an input action submitted by the environment. A transition $S_i \xrightarrow{!x} S_j$, labeled by the output action "!x", represents an output action generated by the implementation.

An example of TIOA is illustrated in Figure 1.

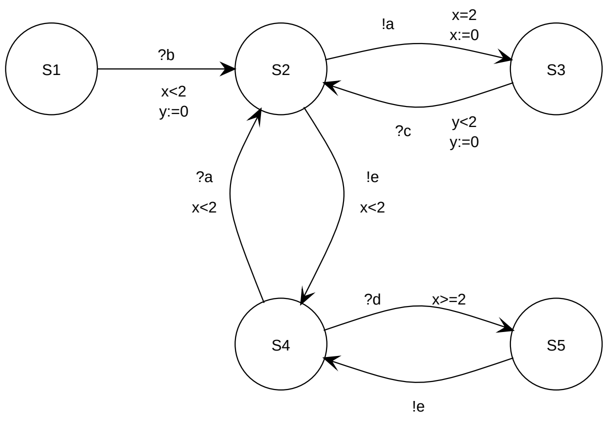


Figure 1: An example of timed automaton

2.2.2 Region graph model

This model has been formally defined in [AD94]. A Region Graph is an equivalent representation of a timed automaton where a state collects all the moments where the system has the same behaviour. Clearly, a region graph state is composed of a timed automaton state (representing the system behaviour) and a clock region (it is a polyhedron representing the inequations of the state timing constraints). Finally, we can say that in region graphs the timing constraints are moved to states. The transformation algorithm of timed automata into region graphs is defined in [AD94]. The theoretical framework about this model is detailed in [AD94].

Definition 2.1 (A region graph) *A region graph RG is a tuple* $(\Sigma_{RG}, S_{RG}, s^0_{RG}, R_{RG}, C_{RG}, E_{RG})$ *where* Σ_{RG} *is the set of actions,* S_{RG} *is the set of states,* s^0_{RG} *is the initial state,* R_{RG} *is the set of clock regions of RG,* C_{RG} *is the set of clocks,* E_{RG} *is the transition relation defined as:*

- (s, s', a) *from state* s *to state* s'*, labeled with the symbol* a*.* s *is a tuple* (x, R) *where* x *is a state of the initial timed automaton and* R *is the clock region during which* a *can be executed.* s' *is a tuple* (x', R') *where* x' *is a state of the initial timed automaton and* R' *is the reached clock region.*

- (s, s', δ) *from state* s *to state* s'*, representing the elapse of time, needed to reach the clock region* R' *from* R*.*

Definition 2.2 (A state clock region) *Let* RG *a region graph as defined below. Let* s *a state of* RG*. There exists only one clock region, denoted* $CReg(s)$*, associated with this state.*

In order to reduce the state explosion problem, some works have focused on the reduction and the minimization of the region graph model [YL93, ACH$^+$92, SV96]. The aim of [YL93, ACH$^+$92] is to generate the portion of the minimized region graph that is reachable in polynomial time. That is, all clock regions, in which the same actions can be executed, are gathered into one clock region, i.e., the state number is strongly reduced, what decreases validation costs.

3 Test cases derivation

In the classical conformance testing techniques (in the protocol engineering area), there are two main techniques :

- the automatic test case generation methods which are able to derive all the possible test cases of the whole system (the Unique Input/Output technique [SD88], the W technique [Cho78], ...). Here, systems are in general described by the Input/Output Finite State Machine model (IOFSM).
- the test purpose based methods which derive test cases from the test purpose given by the designer. But here, systems are usually specified by the Labeled Transition System model (LTS) [Bri87, Tre96].

In the present study, we suggest a testing technique based on timed test purposes. We will define the concept of Timed Test Purpose (TTP), which will express the user needs. And then, we will show how to extract test cases by using a timed synchronous product between the specification and the test purpose.

3.1 Timed test purpose

In classical (untimed) test purpose based methodologies, a test purpose is an abstract description of the specification. It helps the designer to choose behaviours to test and then to reduce the specification exploration. A test purpose is a graph where final states may be either accepting states (the purpose is reached) or refusing states (behaviour parts which would be rejected). It must contain events which should be found on the specification.

For timed systems, test purposes must also contain timing properties. We define such a purpose as a Timed Test Purpose (TTP).

Definition 3.1 (A Timed Test Purpose) *A timed test purpose* TTP *is defined as a tree. Each path from the root expresses a partial trace on the specification. The nodes do not need to have the same labels than the specification ones.*

An example of a timed test purpose required on the system described in Figure 1 is given in Figure 2. We should notice that timed test purposes are not parts of the specification, but are sequences containing behaviour and timing properties of the specification.

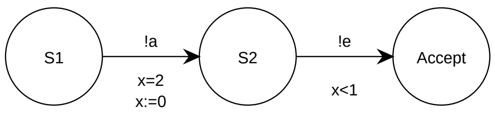


Figure 2: A Timed Test Purpose

Among many works dealing with automatic test purpose generation we can cite [CCKS95, CGPT95, PLC98]. No one deals with the region graph model, and most of them do not consider the time aspect.

3.2 Testing hypothesis

The following assumptions about the specification and the IUT must be satisfied for our methodology.

- Since we will use the region graph model, we should deal carefully with the clock regions. Then the clock regions to use in test purpose will be built with the same clocks than the specification ones. But the test purpose clock regions have not to be as the specification ones. So, the specification and the IUT must have the same number of clocks.
- The specification must be timed deterministic on the set of alphabet. That is, from any state, we cannot have two outgoing transitions labeled with the same symbol, whose timing constraints are satisfied simultaneously.
- From any state, we cannot have an outgoing transition, labeled with an input action and an outgoing transition, labeled with an output action, whose timing constraints are satisfied simultaneously.

3.3 Test case generation

Test cases, containing a timed test purpose and satisfying the specification will be generated in order to be applied to the IUT.

Test cases are generated from a kind of intersection of the timed test purpose TP and the specification S, defined as a timed synchronous product between TP and S. This operation generates a graph including TP and respecting the timing and the behaviour properties of S.

The different steps of this methodology are presented more precisely below. The specification S and the test purpose TP are modelled by the timed automata model:

- The whole specification is not necessary, since the designer checks only a part of it. So, transition sequences of S, containing all the actions of the test purpose, in the same order, are first extracted and named $TS_1(S), ..., TS_n(S)$. If this set is empty, the process terminates and the following steps cannot be performed.
- All the sequences $TS_1(S), ..., TS_n(S)$ and TP are translated into region graphs named respectively $RG(TS_1(S)), ..., RG(TS_n(S))$ and $RG(TP)$.
- Each region graph is then minimized, using the algorithm described in [ACH+92], and named $RGMin(RG(TS_1(S))), ..., RGMin(RG(TS_n(S)))$ and $RGMin(RG(TP))$.
- Each minimized region graph is synchronized with the timed test purpose. The definition of the timed synchronous product is given in section 3.4.

- From each synchronous product we extract all the possible paths which are transformed in timed test cases.

From the timed automaton illustrated in figure 1, we obtain, before the synchronous product, a sequence of transitions which generates the clock regions on Figure 3 and the corresponding region graph is given on Figure 4.

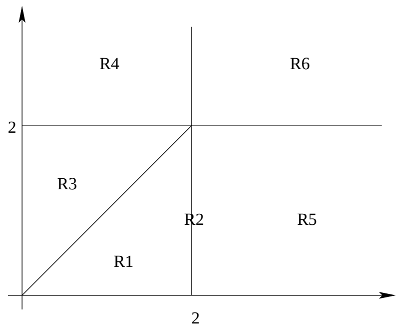


Figure 3: Clock regions obtained from the specification

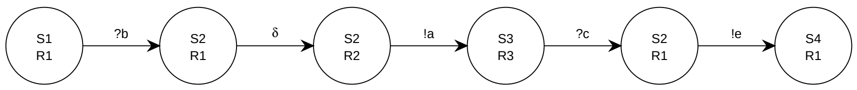


Figure 4: The specification region graph

In the same way, we obtain from the test purpose, illustrated in Figure 2, the clock regions of Figure 5 and the region graph of Figure 6.

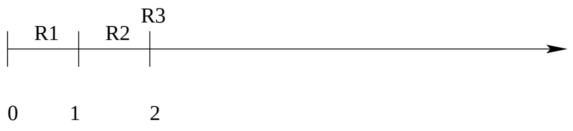


Figure 5: Clock regions obtained from the test purpose

3.4 Timed Synchronous Product

The timed synchronous product between two transitions $s_1 \xrightarrow{A} s_2$ (of a part of the specification S) and $s'_1 \xrightarrow{A} s'_2$ (of the timed test purpose TP), labeled with the same symbol A, can generate different synchronized clock regions, depending on $CReg(s_1)$ and on $CReg(s'_1)$.

The different types of synchronized clock regions are:

- **PASS region:**
  The clock region R_{pass} gathers clock valuations satisfying the execution of the specification action A, and the test purpose constraints, that is, the clock valuations which belong to R_S and R_{TP}. And if an action is executed at

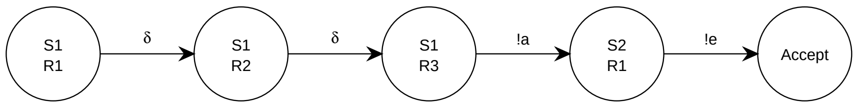


Figure 6: Region graph obtained from the test purpose

clock valuations of the PASS region, then the implementation conforms either to the specification or to the test purpose. If the PASS region is empty, an INCOHERENT one will be obtained.

- **INCONCLUSIVE region:**
  The clock region $R_{inconclusive}$ represents clock valuations which satisfy the execution of A in the specification, but not the execution of A in the test purpose. If, A is executed in this region, the implementation respects the specification, but its behaviour does not match with the test purpose. $R_{inconclusive}$ contains clock valuations of R_S except clock valuations of R_{pass}, and is denoted R_S/R_{pass}.
- **FAIL region:**
  This region represents all of the clock valuations which do not satisfy the execution of A in the specification. In this case, if A is executed, in this region, the implementation is faulty.
- **INCOHERENT region:**
  This is a sub-region of the FAIL region. It represents the clock valuations satisfying the execution of A in the test purpose, but not in the specification. This region is INCOHERENT since the IUT is still faulty while it satisfies the execution of A in the test purpose. This happens if we synchronize two clock regions in the situation that no clock valuation belongs to both clock regions simultaneously. This region is represented by R_{TP}/R_S.

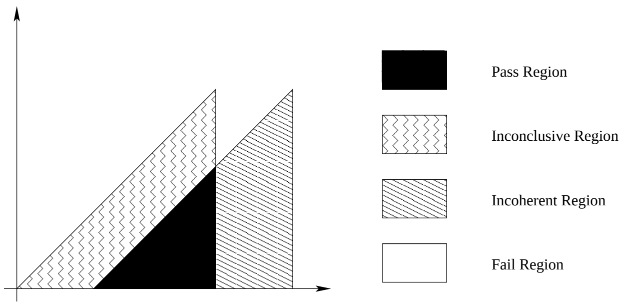


Figure 7: The regions obtained from a timed synchronous product

Figure 7 shows an example of regions which can be generated from a timed synchronous product of two clock regions with two clocks. The Timed synchronous product between a part of the specification S and a test purpose TP, is inspired by the definition of the product of two ETIOSM (Extended Timed Input Output State Machine which are Timed automata with non-temporal variables) [Lau99, PLC98].

In the following, we will consider two region graphs, a part of the specification (named S) and the test purpose (named TP). The resulting product is called SP. We will show in the following how to derive all the paths of this synchronous product.

Algorithm

```
INPUT:
TP(test purpose), S(specification part), PTP (set of TP paths), PS(set of S paths)
OUPUT:
SP(set of paths of the synchronous product)
BEGIN:
countSP := 0
FOR countTP = 0 TO LENGTH(PTP)
    /* computation of all TP paths */
    currentTP := PTP[countTP]
    Spaths:= search (currentTP, S)
1.  FOR countpath=0 TO LENGTH(Spaths)
        path:=Spaths[countpath]
        countTR:=0
2.      FOR countTR=0 TO LENGTH(path)
            tp[countTR].Label := path[countTR].Label
            tp[countTR].pass := CReg(path[countTR].StartingState)
            tp[countTR].inconclusive := $\emptyset$
        ENDFOR
        lengthTP:=countTR
        countTR:=0
3.      FOR count=0 TO LENGTH(currentTP)
            label:=currentTP[count].Label
4.          WHILE (label $different$ tp[countTR].Label) countTR := countTR + 1
            tp[countTR].inconclusive := tp[countTR].pass / CReg(currentTP[count].StartingState)
            tp[countTR].pass := tp[countTR].pass $\cap$ CReg(currentTP[count].StartingState)
        ENDFOR
        SP[countSP].length := lengthTP
        SP[countSP].tp := tp
        countSP := countSP + 1
    ENDFOR
ENDFOR
```

This algorithm can be commented as follows:

1. The *tp* path is a partial path of the specification, then we extract in this loop all the specification paths containing a *tp* path;
2. The *tp* is first set with the transition from the specification path. Each transition is labeled by a tuple (*label*,*pass*,*inconclusive*). *label* is deduced from the transition *ts* of the specification. *pass* is the region of the starting state of *ts*. *inconclusive* is set to the empty set.
3. This loop looks for the common transitions between *tp* and the specification. Here the pass region is set to the intersection between the specification region and the test purpose region. The inconclusive region is set with the specification region less the test purpose one.
4. This loop provides to skip the transitions which are found on the test purpose since the test purpose is a partial path of the specification path.

Finally, test cases are all the paths of the synchronous product.

We should mention that, the synchronous product may also generate a test case where the regions R_1 and R_2 of two successive states s_1 and s_2 are not time successor. In this case, it could be difficult to test any action in R_2: this clock region is not always reachable by the system clocks. This aspect is illustrated by the synchronous product of the figure 9 between the two timed automata of the figures 4 and 6.

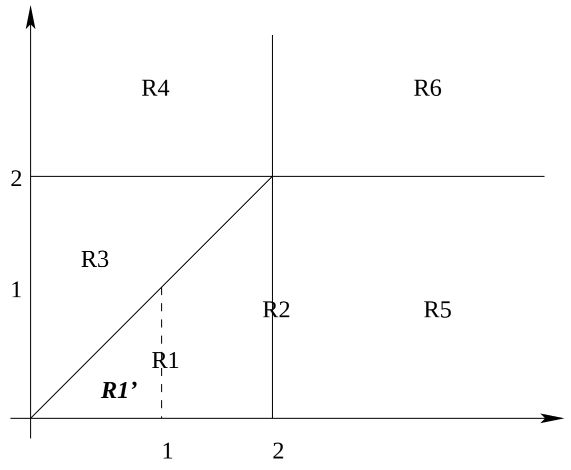


Figure 8: The obtained clock regions from the synchronous product

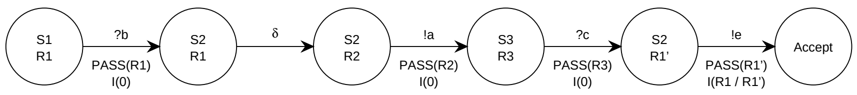


Figure 9: The obtained test case

4 Test Execution

4.1 Test Architecture

In order to apply the test cases on the implementation, a specific architecture for timed systems is described in [PF99b].

In this architecture, no assumption is done about how time is modeled inside the IUT (Implementation Under Test). The tester is composed of two parts, communicating with one another : the clock part, which contains the clocks appearing in the specification, and the behaviour part, whose role is to communicate with the implementation through the single PCO (Point of Control and Observation), i.e. to send inputs to the IUT and receive outputs from it.

The behaviour part of the tester may ask the clock part at any time for the value of one or more clocks and receive instantaneously the answer, this communication is internal to the tester.

On the other hand, the actions to be performed on the clocks, i.e the resets, do not involve the implementation anymore. The tester performs the resets independently from the IUT. During the testing process, temporal and behaviour properties of the implementation are checked, by means of test cases. These ones contain actions which must be checked in the PASS region. Unfortunately, clock regions are dense representation of time, which contain infinite tuples of values. Therefore, we cover the time space by the application of the following three rules:

- Let "?A" be an input action, a PASS region R, and let $v_{init} \in R$ be the first clock valuation reached by the clocks. We propose that the tester sends "A" to the IUT *as soon as possible and the latest possible*, that is for v_{init} and the last clock valuation reached by the clocks $v_{final} \in R$. In order to reach v_{final}, an elapse of time (denoted *wait*) is necessary and it must be computed on the fly during the test, on account of the dynamic behavior of the IUT.
- Let "!A" be an output action to test, R a PASS region and R' an INCONCLUSIVE region. The tester waits the reception of the symbol "A" from the IUT. If it is done in the region R, then the IUT respects the specification and the test purpose. But if it is done only in the INCONCLUSIVE region, the specification is respected but not the test purpose.
- For an elapse of time "δ", allowing to reach the next clock region, no test is needed. However, the tester must compute on the fly the duration of δ and must wait as long as this duration (section 4.2.1 for its calculation)

Consequently, we consider that input actions are checked for two clock valuations within the PASS region. So, a test case must be applied to the IUT 2^n times with n the number of input actions. To express all of these cases, we develop the execution of the test case into a tree, called *Execution Tree*.

4.2 Execution Tree

We suppose that we have a test sequence obtained from the synchronous product of two region graphs. The following algorithm transforms such a sequence into an *Execution Tree*. This tree is composed of states and the edges of the graph region:

- `send`(A) represents the sending of the input symbol ?A by the tester,
- `recv`(A) represents the sending of the output symbol !A by the IUT,
- `wait` represents the time needed to reach v_{final} by the IUT clocks in a clock region,
- δ represents the elapse of time needed to reach the next clock region.

Algorithm

```
Const-tree(SEQ,$s \xrightarrow[Rt]{A,PASS(R),INCONCLUSIVE(R')} s'$)

IF A=$\delta$ /* Elapse of time $\delta$
THEN { Add $\xrightarrow{\delta} s'$
       Cons-tree(SEQ,next transition of SEQ)

IF A=?I /* Input action
    IF two clock valuations can be reached by the tuple of clocks
    THEN { Add a branch $\xrightarrow{\delta}\xrightarrow{wait}\xrightarrow[Rt]{send(I)} s'$
           Cons-tree(SEQ,next transition of SEQ)
           Add a second branch $\xrightarrow{\delta}\xrightarrow[Rt]{send(I)} s'$
           Cons-tree(SEQ,next transition of SEQ)
    ELSE { Add $\xrightarrow{\delta}\xrightarrow[Rt]{send(I)} s'$
           Cons-tree(SEQ,next transition of SEQ)

IF A=!O /* Output action
THEN { Add a branch $\xrightarrow[Rt]{recv(O),PASS(R),INCONCLUSIVE(R')} s'$
       Cons-tree(SEQ,next transition of SEQ)
```

Finally, each branch of this tree can be given to the tester. The elapse of time *wait* and δ are computed as follows:

4.2.1 The calculation of δ and *wait*

Consider the sequence $s_1 \xrightarrow[Rt]{A,R_1} s_2 \xrightarrow{\delta} s_3 \xrightarrow[Rt]{B,R_2} s_4$. We need to calculate the elapse of time needed to reach R_2 from any clock valuation $v_{init} = (v_1, ..., v_n)$, reached after executing the action "A", that is the minimal value d such as $v_{init} + d \in R_2$.

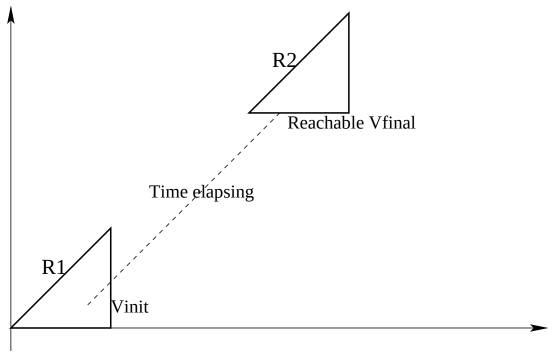


Figure 10: Reaching another clock region in v_{final} from v_{init}

As, the clocks grow with the same manner and strictly monotonically, they take as values the solutions of the equation $\begin{cases} x_1 - x_2 = v_1 - v_2 \\ ... \\ x_{n-1} - x_n = v_{n-1} - v_n \end{cases}$

So, v_{final}, reached by the clocks in R_2, is unique and is obtained by resolving the system of inequations

$$\Delta = \begin{cases} \text{Inequations of } R_2 \\ x_1 - x_2 = v_1 - v_2 \\ ... \\ x_{n-1} - x_n = v_{n-1} - v_n \end{cases}$$

Finally, the elapse of time δ equals to the difference between v_{init} and the minimal solution v_{final} of Δ, minus the time value ϵ, necessary for the resolution of the previous system of inequations. The example given in Figure 10 illustrates this aspect.

In the same manner, we can compute the elapse of time needed to reach a clock valuation of a clock region, denoted *wait*.

5 Complexity

In this section, we will analyze the complexity of each step of our technique. These calculations are difficult to express with accuracy since they depend on the number of states, actions and clocks of the the timed system.

- **Extraction of specification parts:** The complexity to obtain sequences of the specification, including the test purpose is proportional to $M * K$ (M: the number of states of the test purpose, K: the number of transitions of the test purpose). In the worst case, it is necessary to build a tree on S which contains all the transitions of S.
- **Region graph generation:** The complexity to build region graphs depends mainly on the number of states, the number transitions and the number of clocks of the timed automata. Each sequence SEQ, obtained with the previous step, have at most M states, K transitions and $C = card(C_S)$ clocks. So, according to [AD94], for each sequence, the complexity to build a region graph is proportional to $(M + K) * 2^{\delta(SEQ)}$ and where the number of clock regions is proportional to $(2^{\delta(SEQ)})$ and where $\delta(SEQ)$ is the number of clock constraints of SEQ. In the worst case, we will have K sequences, so the complexity to generate all of the region graphs is proportional to $(M + K) * K * 2^{\delta(SEQ)}$.
- **Synchronous product:** The complexity to synchronize two region graphs depends on the maximal number of transitions in both of them and on the number of clocks used during the resolution of the system of inequations to produce the PASS region. The number of transitions is still at most $K*2^{\delta(SEQ)}$, so the complexity of the synchronous product is proportional to $K*2^{\delta(SEQ)}*C$. At most, we have K region graphs, thus the complexity to synchronize theses ones with the test purpose is proportional to $K^2 * 2^{\delta(SEQ)} * C$.
- **Execution tree:** The complexity to build execution trees depends on the length and the number of test cases. Timed automata have at most $K*2^{\delta(SEQ)}$ transitions, the length of one test case, obtained from it, equals to $K * 2^{\delta(SEQ)}$. If test cases contains only input actions, for each one, two branches are added at the tree, thus the complexity to build all of the execution trees is proportional to $2^{K*2^{\delta(SEQ)}} * N$, with N the number of test cases.

Consequently, the global complexity of the methodology depends mainly on the number of clock regions generated in the second step. The number is exponential, the main complexity is exponential. However, the minimization of region graphs strongly reduces this number, but the number ofminimized clock regions is difficult to know before the minimization step.

6 Conclusion

We presented a practical technique to testing timed system based on timed purposes. It insures to check user properties (behavior and temporal aspects) on an implementation. These properties are specified as a tree of partial traces to find on the specification. We performed a synchronous product between the specification and the test purpose. This product selects pertinent clock regions where the testing process is possible. This last step allows to reduce the time space where the test has to be executed. Then, we extracted timed test cases from the synchronous product. Finally, we have shown how to execute these test cases on the implementation.

The originality of this study is to deal with the entire region graph deduced from the timed automaton describing any timed system. We considered the complete cycle from the specification to the execution of test cases.

Recently, we had developed a prototype tool accepting a timed automaton as a system specification and a test purpose as well. It executes the synchronous product and extracts the timed test cases. But this tool does not deal with more than two clocks.

Our main limitation is the use of only two clocks for the specification of timing constraints. We have not present any conformance relation between the specification and the implementation since we do not perform an total fault coverage of the implementation. Our purpose is only to detect errors by applying selected test purposes of the use.

References

[ACH+92] R. Alur, C. Courcoubetis, N. Halbwachs, D. Dill, and H. Wong-Toi. Minimization of timed transition systems. In R. Cleaveland, editor, *Proceedings CONCUR 92,* Stony Brook, NY, USA, volume 630 of *Lecture Notes in Computer Science*, pages 340–354. Springer-Verlag, 1992.

[ACH94] R. Alur, C. Courcoubetis, and T.A. Henzinger. The observational power of clocks. In B. Jonsson and J. Parrow, editors, *Proceedings CONCUR 94,* Uppsala, Sweden, volume 836 of *Lecture Notes in Computer Science*, pages 162–177. Springer-Verlag, 1994.

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[BFG+00] M. Bozga, J.C. Fernandez, L. Ghirvu, C. Jard, T. Jéron, A. Kerbrat, P. Morel, and L. Mounier. Verification and test generation for the sscop protoocl. *Science of Computer Programming*, 36(1):27–52, 2000.

[BLL+98] J. Bengtsson, K.G. Larsen, F. Larsen, P. Petterson, W. Yi, and C. Weise. New Generation of UPPAAL. In *Proceedings of the International Workshop on SOftware Tools and Technology Transfer (Aalborg, Denmark, July 12-13)*, July 1998.

[Bri87] E. Brinksma. A theory for the derivation of tests. In S. Aggarwal, editor, *Proceedings of the Eighth International Conference on Protocol Specification, Testing and Verification.* North-Holland, 1987.

[CCKS95] R. Castanet, C. Chevrier, O. Koné, and B. Le Saec. An Adaptive Test Sequence Generation Method for the User Needs. In *Proceedings of IWPTS'95, Evry, France*, 1995.

[CGPT95] M. Clatin, R. Groz, M. Phalippou, and R. Thummel. Two approaches linking a test generation tool with verification techniques. In *Proceedings of IWPTS'95, Evry, France*, 1995.

[CH96] J.Y. Choi and B.K. Hong. Generation of Conformance Test Suites for b-isdn Signaling Relevant to Multi_party Testing Architecture. In B. Baumgarten, H.-J. Burkhardt, and A. Giessler, editors, *Proceedings of the 8th International Workshop on Test of Communicating Systems IWTCS'96 (Darmstadt, Germany)*, pages 316–330, Amsterdam, september 1996. North-Holland.

[Cho78] T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, 1978.

[CKM92] Ana Cavalli, Sung Un Kim, and Patrick Maigron. Automated Protocol Conformance Test Generation Based on Formal Methods for LOTOS Specifications. In North Holland, editor, *Proceedings of the 5th International Workshop on Protocol Test System IWPTS'92 (Montreal, Canada)*, October 1992.

[CL97] D. Clarke and I. Lee. Automatic generation of tests for timing constraints from requirements. In *Proceedings of the Third International Workshop on Object-Oriented Real-Time Dependable Systems,* Newport Beach, California, February 1997.

[COG98] R. Cardel-Oliver and T. Glover. A practical and complete algorithm for testing real-time systems. In *Proc. of the 5th. Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1486 of Lecture Notes in Computer Science, pages 251–261. SpringerVerlag, 1998.

[DNH84] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[DOTY95] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool Kronos. In R. Alur, T.A. Henzinger, and E.D. Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages –. Springer-Verlag, 1995.

[DOY94] C. Daws, A. Olivero, and S. Yovine. Verifying ET-LOTOS programs with KRONOS. In D. Hogrefe and S. Leue, editors, *Proceedings of the 7th International Conference on Formal Description Techniques, FORTE'94*, pages 207–222. North-Holland, 1994.

[DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proceedings of the 1995 IEEE Real-Time Systems Symposium, RTSS'95,* Pisa, Italy. IEEE Computer Society Press, 1995.

[DY96] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. In *Proceedings of the 1996 IEEE Real-Time Systems Symposium, RTSS'96,* Washington DC, USA. IEEE Computer Society Press, 1996.

[ENDKE98] A. En-Nouaary, R. Dssouli, F. Khendek, and A. Elqortobi. Timed test cases generation based on state characterization technique. In *19th IEEE Real Time Systems Symposium (RTSS'98)* Madrid, Spain, 1998.

[FBK$^+$91] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite-state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, June 1991.

[Kon95] O. Koné. Designing test for time dependant systems. In *Proceedings of the* 13th *IFIP International Conference on Computer Communication* Séoul, South Korea, 1995.

[Lau99] P. Laurençot. *Integration du temps dans les tests de protocoles de communication.* PhD thesis, Univ. of Bordeaux 1, January 1999.

[LC97] Patrice Laurencot and Richard Castanet. Integration of Time in Canonical Testers for Real-Time Systems. In *International Workshop on Object-Oriented Real-Time Dependable Systems, California.* IEEE Computer Society Press, 1997.

[NS01] Brian Nielsen and Arne Skou. Automated Test Generation from Timed Automata. In T. Margaria and W. Yi, editors, *Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems,* Genova, Italy, volume 2031 of *Lecture Notes in Computer Science*, pages 343–357. Springer-Verlag, April 2001.

[PF99a] Eric Petitjean and Hacène Fouchal. From Timed Automata to Testable Untimeed Automata. In *24th IFAC/IFIP International Workshop on Real-Time Programming, Schloss Dagstuhl, Germany*, 1999.

[PF99b] Eric Petitjean and Hacène Fouchal. A Realistic Architecture for Timed Systems. In *5th IEEE International Conference on Engineering of Complex Computer Systems, Las Vegas, USA*, pages 109–118, 1999.

[PLC98] O. Koné P. Laurencot and R. Castanet. On the Fly Test Generation for Real Time Protocols. In *International Conference on Comnputer Communications and Networks, Louisiane U.S.A*, 1998.

[RNHW98] P. Raymond, X. Nicollin, N. Halbwatchs, and D. Waber. Automatic testing of reactive systems, madrid, spain. In *Proceedings of the 1998 IEEE Real-Time Systems Symposium, RTSS'98*, pages 200–209. IEEE Computer Society Press, December 1998.

[SD88] K. Sabnani and A. Dahbura. A protocol test generation procedure. *Computer Networks and ISDN Systems*, 15:285–297, 1988.

[SV96] J. Springintveld and F. Vaandrager. Minimizable timed automata. In B. Jonsson and J. Parrow, editors, *Proceedings of the 4th International School and Symposium on Formal Techniques in Real Time and Fault Tolerant Systems,* Uppsala, Sweden, volume 1135 of *Lecture Notes in Computer Science*, pages –. Springer-Verlag, 1996.

[SVD01] J. Springintveld, F.W. Vaandrager, and P. R. D'Argenio. Timed Testing Automata. *Theoretical Computer Science*, 254(254):225–257, 2001.

[Tre96] J. Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29:49–79, 1996.

[YL93] M. Yannakakis and D. Lee. An efficient algorithm for minimizing real-time transition systems (Extended abstract). In C. Courcoubetis, editor, *Proceedings of the 5th International Conference on Computer Aided Verification,* Elounda, Greece, volume 697 of *Lecture Notes in Computer Science*, pages 210–224. Springer-Verlag, 1993.

Test case characterisation by regular path expressions

Ralf Lämmel[1,2] and Jörg Harm[3]

[1] CWI, P.O. Box 94079, NL-1090 GB Amsterdam
[2] Vrije Universiteit, De Boelelaan 1081a, NL-1081 HV Amsterdam
[3] T-Systems, debis Systemhaus GEI, Lademannbogen 21-23, D-22339 Hamburg
Ralf.Laemmel@cwi.nl and Joerg.Harm@t-systems.de

Abstract. A signature-parametric (say generic) framework for test case characterisation, and definition of test set coverage criteria is developed. The signature might correspond to a programming language syntax, the format of a data structure, a computational or semantical structure, e.g., for derivations, proof trees, or control-flow graphs. Test set characterisation is based on regular expressions describing paths for terms over the signature at hand. Necessary and convenient properties for test set coverage criteria can be conceived in the framework. The framework is simple in the sense that it is solely based on term algebras and basic regular language theory. The specifications in the framework can be effectively used for coverage analysis, and for test case and test set generation.

1 Introduction

Generic automated testing We are interested in specification languages and tooling involved in automated testing of software and specifications. We relate test case characterisations and coverage criteria either directly to the programs and specifications themselves, or to the corresponding computational, implementational or semantical models. The fundamental contribution of the present article is that we setup a formal testing framework which is *generic* w.r.t. the actual syntax, format or model. All what we assume is the following:

- One can extract a signature Σ from the concrete testing scenario.
- The signature is useful to characterise test cases by regular expressions for the scenario.
- Terms over Σ are related to test data for the scenario in an effective manner.

These preconditions are met by many testing scenarios where in some sense a grammar, or a graph- or tree-like structure is a primary artifact, namely in compiler testing (cf. [Bur94]), and traditional program testing (cf. [Mye79,Bei90]). Such a link is not obvious for testing centered around observable behaviour, e.g., in testing transition system [BT00]. In the generic framework, one can characterise test cases and define coverage criteria for test sets, and one can perform coverage analysis for test sets, and generation of test cases and test sets. The framework is based on regular language theory to deal with paths for terms over Σ. There are coverage criteria which were generically approved to meet valuable properties like completeness. The framework enforces an effective style of specifications for test cases and test sets, that is, there exist simple algorithms to perform coverage analysis, test case generation, and test set generation.

Testing a program Let us discuss a few scenarios where we want to generate test data from a program P to test this very program, or we want to assess a given test set for P. As an aside, in both cases, we are concerned with white-box testing. Suppose P is a Prolog program. Several scenarios with different extracted signatures Σ are conceivable, e.g.:

- Let us choose the scenario that we want to generate test data (say roots of proof trees of the goal clause of P) which exercise the clauses of P in a certain way, namely all clauses are applied in all possible contexts. We choose the skeleton of P (i.e., the clauses with the parameters stripped away), as Σ. The generic framework immediately provides a suitable

coverage criterion, namely context-dependent branch coverage. We perform test set generation in the generic framework accordingly. Thereby, we obtain a set of terms over Σ. In this scenario, a term over Σ corresponds roughly to a proof tree skeleton (where the parameters are not constrained). Mapping back such a term to the Prolog context means that we need to execute the program P in a way to enforce a proof tree which matches the given proof tree skeleton. This can be done by controlling the program execution via a meta-program interpreter (cf. [Den91]). In worst case, we cannot complete the given proof tree skeleton into a proof tree.
– Alternative choices for Σ are the structure of proof trees for P, the structure of SLD derivations for P, the functors used for data representations in P, the interface for library functionality used in P, or some signature of debugging traces for P. All these choices are useful for some scenario of testing. We need to employ, for example, the structure of proof trees for P (instead of the skeleton of P above) if we want to constrain the parameters of predicates in some way or another, e.g., if we want to enforce that a certain depth of nesting is exercised in some parameter position of P.

As an aside, extracting Σ from a testing scenario is usually simple. Mapping back terms to the testing scenario often involves computational, semantical and implementational models as in the above scenario where proof tree skeletons had to be completed to proof trees. In the present article, we focus on the generic part, that is, the characterisation of test cases and coverage criteria w.r.t. a signature Σ.

Testing a program against a specification If we, for example, talk about a programming language L, the signature Σ might correspond to the concrete or abstract syntax of L, to a library signature, to an intermediate format, or to a derived format (e.g., for control-flow and data-flow analysis). Let us consider the simple case that we want to test the frontend implementation I (say parser) of the language L, and that a grammar specification G is available for L. The signature Σ is then simply extracted from G. By this, we can pursue test set generation or coverage analysis in the generic framework. If we indeed generate test cases, we can easily map them back to the testing scenario, that is, terms over the extracted Σ have to be rephrased as words in the language L. Subsequently, we can feed the obtained strings to the frontend to perform black-box testing of the frontend implementation I against the grammar specification G. As for the purpose of testing, we might want to test for completeness of I by exercising all language constructs admitted by G, or we deal with stress testing. In yet another case, a test suite for the implementation I might be available, and we want to assess it to decide if the suite achieves coverage of the specification G according to a criterion adopted from the generic framework.

Research context In white-box testing for programs of a certain language L, one can use different structural properties (cf. [Mye79,Bei90]), e.g., folklore properties such as statement or path coverage. One can also try to use more abstract aspects like the data flow in a program (cf. [RW85,FW88]). Even if the derivation of a data-flow (or a control-flow) graph involves the semantics of L, the graph is finally again a structure which admits testing concepts similar to those for abstract syntax trees. Hence, there are several candidates for the signature Σ, namely the abstract or concrete syntax of L, the structure of control-flow graphs or data-flow graphs. Our generic framework is the result of an attempt to abstract from the actual structure of a testing scenario, and to study testing notions once and for all in a largely generic manner. On the other hand, we were driven by the intriguing question if there is some unified style for test case characterisation and for coverage criteria for different kinds of specifications like syntax definitions, attribute grammars, algebraic specifications, and logic programs. Such a unified style could be helpful to compare coverage criteria, and to realise the potential for additional criteria. So far, coverage criteria were defined in a rather specific context. The seminal paper by Purdom [Pur72b] defines a coverage criterion for rule coverage of a context-free grammar, and algorithms for test set generation are given. In logic programming, some coverage criteria have been studied (cf. [Den91,Jac96]) which seem to have a useful interpretation for other forms of specifications and programs, too. In the context of language implementation, the problem of generating test sets from various kind of grammars has been studied extensively (cf. [CRV$^+$80,BSd82,HS89,Rie92]). Most approaches suggested in the literature are not as formal as Purdom's approach, that is, they are hardly based on a formal coverage criterion. Instead, heuristics, mutation, and randomization are employed. Grammars are often instrumented in a pragmatic manner. In [CRV$^+$80], for example, a special grammar formalism (with actions working on the internal data structures of the test set generator) is used. A survey on test set generation for compiler testing is given in [Bur94]. Recently, we contributed approximation coverage for attribute grammars (cf. [HL00]), and context-dependent branch coverage for syntax definitions (cf. [Läm01]). In the generic framework, we favour regular path expressions on terms over a signature as the formal tool for characterising test cases and coverage criteria. In certain application domains related to testing, some kind of path expressions also have been used, e.g., for querying graph-models of a database and for querying semi-structured data (cf. [Abi97]), or for testing of VHDL programs based on control-flow paths (cf. [VK95]). Our abstract and unified style indeed allows us to compare coverage criteria, and to enforce general properties like completeness. In [Wey89], informal requirements for coverage criteria are stated and reviewed. By contrast, the present article contributes a formal model for coverage criteria.

Structure of the article In Section 2, some notation for dealing with regular expressions and terms is spelled out for convenience. In Section 3, a formal framework for test case characterisation is developed. Test cases according to a signature Σ are essentially characterised by regular expressions over the paths for terms over Σ. In Section 4, test set characterisation as opposed to simple test case characterisation is accomplished. The corresponding sets of classes of paths are subject to a number of properties, e.g., minimality and completeness, ultimately leading to the notion of a proper coverage criterion. As for the effectiveness of the specifications, we rely on basic regular language theory. In Section 5, a number of coverage criteria are developed. We start from of an abstraction of rule coverage [Pur72b]. Then, a number of more involved criteria are defined. A more elaborated article (with proofs, more examples, and hints on applications) is under the way.

Acknowledgement The authors are indebted to Guido Wachsmuth for helping us with the implementation of the ideas formalised in the article. We used the FSA utilities [Noo97] by Gertjan van Noord on top of SWI-Prolog [Wie00] by Jan Wielemaker to implement the framework (with great pleasure). The work of the first author was supported, in part, by the Dutch research organisation *NWO*, in the project "*Generation of Program Transformation Systems*".

2 Preliminaries

Term algebras A signature Σ is pair $\langle S, F\rangle$, where S is a finite set of sorts, and F is a finite set of function types of the form $f : \sigma_1 \times \cdots \times \sigma_n \to \sigma_0$, $n \geq 0$. Here, f is a function symbol, and the σ_i are sorts in S. We denote the arity n of f as $arity(f)$. For convenience, we also sometimes write $f \in F$, i.e., without giving the type of f. We use $\mathcal{T}_\sigma(\Sigma)$ to denote the set of ground or variable-free terms of sort σ in the common term-algebraic sense. We assume terminated term algebras in the sense of context-free grammars, that is for all $\sigma \in S$ it holds that $\mathcal{T}_\sigma(\Sigma) \neq \emptyset$. We also assume that function symbols are not overloaded. These restrictions are easy to lift.

Regular expressions We assume familiarity with basic regular language theory (cf. [HU80,ASU86]). We use finite state automata (FSAs) to visualise regular languages. We use the common notation for regular expressions. Below, we provide an inductive definition of the domain $\mathcal{RE}_T$ of regular expressions over some set of terminals T.

- $t \in \mathcal{RE}_T$ for $t \in T$ (terminals)
- $e, e' \in \mathcal{RE}_T \Rightarrow e\,e' \in \mathcal{RE}_T$ (concatenation)
- $e, e' \in \mathcal{RE}_T \Rightarrow e \,|\, e' \in \mathcal{RE}_T$ (alternatives)
- $e \in \mathcal{RE}_T \Rightarrow \tilde{\ }e \in \mathcal{RE}_T$ (complement)
- $e \in \mathcal{RE}_T \Rightarrow e^* \in \mathcal{RE}_T$ (star-notation)
- $e \in \mathcal{RE}_T \Rightarrow e^+ \in \mathcal{RE}_T$ (plus-notation)

Furthermore, we use some less common notation for star- and plus-notation to constrain the number of iterations:

- $e \in \mathcal{RE}_T, \alpha, \omega \in \mathcal{N}_0, \alpha \leq \omega \Rightarrow e^{*^{\omega}_{\alpha}} \in \mathcal{RE}_T$
- $e \in \mathcal{RE}_T, \alpha, \omega \in \mathcal{N}_1, \alpha \leq \omega \Rightarrow e^{+^{\omega}_{\alpha}} \in \mathcal{RE}_T$

The intended meaning of $e^{*^{\omega}_{\alpha}}$ and $e^{+^{\omega}_{\alpha}}$ is that the number of iterations is at least α and at most ω. $\mathcal{N}_0$ and $\mathcal{N}_1$ are the naturals starting from 0 or 1, respectively. e^{*_i} and e^{+_i} are abbreviations for $e^{*^{i}_{i}}$ and $e^{+^{i}_{i}}$, respectively. The set of terminal strings generated by a regular expression e is denoted with $[\![e]\!]$. We also might be interested in terminal strings generated by a regular grammar starting from one of its nonterminals. The terminal strings generated from the nonterminal n according to G is denoted by $[\![n]\!]_G$. In general, we can also consider the terminal strings $[\![e]\!]_G$ generated by a regular expression which contains nonterminals. If G is obvious from the context, we omit G in $[\![e]\!]_G$. Finally, we declare a convenient abbreviation, that is Some. Its intended meaning is $[\![\mathsf{Some}]\!] = T^*$. Assuming $T = \{t_1, \ldots, t_n\}$, we can easily give a regular expression defining Some, namely $\mathsf{Some} = (t_1 \mid \cdots \mid t_n)^*$.

3 Test case characterisation

A signature Σ induces a class of paths for terms over Σ.[1] Obviously, a term can be regarded as a set of paths. Test cases can be characterised by regular expressions intended to model classes of paths. We have chosen a simple functional language as the running example. We want to characterise some programs in this language.

3.1 Paths

We define a regular grammar $\mathcal{PG}(\Sigma)$ which models the paths for the terms according to a given signature Σ. Technically, $\mathcal{PG}(\Sigma)$ is defined to generate selector sequences to descend into terms. These selector sequences are called paths in the sequel.

Definition 1. *$\mathcal{PG}(\Sigma)$ denotes the Σ-*paths grammar *with a finite set of nonterminals N, a finite set of terminals T, $N \cap T = \emptyset$, and a finite set of productions P. N and T are defined as follows:*

$$N = S$$
$$T = \{f \mid f \in F\} \cup \{1, \ldots, \mathit{max}\,(\{\mathit{arity}(f) \mid f \in F, \mathit{arity}(f) > 0\})\}$$

For all $f : \sigma_1 \times \cdots \times \sigma_n \to \sigma_0 \in F$ there are the following rules in P:

- $n = 0$: $\sigma_0 \to f$
- $n > 0$: $\sigma_0 \to f\,1\,\sigma_1$
  $\ldots$
  $\sigma_0 \to f\,n\,\sigma_n$

According to the definition, the function symbols from F are regarded as terminals in $\mathcal{PG}(\Sigma)$, and there are special terminals corresponding to parameter positions $1, 2, \ldots$ modelling selection of subterms. For each constant symbol in F there is a trivial rule in $\mathcal{PG}(\Sigma)$. For each function symbol $f \in F$ with $\mathit{arity}(f) > 0$, there is one rule per parameter position of f in $\mathcal{PG}(\Sigma)$. Paths are complete per definition, i.e., they are terminated with a constant symbol.

Example 1. Let us consider a syntax definition of a simple functional language, and the corresponding grammar of paths. There are three forms of expressions, namely variables (cf. **lvar**), λ-abstractions (cf. **lambda**), and function applications (cf. **apply**). Furthermore there are two

[1] We usually talk about classes of paths as opposed to sets of paths for the sake of a suggestive terminology. We will later have to consider sets of classes of paths. In fact, sets of sets of paths does not sound very appealing.

forms of type expressions, namely type-variables (cf. **tvar**) and arrow types (cf. **arrow**). We use naturals to represent λ-variables and type variables.

Σ_λ:

$$\begin{aligned}
\textbf{lvar} &: \mathit{Nat} \to \mathit{Exp}\\
\textbf{lambda} &: \mathit{Nat} \times \mathit{Type} \times \mathit{Exp} \to \mathit{Exp}\\
\textbf{apply} &: \mathit{Exp} \times \mathit{Exp} \to \mathit{Exp}\\
\textbf{tvar} &: \mathit{Nat} \to \mathit{Type}\\
\textbf{arrow} &: \mathit{Type} \times \mathit{Type} \to \mathit{Type}\\
\textbf{zero} &: \to \mathit{Nat}\\
\textbf{succ} &: \mathit{Nat} \to \mathit{Nat}
\end{aligned}$$

$\mathcal{PG}(\Sigma_\lambda)$:

$$\begin{aligned}
\mathit{Exp} &\to \textbf{lvar}\,1\,\mathit{Nat} & \mathit{Type} &\to \textbf{tvar}\,1\,\mathit{Nat}\\
\mathit{Exp} &\to \textbf{lambda}\,1\,\mathit{Nat} & \mathit{Type} &\to \textbf{arrow}\,1\,\mathit{Type}\\
\mathit{Exp} &\to \textbf{lambda}\,2\,\mathit{Type} & \mathit{Type} &\to \textbf{arrow}\,2\,\mathit{Type}\\
\mathit{Exp} &\to \textbf{lambda}\,3\,\mathit{Exp} & \mathit{Nat} &\to \textbf{zero}\\
\mathit{Exp} &\to \textbf{apply}\,1\,\mathit{Exp} & \mathit{Nat} &\to \textbf{succ}\,1\,\mathit{Nat}\\
\mathit{Exp} &\to \textbf{apply}\,2\,\mathit{Exp}
\end{aligned}$$

Paths of sort *Exp* are represented as an FSA in Figure 1. To this end, the above regular grammar is trivially transformed into an FSA. To be precise, the FSA accepts the regular language $[\![\mathit{Exp}]\!]$.

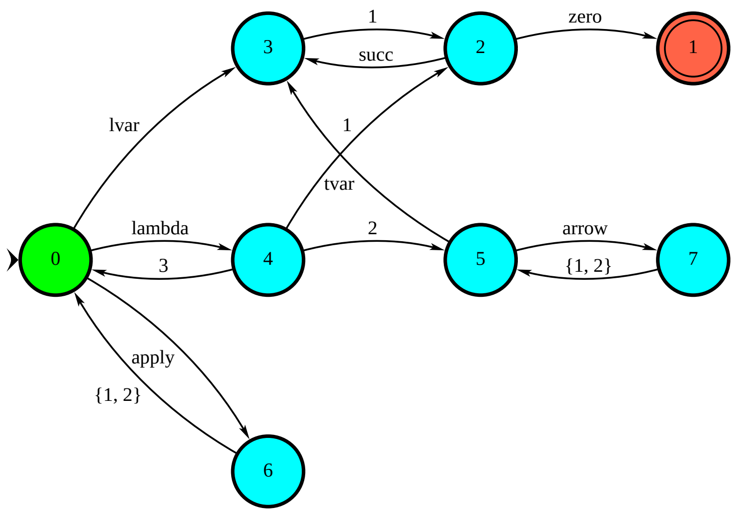


Fig. 1. The FSA accepting $[\![\mathit{Exp}]\!]$

A term can be associated with the set of all its paths in a natural way. Such a mapping is obviously useful to decide if a given term exercises a given set of paths (meant to characterise a test case). The decision would be simply based on a test for non-empty intersection.

Definition 2. *The* set $[\![t]\!] \subseteq [\![\sigma]\!]$ of paths of the term t of sort σ *is defined as follows:*

$$[\![t]\!] = \begin{cases} \{f\}, & \text{for } t = f\\ \bigcup_{i=1}^{n} \{f\,i\,x \mid x \in [\![t_i]\!]\}, & \text{for } t = f(t_1,\ldots,t_n), n \geq 1 \end{cases}$$

Example 2. Consider the following term of sort *Exp* over the signature from Example 1:

$$\textbf{apply}(\textbf{lvar}(\textbf{zero}), \textbf{apply}(\textbf{lvar}(\textbf{zero}), \textbf{lvar}(\textbf{succ}(\textbf{zero}))))$$

The set of paths for this term is the following:

$$\begin{aligned}
\{\ &\textbf{apply}\,1\,\textbf{lvar}\,1\,\textbf{zero},\\
&\textbf{apply}\,2\,\textbf{apply}\,1\,\textbf{lvar}\,1\,\textbf{zero},\\
&\textbf{apply}\,2\,\textbf{apply}\,2\,\textbf{lvar}\,1\,\textbf{succ}\,1\,\textbf{zero}\ \}
\end{aligned}$$

3.2 Regular path expressions

We use regular expressions over the terminals and nonterminals of $\mathcal{PG}(\Sigma)$ to specify classes of paths, hence we call them regular path expressions. Such sets characterise the kind of terms, that is, the test cases which we are interested in. Regular path expressions serve as convenient finite representations of potentially infinite sets of paths. Usually, we consider paths of a certain sort σ. A regular expression e, which one writes down in the first place, is often more productive, and hence, $[\![e]\!]$ has to be restricted to proper paths (of sort σ) by intersection with $[\![\sigma]\!]$. To express our intention that e should characterise paths of sort σ, we use declarations of the form $e : \sigma$. Since regular languages are closed under intersection, the restriction of e to proper paths of sort σ is solely performed at the level of regular languages.

Definition 3. *$c \subseteq T^*$ is a* set of paths of sort σ *if $c \subseteq [\![\sigma]\!]$. Given a sort-annotated regular path expression $e : \sigma$, we use $[\![e : \sigma]\!] = [\![e]\!] \cap [\![\sigma]\!]$ to denote the* set of paths of sort σ corresponding to *e. A term $t \in \mathcal{T}_\sigma(\Sigma)$* exercises the set c *of paths of sort σ if $c \cap [\![t]\!] \neq \emptyset$.*

Example 3. Consider again the language syntax introduced in Example 1. We want to characterise paths for nested function applications. The following regular expression provides a suitable characterisation for paths with precisely $i \geq 0$ nested function applications:

$$apply_i = (\tilde{\ }\mathbf{apply})^* \, (\mathbf{apply}\, (1|2) \, (\tilde{\ }\mathbf{apply})^*)^{*i}$$

Our intention is to consider expressions only, hence we annotate $apply_i$ with the sort Exp. The FSA accepting the language $[\![apply_2 : Exp]\!] = [\![apply_2]\!] \cap [\![Exp]\!]$ is shown in Figure 2. As an aside, the example instantiates a simple scenario which we often find in testing parsers or language processors, that is, stress testing. Using a larger i in the above specification, and applying test set generation to inhabit the corresponding set of paths, we can enforce complex terms for stress testing.

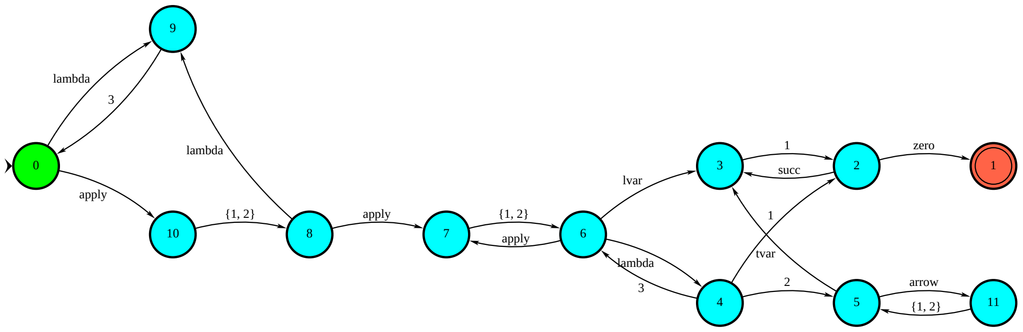


Fig. 2. The FSA for $[\![apply_2 : Exp]\!]$

Note that it does not really make a difference if we say that a term t exercises

- an annotated regular path expression $e : \sigma$, or a
- class $c \subseteq T^*$ of paths of sort σ

since the class of paths corresponding to $e : \sigma$ is trivially obtained by $[\![\cdot]\!]$. In the sequel, we will assume the convention that regular path expressions can be used whenever classes of paths are appropriate. This will simplify our presentation.

Example 4. The sample term given in Example 2 exercises the i-indexed expressions $apply_i : Exp$ for nested function applications characterised in Example 3 as follows:

Path	i in $apply_i : Exp$
apply 1 **lvar** 1 **zero**	1
apply 2 **apply** 1 **lvar** 1 **zero**	2
apply 2 **apply** 2 **lvar** 1 **succ** 1 **zero**	2

3.3 Feasibility and productivity

We are hardly interested in regular path expressions which do not admit any proper path. They are somewhat uselus since they cannot be exercised. By contrast, we mostly favour feasible regular path expressions, that is:

Definition 4. *The sort-annotated regular path expression* $e : \sigma$ *is* feasible *if* $[\![e : \sigma]\!] \neq \emptyset$.

An expression $e : \sigma$ can be infeasible for different reasons. A function symbol might be used with the wrong arity, or the path expression is incomplete. We might disclose such expressions by a static well-formedness check for expressions. This is not pursued in the article. Other reasons for infeasibility are more like an indication that paths of the specified kind do not exist due to reachability arguments. Feasibility is a fundamental requirement. There is a related problem. One might want to require that all subexpressions in a regular path expression e, especially all alternatives and all stars and pluses in e, contribute to $[\![e : \sigma]\!]$ (and not just to $[\![e]\!]$). We call that property *productivity*. The property could be enforced by a static productivity check, too.

Example 5. The following expressions deal with the signature in Example 1.

$$\begin{aligned}
e_1 &= \textbf{lambda}\,1\\
e_2 &= \textbf{lambda}\,1\,\textbf{lvar}\,\mathsf{Some}\\
e_3 &= \mathsf{Some}\,\textbf{lambda}\,\mathsf{Some}\\
e_4 &= \mathsf{Some}\,\textbf{lambda}\,\mathsf{Some} \mid \textbf{arrow}\,\mathsf{Some}
\end{aligned}$$

Let us examine these expressions with regard to feasibility and productivity. To this end, we select certain sorts to annotate the expressions. $e_1 : \mathit{Exp}$ is not feasible because e_1 is obviously incomplete, that is, it is not terminated with a constant symbol. $e_2 : \mathit{Exp}$ is not feasible for simple type arguments. The symbol **lvar** is of sort *Exp* as opposed to the first parameter position of **lambda** which is of sort *Nat*. $e_3 : \mathit{Type}$ is not feasibly for reachability reasons, that is the function symbol **lambda** is of sort *Exp* but expressions are not reachable on paths of sort *Type*. The first alternative of $e_4 : \mathit{Type}$ is not productive because it is equal to the infeasible e_3.

4 Sets of test cases

A single regular expression e is usually sufficient to provide an abstract description of a test case or a class of test cases. If we want to characterise a test set, we also need to resort to a set E of regular expressions. Each expression in such a set corresponds to a different class of test cases. For single expressions, we considered feasibility and productivity. For sets of expressions, several other properties are naturally defined, e.g., minimality and completeness. Ultimately, the section provides requirements for sets of regular path expressions to denote a proper coverage criterion.

4.1 Sets of regular path expressions

The following definition provides a kind of example for sets regular path expressions. It introduces two basic sets of expressions. The set CC_Σ models coverage of all constant symbols in Σ. The set FC_Σ models coverage of all other function symbols f in Σ (i.e., $\mathit{arity}(f) > 0$).

Definition 5.

$$\begin{aligned}
CC_\Sigma : \sigma &= \{\mathit{const}_c \mid c \in F, \mathit{arity}(c) = 0\} \ \textit{where}\ \mathit{const}_c = \mathsf{Some}\, c\\
FC_\Sigma : \sigma &= \{\mathit{func}_f \mid f \in F, \mathit{arity}(f) > 0\} \ \textit{where}\ \mathit{func}_f = \mathsf{Some}\, f\, \mathsf{Some}
\end{aligned}$$

Note that it is essential to consider sets of regular path expressions in order to effectively separate the classes of test cases. One might feel tempted to define these sets as a list of alternatives using the regular operator "|", e.g., for $FC_{\Sigma_\lambda} : \mathit{Exp}$:

$$\mathsf{Some}\,\textbf{lvar}\,\mathsf{Some} \mid \mathsf{Some}\,\textbf{lambda}\,\mathsf{Some} \mid \cdots \mid \mathsf{Some}\,\textbf{succ}\,\mathsf{Some}$$

In accordance to Definition 3, it would be sufficient to exercise one alternative to exercise the entire expression. This is in conflict with the desired meaning of $FC_{\Sigma_\lambda} : Exp$ to characterise a set of a different classes of test cases. To exercise a set of classes of paths, in general, we might also need a set of terms rather than just a single term. Let us generalise Definition 3 accordingly.

Definition 6. *$c \subseteq \mathcal{P}(T^*)$ is a* set of classes of paths of sort σ *if every $c \in \mathcal{C}$ is a set of paths of sort σ. Given a sort-annotated set of regular expression $E : \sigma$, we use $[\![E : \sigma]\!] = \{[\![e]\!] \cap [\![\sigma]\!] \mid e \in E\}$ to denote the* set of classes of paths of sort σ corresponding to E. *The set $\mathcal{S} \subseteq \mathcal{T}_\sigma(\Sigma)$ of terms exercises the set $\mathcal{C}$ of classes of paths of sort σ if for all classes $c \in \mathcal{C}$ there exists a term $t \in \mathcal{S}$ such that t exercises c.*

Consider the case that different sets of sort-annotated regular expressions should be compared. An interesting question is if one can define a pre-order on sets for the same sort σ. Such a pre-order could model if one set of expressions is more challenging than the other in the sense that all test sets for the former also exercise the latter. In fact, in Section 5 we will examine several concrete coverage criteria in this respect.

Definition 7. *Given two sets $\mathcal{C}$ and $\mathcal{C}'$ of classes of paths of sort σ, $\mathcal{C}'$ is called a refinement of $\mathcal{C}$ (notation $\mathcal{C} \sqsubseteq \mathcal{C}'$) if for all $\mathcal{S} \subseteq \mathcal{T}_\sigma(\Sigma)$ it holds that $\mathcal{S}$ exercises $\mathcal{C}'$ implies $\mathcal{S}$ exercises $\mathcal{C}$. $\mathcal{C}'$ and $\mathcal{C}$ are* equivalent *(notation $\mathcal{C} \equiv \mathcal{C}'$) if both $\mathcal{C} \sqsubseteq \mathcal{C}'$ and $\mathcal{C}' \sqsubseteq \mathcal{C}$.*

4.2 Properties

For single expressions (or classes of paths) we considered the properties of feasibility and productivity. Feasibility can be trivially lifted for sets. Regarding a set of expressions $E : \sigma$, it might be the case that some of the corresponding sets of classes coincide in $[\![E : \sigma]\!]$. We call E erasing in that case. One could also say that E is not productive in this case. Of course, E is also not productive if some expressions in E are not productive on their own.

Definition 8. *The set $\mathcal{C}$ of classes is* feasible *if $\emptyset \notin \mathcal{C}$. $E : \sigma$ is called* erasing *if $|E| > |[\![E : \sigma]\!]|$.*

Like for infeasibility, an erasing set of expressions E might indicate a specification error, or it might just be implied by the concrete signature at hand. Note also that if two or more classes are not feasible in $[\![E : \sigma]\!]$, E is necessarily erasing.

Example 6. Consider the following trivial signature Σ_{foo} with the function symbols $f : s_2 \to s_1$, $g : s_3 \to s_2$, $h : \to s_3$. One can check that $FC_{\Sigma_{foo}} : s_1$ is erasing because $[\![func_f : s_1]\!] = [\![func_g : s_1]\!]$, i.e., f and g always "go together". For a signature Σ_{bar}, which in addition to Σ_{foo} also contains $f' : s_2 \to s_1$ and $g' : s_3 \to s_2$, $FC_{\Sigma_{bar}} : s_1$ is not erasing.

Let us consider more interesting properties of sets of classes of paths, e.g., completeness (i.e., all proper paths are in some class), and minimality (i.e., no class of paths is completely subsumed by another).

Definition 9. *Let $\mathcal{C}$ be a set of classes of paths of sort σ w.r.t. a signature Σ. The* complement $\overline{\mathcal{C}}$ of $\mathcal{C}$ *is defined as $[\![\sigma]\!] \setminus \bigcup \mathcal{C}$. We say that the set $\mathcal{C}$ of classes is*
- complete *if $\overline{\mathcal{C}} = \emptyset$,*
- minimal *if $\nexists c, c' \in \mathcal{C}.\ c \subset c'$,*
- disjunctive *if $\nexists c, c' \in \mathcal{C}.\ c \neq c' \wedge c \cap c' \neq \emptyset$.*

There are some simple observations worth mentioning:

- If $\mathcal{C}$ is minimal, then $\mathcal{C}$ is feasible.
- If $\mathcal{C}$ is disjunctive and feasible, then it is minimal.
- If $\mathcal{C}$ is complete, then $\mathcal{C} \cup \mathcal{C}'$ is complete for all $\mathcal{C}'$.
- $\mathcal{C}$ is a partitioning of $[\![\sigma]\!]$, iff it is feasible, complete and disjunctive.

4.3 Recovery of feasibility, minimality, completeness

In turns out that the specifications one writes down in the first place, are initially hardly non-erasing, feasible and minimal, and sometimes they are not complete. We will encounter such situations in Section 5 when we define certain coverage criteria. Feasibility, minimality, and completeness can be recovered by the following operators:

Definition 10.
Let $\mathcal{C}$ be a set of classes of paths of some sort σ.
The sets $\lfloor\mathcal{C}\rfloor$, $\lceil\mathcal{C}\rceil$ are defined as follows:

$$\lfloor\mathcal{C}\rfloor = \left\{ c \setminus \bigcup \mathcal{C}' \;\middle|\; c \in \mathcal{C}, \mathcal{C}' = \{c' \in C \mid c' \subset c\} \right\} \setminus \{\emptyset\}$$

$$\lceil\mathcal{C}\rceil = \begin{cases} \mathcal{C}, & \text{if } \overline{\mathcal{C}} = \emptyset \\ \mathcal{C} \cup \{\overline{\mathcal{C}}\}, & \text{otherwise} \end{cases}$$

We might also apply $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ to sets of expressions without further notice. The definition of $\lceil\mathcal{C}\rceil$ for completing $\mathcal{C}$ is straightforward, that is, completion is done by including another set of paths in $\mathcal{C}$ corresponding to the complement of (the union of all sets in) $\mathcal{C}$. The definition of $\lfloor\mathcal{C}\rfloor$ to minimalise $\mathcal{C}$ is more involved. For every $c \in \mathcal{C}$ the subset $c \setminus \bigcup \mathcal{C}'$ is preserved as a class. Here, $\mathcal{C}'$ denotes all classes c' which are proper subsets of c. Thus, in a sense, we preserve greatest subsets of c not covered by $\mathcal{C}'$. The operators meet some convenient properties summarized in the following theorem. All these properties are easily proved.

Theorem 1. *For all sets $\mathcal{C}$, $\mathcal{C}'$ of classes of paths of some sort σ:*

1. *$\lfloor\mathcal{C}\rfloor$ is minimal.*
2. *The operator $\lfloor\cdot\rfloor$ is idempotent.*
3. *$\lceil\mathcal{C}\rceil$ is complete.*
4. *The operator $\lceil\cdot\rceil$ is idempotent.*
5. *$\lfloor\lceil\mathcal{C}\rceil\rfloor = \lceil\lfloor\mathcal{C}\rfloor\rceil$.*
6. *If $\mathcal{C}$ is minimal, then $\lceil\mathcal{C}\rceil$ is also minimal.*
7. *If $\mathcal{C}$ is complete, then $\lfloor\mathcal{C}\rfloor$ is also complete.*
8. *$\bigcup\mathcal{C} = \bigcup\lfloor\mathcal{C}\rfloor$.*
9. *$\mathcal{C} \sqsubseteq \lfloor\mathcal{C}\rfloor$.*
10. *$\mathcal{C} \sqsubseteq \lceil\mathcal{C}\rceil$.*
11. *$\lfloor\mathcal{C}\rfloor \sqsubseteq \lfloor\mathcal{C} \cup \mathcal{C}'\rfloor$.*

It is instructive to consider two candidates for simpler definitions for $\lfloor\cdot\rfloor$. These definitions are appealing at a first glance:

1. $\{c \in \mathcal{C} \mid c \neq \emptyset \wedge \nexists\, c' \in \mathcal{C}.\; c \subset c'\}$ (Greatest elements)
2. $\{c \in \mathcal{C} \mid c \neq \emptyset \wedge \nexists\, c' \in \mathcal{C}.\; c' \neq \emptyset \wedge c' \subset c\}$ (Smallest elements)

In the first formulation, the smaller elements do not contribute separate classes. In the second formulation, the additional paths covered by the larger elements do not contribute. For both formulations, the valuable property (9.) in Theorem 1 is invalid. For the second formulation, not even (7.) is valid anymore.

If we are faced with infeasibility, non-minimality, and incompleteness, $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ should not be applied blindly for the sake of minimality and completeness. We should understand what causes non-minimality and incompleteness in each particular case. In fact, there are sometimes good reasons why some of the desirable properties do not hold. We illustrate this problem in the following theorem and the corresponding proof.

Theorem 2.

1. *$CC_{\Sigma} : \sigma$ is disjunctive and complete but not necessarily feasible for all Σ, σ.*
2. *$FC_{\Sigma} : \sigma$ is neither necessarily feasible, complete, minimal, nor disjunctive for all Σ, σ.*

Proof.

1. CC is disjunctive because constants only occur as path terminators, i.e., a path can never contain two (different) constants. Thereby, the classes of paths for the various constants are disjoint. CC might indeed contain the empty set if certain sorts and thereby constants of that sort are not reachable from the given σ. CC is complete because all constants are exhausted by it, and otherwise the paths are not constrained.
2. FC might indeed contain the empty set if certain sorts and thereby symbols of that sort are not reachable from the given σ. $FC_{\Sigma} : \sigma$ is incomplete if there are no other function symbols of sort σ but constant symbols. $FC_{\Sigma} : \sigma$ is not disjunctive. Here is an example: Consider $FC_{\Sigma_\lambda} : Exp$. Obviously, $func_{\mathbf{apply}} : Exp$ and $func_{\mathbf{lambda}} : Exp$ have some paths in common. This reflects that there are paths in which both **apply** and **lambda** occur. As for non-minimality, we see that $func_{\mathbf{lvar}} : Exp$ subsumes $func_{\mathbf{apply}} : Exp$ and $func_{\mathbf{lambda}} : Exp$, since any expression of sort Exp necessarily has to exercise the symbol **lvar**.

The section is concluded with the ultimate definition of a coverage criterion. It is certainly desirable that a proper coverage criterion $\mathcal{C}$ meets completeness, that is, every possible path is contained in some class in $\mathcal{C}$. In the following definition, we also postulate minimality. Our experience with defining coverage criteria indicates that the requirement for minimality is possibly debatable. We definitely do not insist on disjunctive sets since the specifications one writes down in the first place hardly meet this requirement. In fact, the above proof illustrates that non-disjunctive sets of classes might be sensible. An operator for recovery of disjunctive sets is conceivable.

Definition 11. *A set $\mathcal{C}$ of classes of paths of sort σ is a* proper coverage criterion *if $\mathcal{C}$ is minimal and complete.*

Although the definition talks about sets of classes of paths rather than sets of regular path expressions, we will usually reason about coverage criteria mainly at the level of the path expressions.

5 Coverage criteria

We develop various coverage criteria starting from an abstract form of rule coverage known for context-free grammars. The subsequent criteria are original contributions of the article.

5.1 Branch coverage

CC and FC from Definition 5 provide the basis for a simple coverage criterion called *branch coverage*. This criterion corresponds to rule coverage when applied to context-free grammars [Pur72b]. Branch coverage (BC) is a suitable term because given a sort τ, the various function symbols of sort τ provide the alternatives to construct terms of sort τ. BC enforces that all symbols of all sorts are exercised. Thus, we might say that all branches are covered.

Definition 12. *BC_{Σ} denotes the* branch coverage criterion for *Σ:*

$$BC_{\Sigma} : \sigma = \lfloor CC_{\Sigma} \cup FC_{\Sigma} \rfloor$$

Example 7. Consider the following two terms:

– **apply(lambda(zero, tvar(zero), lvar(zero)), lvar(succ(zero)))**
– **apply(lvar(zero), lvar(zero))**

The first term exercises BC_{Exp} because it covers all function symbols of the signature from Example 1. Instead of this large term, we could also favour smaller terms covering the various function symbols in separation. The second term above is the smallest term (in terms of the number of function symbols involved) covering **apply**.

Theorem 3.

1. *$BC_\Sigma : \sigma$ is a proper coverage criterion for all Σ, σ.*
2. *$CC_\Sigma : \sigma \sqsubseteq BC_\Sigma : \sigma$, $FC_\Sigma \sqsubseteq BC_\Sigma$ for all Σ, σ.*
3. *There exists Σ, σ such that $CC_\Sigma : \sigma \not\equiv BC_\Sigma : \sigma$, $FC_\Sigma : \sigma \not\equiv BC_\Sigma : \sigma$.*

Proof.

1. Completeness follows from the union with CC_Σ which is complete on its own.
2. Implied by (11.) in Theorem 1.
3. Here are examples exploring the additional expressiveness gained by BC_Σ compared to CC_Σ and FC_Σ: Consider again the signature from Example 1. We have that $CC_{\Sigma_\lambda} : \mathit{Exp} \not\equiv BC_{\Sigma_\lambda} : \mathit{Exp}$ because **lvar**(**zero**) exercises $CC_{\Sigma_\lambda} : \mathit{Exp}$, but not $BC_{\Sigma_\lambda} : \mathit{Exp}$. We also have that $FC_{\Sigma_\lambda} : \mathit{Exp} \equiv BC_{\Sigma_\lambda} : \mathit{Exp}$ because all terms necessarily exercise the one and only constant **zero**. Assume that we add another constant **int** $:\to$ *type* to Σ_λ resulting in an extended signature $\Sigma_{\lambda'}$. Then, we can show that $FC_{\Sigma_{\lambda'}} : \mathit{Exp} \not\equiv BC_{\Sigma_{\lambda'}} : \mathit{Exp}$ because there are terms exercising $FC_{\Sigma_{\lambda'}} : \mathit{Exp}$ without using **int**, e.g.:

$$\mathbf{apply}(\mathbf{lambda}(\mathbf{zero}, \mathbf{arrow}(\mathbf{tvar}(\mathbf{zero}), \mathbf{tvar}(\mathbf{zero})), \mathbf{lvar}(\mathbf{zero})), \mathbf{lvar}(\mathbf{succ}(\mathbf{zero})))$$

5.2 Position coverage

A minor generalisation of branch coverage is to take parameter positions of the function symbols into account. We call this generalisation position coverage (PC). As it will turn out, PC is not a proper refinement of BC, but PC and BC are simply equivalent. Hence, we consider PC as an illustrative example.

Definition 13. *PC_Σ denotes the* position coverage criterion *for Σ:*

$$PC_\Sigma : \sigma = \lfloor CC_\Sigma \cup \{ pos_{f,i} \mid f \in F, 1 \leq i \leq arity(f) \} \rfloor \text{ where } pos_{f,i} = \mathsf{Some}\ f\ i\ \mathsf{Some}$$

Theorem 4.

1. *$[\![pos_{f,i} : \sigma]\!] \subseteq [\![func_f : \sigma]\!]$ for all Σ, σ, f, i where $1 \leq i \leq arity(f)$.*
2. *$\ldots \subset \ldots$ in 1. if $arity(f) > 1$.*
3. *There exist Σ, σ such that $|BC_\Sigma : \sigma| < |PC_\Sigma : \sigma|$.*
4. *t exercises $[\![func_f : \sigma]\!] \Leftrightarrow t$ exercises $[\![pos_{f,i} : \sigma]\!]$ for all Σ, σ, $t \in \mathcal{T}_\sigma(\Sigma)$.*
5. *$BC_\Sigma : \sigma \equiv PC_\Sigma : \sigma$ for all Σ, σ.*

Proof.

1. $func_f$ is subdivided into the various $pos_{f,i}$.
2. Obvious. (Hence, one might think that PC is more fine-grained than BC.)
3. Obvious. (Still, one might think that PC is more fine-grained than BC.)
4. Any term with an application of f necessarily exercises the various parameter positions of f. Hence, there are paths in $[\![t]\!]$ which do not exercise a certain parameter position i of f, but there are necessarily other paths which do.
5. Directly implied by 4.

5.3 Context-dependent branch coverage

Let us consider a more involved criterion. We want to characterise paths where a function symbol g is used on the i-th parameter position of another function symbol f. If we consider all possible classes, we obtain—in a sense—a context-dependent version of BC. Thus we call it *context-dependent branch coverage* ($CDBC$).

Definition 14. *$CDBC_\Sigma$ denotes the* context-dependent branch coverage criterion for *Σ:*

$$CDBC_\Sigma : \sigma = \left\lfloor CC_\Sigma \cup \left\{ link_{f,i,g} \;\middle|\; \begin{array}{l} f : \sigma_1 \times \cdots \times \sigma_n \to \sigma_0 \in F, \\ g : \cdots \to \sigma_i \in F, \\ 1 \leq i \leq n \end{array} \right\} \right\rfloor$$

where $link_{f,i,g}$ $=$ Some $f\ i\ g$ Some.

Note how the types of f and g are constrained in the definition to enforce well-formed expressions, that is, g's result sort is the same as f's i-th parameter sort.

Theorem 5.

1. *$CDBC_\Sigma : \sigma$ is a proper coverage criterion for all Σ, σ.*
2. *$BC_\Sigma \sqsubseteq CDBC_\Sigma$ for all Σ, σ.*
3. *There exist Σ, σ such that $BC_\Sigma : \sigma \not\equiv CDBC_\Sigma : \sigma$.*

Proof. Omitted. The scheme of the proof for Theorem 3 can be reused.

Example 8. These are all combinations of function symbols, parameter positions, and function symbols on the latter positions for the signature Σ_λ from Example 1:

lvar 1 **zero**	**lambda** 3 **apply**	**tvar** 1 **succ**
lvar 1 **succ**	**apply** 1 **lvar**	**arrow** 1 **tvar**
lambda 1 **zero**	**apply** 1 **lambda**	**arrow** 1 **arrow**
lambda 1 **succ**	**apply** 1 **apply**	**arrow** 2 **tvar**
lambda 2 **tvar**	**apply** 2 **lvar**	**arrow** 2 **arrow**
lambda 2 **arrow**	**apply** 2 **lambda**	**succ** 1 **zero**
lambda 3 **lvar**	**apply** 2 **apply**	**succ** 1 **succ**
lambda 3 **lambda**	**tvar** 1 **zero**	

As we argued for branch coverage in Example 7, we might favour either small test cases exercising these combinations, or larger test cases covering more of the combinations at once. For brevity, we do not include an (already somewhat larger) test set exercising *CDBC* for the running example. In Figure 3, one particular set contributing to $CDBC_{\Sigma_\lambda}$: *Exp* is shown, namely the class which deals with a λ-abstraction on the first parameter position of a function application.

5.4 Reachability coverage

There is a rather obvious way how context-dependent coverage can be generalised. *CDBC* is sufficient to exercise all possible function symbols in all direct contexts, that is in parameter positions of function symbols of a suitable sort. We can also look for remote pairs of function symbols. Context-dependent branch-coverage only relates adjacent symbols in terms.

Definition 15. *RC_Σ denotes the* reachability coverage criterion for *Σ:*

$$RC : \sigma = \left\lfloor CDBC_\Sigma \cup \left\{ remote_{f,i,g} \;\middle|\; \begin{array}{l} f : \sigma_1 \times \cdots \times \sigma_n \to \sigma_0 \in F, \\ g : \cdots \to \sigma_0' \in F, \\ 1 \leq i \leq n, \sigma_0' \neq \sigma_i \end{array} \right\} \right\rfloor$$

where $remote_{f,i,g}$ $=$ Some $f\ i$ Some g Some.

The expression $remote_{f,i,g}$ characterises paths where g is reachable from f via the i-th parameter position of f. We restrict the types of f and g in a way that g cannot directly be applied on the i-th parameter position of f because this case is handled by *CDBC* anyway. Note also that a more coarse-grained formulation of *RC* is conceivable, where g to be reachable from f is not restricted to a certain parameter position i of f.

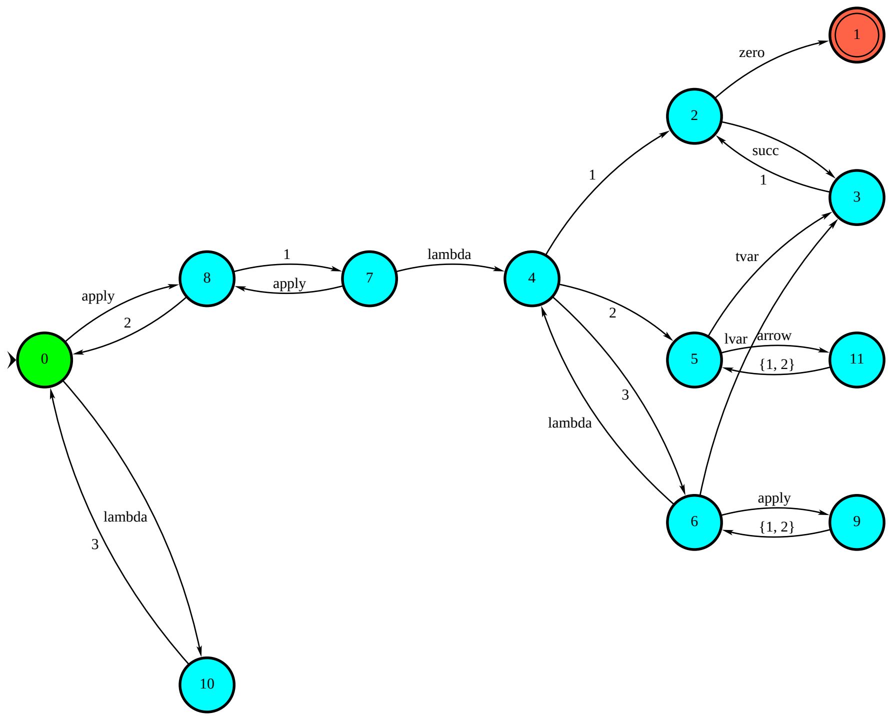


Fig. 3. The FSA accepting $[\![link_{\mathbf{apply},1,\mathbf{lambda}} : Exp]\!]$

Theorem 6.

1. *$RC_\Sigma : \sigma$ is a proper coverage criterion for all Σ, σ.*
2. *$CDBC_\Sigma \sqsubseteq RC_\Sigma$ for all Σ, σ.*
3. *There exist Σ, σ such that $CDBC_\Sigma : \sigma \not\equiv RC_\Sigma : \sigma$.*

Proof. Omitted. The scheme of the proof for Theorem 3 can be reused.

RC is potentially erasing in the sense that a considerable amount of classes is likely to be infeasible, i.e., certain combinations of function symbols cannot occur in the given order in a path. This insight triggers the question if RC could maybe be formulated in a more verbose style so that only feasible classes are included. In principle, this is possible since we can constrain the combinations by reachability arguments. There is room for future work to suggest an ultimate style of the definition of coverage criteria so that feasibility or other properties are more likely to hold.

5.5 Unfolding coverage

The coverage criteria so far did not address recursion in the underlying term algebra. A specific treatment of recursion is sensible in the same way as special rules do exist for testing loops and recursive functions in imperative programs. We present a coverage criterion enforcing a certain number of recursive unfoldings for a fixed sort τ. To this end, we also need to introduce an inductive scheme for the definition of sets of regular expressions. Let us start with $UC_{\Sigma,\tau,0}$ characterising all paths not at all restricted regarding possibly recursive occurrences of sort τ:

$$UC_{\Sigma,\tau,0} : \sigma = \{\mathsf{Some}\}$$

We defined $UC_{\Sigma,\tau,0}$ as a set of expressions rather than a single expression for conformity because the subsequent $UC_{\Sigma,\tau,i}$ are proper sets of expressions. $UC_{\Sigma,\tau,1}$ exhausts all symbols of sort τ at least once:

$$UC_{\Sigma,\tau,1} : \sigma = \{\mathsf{Some}\, f\, \mathsf{Some} \mid f : \cdots \to \tau \in F\}$$

The corresponding set of classes is potentially incomplete since there might be paths where no function symbol of sort τ is involved. Since we are solely interested in recursive unfoldings of symbols of sort τ, we can accept this incompleteness. Non-minimality can happen in the same harmless way as for FC_{Σ}. The next level of recursive unfolding is modelled by the following set:

$$UC_{\Sigma,\tau,2} : \sigma = \{\mathsf{Some}\, f\, \mathsf{Some}\, g\, \mathsf{Some} \mid f, g : \cdots \to \tau \in F\}$$

Note that we cannot use star-notation (*) subscripted with an i for the number of recursive unfoldings to define $UC_{\Sigma,\tau,i}$. The problem is that we rely on a new variable for τ-sorted function symbols for each new level of recursive unfolding. Instead, an inductive scheme is convenient for the definition of $UC_{\Sigma,\tau,i}$.

Definition 16. $\lceil \lfloor UC_{\Sigma,\tau,i} \rfloor \rceil$ *denotes the* unfolding coverage criterion for Σ with i unfoldings of sort τ *where*

$$UC_{\Sigma,\tau,i} : \sigma = \begin{cases} \{\mathsf{Some}\}, & \textit{for } i = 0 \\ \{\mathsf{Some}\, f\, e \mid f : \cdots \to \tau \in F, e \in UC_{\Sigma,\tau,i-1}\}, & \textit{for } i > 0 \end{cases}$$

Unfolding coverage is not a refinement of any other coverage criterion discussed so far. It is, for example, different from FC, CC, and BC because we do not attempt to exercise all symbols but only symbols of the distinguished sort τ. It would be possible to derive UC from another criterion. We favour the status of UC to be solely concerned with the coverage of recursive unfoldings. The operator $\lceil \cdot \rceil$ is used to recover completeness. Note that in all other definitions of coverage criteria, we only had to recover minimality because the complete CC was used as a starting point.

6 Concluding remarks

Implementation There are two important uses of test case characterisation and test set coverage criteria, namely coverage analysis and test set generation. Both uses are easy to accomplish using basic regular language theory (cf. [HU80,ASU86]) and corresponding tool support.

As for coverage analysis, we are interested in the coverage of a test set T w.r.t. a criterion modelled by some set $\mathcal{C}$. First, we accumulate all paths induced by the test data. Because the classes $\mathcal{C}$ of paths induced by a coverage criterion are usually infinite, the various classes in $\mathcal{C}$ are rather represented by regular expressions or grammars. Then, all the paths in T are just parsed by the FSAs corresponding to $\mathcal{C}$. If a certain FSA accepts some path, the corresponding class is covered. Alternatively, T can also be regarded as a regular language. A class in $\mathcal{C}$ is covered if the intersection with the regular language corresponding to T is non-empty.

As for test set generation, we are interested in the generation of test sets achieving coverage w.r.t. a coverage criterion modelled by some set $\mathcal{C}$ of classes of paths. All possible test data can basically be enumerated because the terms of a term algebra can be enumerated. For every term t, we check if it exercises one of the classes in $\mathcal{C}$. If this is the case, we add t to the test set, and c is removed from $\mathcal{C}$ for the rest of the generation process. This process will terminate since all the classes in $\mathcal{C}$ are feasible. The efficiency of the basic generation algorithm can be very much improved if the generation of terms is driven by $\mathcal{C}$ rather than solely by the underlying signature. As we will see below, the bottleneck (if any) of the approach is not the generation but the class minimalisation.

Proof of concept Some feasibility experiments have been performed. We considered, for example, context-dependent branch coverage for full Pascal. The experiment is summarized in Figure 4. We derived all FSAs for the CDBC classes for the start symbol of the Pascal grammar. We also minimalised the classes via $\lfloor \cdot \rfloor$. Finally, we also generated a test set achieving coverage. Test set generation consists of two phases. First, paths are derived from the automata for the various classes. Then, the paths are completed to complete terms based on shortest completions [Pur72b]. All the computations were performed in SWI-Prolog [Wie00] relying on the FSA utilities [Noo97]. The system we used was a Sun Ultra 5. It turns out that the performance for computing the

Category	Figure
Grammar constructors	142
States FSA for $\mathcal{PG}$	141
Generation time FSA for $\mathcal{PG}$	5 sec
CDBC	
Number of classes	598
States for FSA of largest class	153
Generation of all automata	5 min
Minimalisation time	2 h
Path generation	5 min
Path completion	7 min

Fig. 4. Case study for Pascal

FSAs from a CDBC is acceptable. Coverage analysis and test set generation does also not pose performance problems. Minimalisation is problematic which is not too much of a surprise due to the quadratic formulation of $\lfloor\cdot\rfloor$. We already had to use a simple heuristic for $\cdot \subset \cdot$ on FSAs to speed up the computation of $\lfloor\cdot\rfloor$. In this heuristic, we first check $\cdot \subset \cdot$ for the alphabets of the FSAs. The challenging question is if the approach will scale up for even larger grammars, and other coverage criteria than CDBC. The problem with the complexity of the minimalisation encouraged us to think of other ways to approve coverage criteria. Since one is obliged to approve non-minimality anyway, an obvious question is, if we can discipline the definition of coverage criteria any further so that minimality is actually guaranteed. This is a topic for future work.

Generic testing technology We envision that the discussed framework contributes to generic testing technology to form an integral part of generic language technology (just in the same way as nowadays parser generators, tree-walkers, code generators and others do). Similar ideas are pursued in [DRW96] but on less formal grounds. According to the general tone in [HK00], a testing framework for a language is yet another tool to be derived from the formal language definition (maybe enriched by ingredients specific to testing). Generic language technology enriched by generic testing technology would be beneficial in the following contexts:

- Design of domain-specific languages: To get acquainted with a language under development, test case generation is helpful. One could simply derive a complete program from a particular pattern at hand. Coverage analysis is useful to approve the completeness of a test suite for an (evolving) language. The link between testing and language design has first been pointed out in [Rie92]. We might also include intermediate and exchange formats, and also virtual machines in our discussion (cf. [SB99]). Language design is a challenging application domain for testing because one is maybe not satisfied with syntactically correct test programs, but the generated programs should also be statically correct. In [HL00], we show in a specific setting how this can be accomplished.
- Automated software renovation: Test case characterisation can be used for querying source code (cf. [PP94,MER99]) to approve transformation rules in automated software renovation [CC90,BSV00]. Also, coverage analysis can be used to backup assumptions in the renovation tools. One particular technique is to minimalise a grammar according to a code base at hand. Testing concepts are also useful for the mere recovery of the grammars needed in software renovation if syntax-based tools are employed (for the relevant languages in the code base). We call the corresponding discipline grammar re-engineering or grammar recovery [LV01]. Coverage analysis, coverage visualisation, and test set generation considerably helps to validate the correctness and completeness of a recovered grammar (cf. [Läm01]).

References

[Abi97] S. Abiteboul. Querying Semi-Structured Data. In F.N. Afrati and P. Kolaitis, editors, *Database Theory—ICDT'97, 6th International Conference*, number 1186 in LNCS, pages 1–18, Delphi, Greece, 8–10 January 1997. Springer-Verlag.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: principles, techniques, tools.* Addison-Wesley, 1986.

[Bei90] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New York, 2nd edition, 1990.

[BSd82] F. Bazzichi and I. Spadafora. An Automatic Generator for Compiler Testing. *IEEE Transactions on Software Engineering*, 8(4):343–353, July 1982.

[BSV00] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of Components for Software Renovation Factories from Context-free Grammars. *Science of Computer Programming*, 36(2–3):209–266, 2000.

[BT00] E. Brinksma and J. Tretmans. Testing Transition Systems: An Annotated Bibliography. In F. Cassez, C. Jard, B. Rozoy, and M. Ryan, editors, *Summer School MOVEP'2k - Modelling and Verification of Parallel Processes*, pages 44–50, Nantes, July 2000.

[Bur94] C.J. Burgess. The Automated Generation of Test Cases for Compilers. *Software Testing, Verification and Reliability*, 4(2):81–99, jun 1994.

[CC90] E.J. Chikofsky and J.H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13—17, 1990.

[CRV$^+$80] A. Celentano, S.C. Reghezzi, P.D. Vigna, C. Ghezzi, G. Granata, and F. Savoretti. Compiler Testing using a Sentence Generator. *Software - Practice and Experience*, 10:897–918, 1980.

[Den91] R. Denney. Test-Case Generation from Prolog-Based Specifications. *IEEE Software*, 8(2):49–57, March 1991.

[DRW96] P.T. Devanbu, D.S. Rosenblum, and A.L. Wolf. Generating Testing and Analysis Tools with Aria. *ACM Transactions on Software Engineering and Methodology*, 5(1):42–62, January 1996.

[FW88] P.G. Frankl and E.J. Weyuker. An Applicable Family of Data Flow Testing Criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[HK00] J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM SIGPLAN Notices*, 35(3):39–48, March 2000.

[HL00] J. Harm and R. Lämmel. Two-dimensional Approximation Coverage. *Informatica*, 24(3), 2000.

[HS89] William Homer and Richard Schooler. Independent testing of compiler phases using a test case generator. *Software - Practice and Experience*, 19(1):53–62, January 1989.

[HU80] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, N. Reading, MA, 1980.

[Jac96] O. Jack. *Software Testing for Conventional and Logic Programming.* Number 10 in Programming Complex Systems. Walter de Gruyter, Berlin, 1996.

[Läm01] R. Lämmel. Grammar Testing. In *Proc. of Fundamental Approaches to Software Engineering (FASE) 2001*, number 2029 in LNCS. Springer-Verlag, 2001.

[LV01] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. Submitted; Available at `http://www.cwi.nl/~ralf/`, July 2001.

[MER99] MERANT International Limited. *Revolve User Guide*, May 1999. Issue 7.

[Mye79] G.J. Myers. *The Art of Software Testing.* Wiley - Interscience, New York, 1979.

[Noo97] G. van Noord. FSA Utilities: A Toolbox to Manipulate Finite-State Automata. In *Automata Implementation*, number 1260 in LNCS. Springer-Verlag, 1997.

[PP94] S. Paul and A. Prakash. Supporting Queries on Source Code: A Formal Framework. *International Journal of Software Engineering and Knowledge Engineering*, 4(3):325–348, 1994.

[Pur72a] P. Purdom. Erratum: "A Sentence Generator for Testing Parsers" [BIT **12**(3), 1972, p. 372]. *BIT*, 12(4):595–595, 1972. See [Pur72b].

[Pur72b] P. Purdom. A sentence generator for testing parsers. *BIT*, 12(3):366–375, 1972. See [Pur72a].

[Rie92] G. Riedewald. The LDL—Language Development Laboratory. In U. Kastens and P. Pfahler, editors, *Compiler Construction, 4th International Conference, CC'92, Paderborn, Germany*, number 641 in LNCS, pages 88–94. Springer-Verlag, October 1992.

[RW85] S. Rapps and E.J. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.

[SB99] E.G. Sirer and B.N. Bershad. Using Production Grammars in Software Testing. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, pages 1–14, Berkeley, CA, October 3–5 1999. USENIX Association.

[VK95] R. Vemuri and R. Kalyanaraman. Generation of design verification tests from behavioral VHDL programs using path enumeration and constraint programming. *Trans. VLSI Systems*, 3:201–214, 1995.

[Wey89] E.J. Weyuker. In Defense of Coverage Criteria. In *Proceedings of the 11th International Conference on Software Engineering*, page 361, May 1989.

[Wie00] J. Wielemaker. SWI-Prolog 3.4, Reference Manual. University of Amsterdam, Dept. of Social Science Informatics (SWI), 2000.

Using SDL Tools to Test Properties of Distributed Systems

Hesham Hallal[1], *Alex Petrenko*[1], *Andreas Ulrich*[2], *Sergiy Boroday*[1]

[1] Centre de Recherche Informatique de Montreal (CRIM),
550 Sherbrooke West, Suite 100
Montreal, H3A 1B9, Canada
{hallal, petrenko, boroday}@crim.ca

[2] Siemens AG, CT SE 1,
Otto-Hahn-Ring 6,
D-81730 Munich, Germany
andreas.ulrich@mchp.siemens.de

Abstract

We present an ongoing project on reasoning on properties of distributed systems based on monitoring of their executions. The proposed approach uses SDL to model an execution trace of the system under test and an existing model checker to perform the analysis of properties of interest specified in the SDL-like language GOAL. For this purpose, we use the available ObjectGEODE tool set. We describe how SDL models are built from collected traces, and show how the desired properties are specified. An example is used to illustrate the approach. The proposed methodology can be applied to test distributed systems and to diagnose their faults.

1 Introduction

Recent technological advances, especially in communications, triggered a growing need for distributed systems. However, the cost of developing such applications is getting ever higher. In fact, the main characteristics of distributed systems, which include asynchrony and absence of a global timing reference, add to the complexity of their design and test. In addition, development of distributed systems rarely yields formal specifications of their behavior that would make formal methods fully applicable in the testing phase. A number of tools that ease the development problems in the debugging and testing phase have been developed, both in the academia and industry. Such tools usually rely on other tools that provide monitoring of distributed systems and produce log files of execution traces. Debugging tools offer various levels of automation of testing activities, from visualization of traces to property verification. There exists a large body of work on developing various tools to visualize traces, see, e.g., [2, 15, 19]. Their goal is to facilitate efforts of the designer or tester for locating and correcting bugs by filtering out unrelated data and properly visualizing executions of his concern. The analysis is performed manually either online (simultaneously with the system execution) or post mortem. The other group of tools targets the analysis phase by offering means to automatically verify whether the system under test exhibits certain properties [8, 9, 13]. It is the task of the tester to formally specify the “suspected” property of the system. Such property verification is based on execution traces using specially developed model checking mechanisms.

Developing a tool for testing properties in execution traces, one usually faces a choice of either to elaborate algorithms and to implement them in a specialized tool for a specific class of properties or to reuse an off-the-shelf model checker. In the first scenario, one faces the daunting task of implementing from scratch all the phases of the approach, i.e., modeling the system, specifying the properties of interest, and, most importantly, building a model checker. Realizing difficulties inherent to this approach, the work in [13] proposes to reuse an existing model checker in the last step of the analysis. However, we are not aware of any attempt to apply commercial or research model checkers to the whole process of property verification even in the post mortem mode. Reuse of these tools allows the developers of debugging tools to rely on reliable and highly sophisticated products, in which many years of research and experience have been invested.

In this paper, we report on an ongoing research project that has the goal to evaluate the applicability of a commercial model checking environment to automate the process of post-mortem property testing in a recorded execution trace of a distributed system under test. We decide to use the Specification and Description Language (SDL) to model the aspects of the system under test. SDL was chosen because of its ability to model adequately communicating systems as well as its user friendliness. In addition, the existence of commercial design and analysis tools that support the language is a major point in favor of SDL. We selected ObjectGEODE (OG) from Verilog (now Telelogic). This tool allows not only modeling systems in SDL but also performing simulation and model checking. The OG tool set provides the SDL-like language GOAL (GEODE Object Automata Language) to specify the desired properties, which simplifies the task of the user by relieving him from the burden of mastering temporal logic [1]. This, in fact, adds greatly to the practicality of the tool.

The main issue in our project — automating the process of performing trace-based analysis of distributed systems — can be detailed as follows: Given a system under test (SUT), an executed trace that was collected by monitoring the SUT's behavior before, and a set of properties (certain characteristics of interest), we need to verify within the OG environment whether the SUT's behavior represented by the trace exhibits the required properties. To do so, we build an SDL model of the system based on the recorded trace, specify the properties of interest in GOAL, and use the OG model checker to verify whether the specified properties are present or missing in the trace.

The remainder of this paper is organized as follows. The next section summarizes related work. In Section 3, we describe our approach to solve the trace-based analysis problem in distributed systems. We detail the approach and discuss the basic conditions for its validity and consistence. In Section 4, we discuss some important properties for the trace-based verification of distributed systems. Then in Section 5, we illustrate the approach and the use of the tools through an example. Finally, we conclude the paper in Section 6.

2 Related Work

Different approaches exist for modeling properties of distributed systems. In [16], an approach is devised to allow a debugger to halt the execution of the system at some specified breakpoints that are defined as predicates of system's events. However, the expression power of the linked predicates is limited. In addition, the local state of each process needs to be known which is not always feasible, especially in the case of distributed systems.

Much work on trace based analysis was done in the field of intrusion detection. The work in [14] shows such an approach that targets intrusion detection in network systems and models intrusion patterns using Colored Petri-Nets. The approach can be generalized to cover a wider range of properties, but it still lacks an implementation that shows its efficiency. In [8], flow graphs are used to represent potential communications between the processes of a distributed system. Properties, meanwhile, are represented using quantified regular expressions (QRE). This approach requires deep knowledge of tiny details in the processes of the tested system. This approach is not feasible all the time and defies the advantage of using execution traces to test the system's behavior. The approach in [18] describes the tool GrIDS to detect large scale intrusion attacks on network systems. The concept is to build activity graphs of the executions of the various processes in the system by monitoring them individually and to analyze them based on some reference rules to decide on an intrusion. The approach requires heavy means to protect the GrIDS modules themselves against attacks.

The papers [2] and [19] present an approach to visualize the collected traces. [9] and [13] describe an approach, centered on the concept of the lattice, to perform trace checking in distributed systems. Following this approach, a lattice is built, based on the monitored events and the relations between them, to represent the system under test. The lattice is then used in a model-checker to verify the behavior of the system against a desired property. In a recent work [15], a method of specifying abstraction hierarchies to define level-wise views of a distributed message-based system is outlined. This method utilizes event-pattern mappings and complex events to represent a system's behavior.

Similar to the approach in [13], we base our model on the concept of the lattice. However, we suggest going further. We describe in SDL each component of the SUT involved in the given trace as a state machine, and we let the SDL simulator in OG build the composite (global) machine of the system (an SDL state graph) that represents the same interleavings as the lattice. This is done at runtime when the model checker verifies the given property (pattern). We believe that our approach steps further in the direction of full automation of the process especially by means of a front-end tool that builds the SDL models from the collected trace and helps the user specify the properties of interest in the GOAL language.

3 The Proposed Approach

3.1 Overview

We base our work on two fundamental concepts that reflect concurrency in communicating systems: a partially ordered set (poset) of events, where the partial order is the traditional "happens before" relation [3], and the corresponding lattice. In fact, these two concepts are at the heart of the main existing approaches to analyze traces of distributed systems. For example, the approach of [13] considers building the lattice of the poset of the collected events and performing the verification of a pattern on the lattice using existing model checking techniques. Similar to this approach, we consider using an existing tool, rather than relying on home developed algorithms and methods for model checking. In our post mortem analysis approach, we first build an SDL model of a system from a collected trace (the trace is completed, i.e., we do a post mortem analysis of a performed system run). Then we use the model checker, i.e., the ObjectGEODE simulator, to perform the checking of a given property.

In detail, an SDL model of the tested system that relies on a given trace reflects the following aspects: structure, behavior, communication, and data. The structure of the system can be modeled using the hierarchy of system/block/process/procedure statements in SDL. In our case, it is sufficient to define a system with a single block that is composed of several processes. The overall behavior of the system is modeled by the joint behavior of a set of communicating processes in SDL. These processes correspond to entities of the real system whose behaviors are recorded in the trace; thus making each of the processes linear, as in most cases we cannot identify any two states of it, so no cycles can be deduced. The representation of a process can be obtained by projecting the collected trace into the set of events it executes: send, receive, and local events and subsequently inserting states in between communication events, while representing local events as SDL tasks. The asynchronous communication between processes is achieved via signals with optional signal parameters (that represent exchanged data) and channels. Input signals to a process are stored in a queue before reading them. Thus, unknown delays in real communication channels of the distributed system are represented by those of input queues, associated to each SDL process. This means that in our framework, the whole communication media of the system is simply modeled with individual queues of SDL processes.

The property to be checked (the pattern) is expressed as an observer [10] in the GOAL language. A GOAL observer implements in fact a finite automaton with accepting states [1]. GOAL is similar to SDL, but has some syntactic and semantic differences [1]. Observers are usually described in terms of entities (objects, signals etc.) of the tested system, e.g., they can be associated to the SDL system. This makes communication signals directly accessible for observation while other model data, e.g., variables and states, are accessible to observers using probes. The latter represent pointers to SDL entities. In addition, GOAL allows the declaration of two types of designated states: *success* states and *error* states. Entering success states (error states) indicates that the system respects (violates) the property expressed in the observer. The use of the success/error convention is completely up to the user and has no formal meaning.

Once the representation of the distributed system and the property is complete, the ObjectGeode model checker can be employed to verify whether the system satisfies the property. This is achieved in the so-called exhaustive simulation mode, when the tool performs all the possible executions of the system and builds a state graph of the SDL system.

The state graph represents all the possible interleavings of events in the collected trace. To perform model checking, the tool builds the synchronous product of the observer and the specification of the system. The OG simulator outputs a report of the number of errors and successes encountered, and presents the scenarios that lead to the observer errors and successes.

The approach based on the use of the OG environment can be summarized in the form of a workflow shown in Figure 1. As it is clear from this figure, the implementation efforts are reduced to building just a front-end to OG containing the three main blocks:

1. System specification tool that builds an SDL specification from the collected trace.
2. Pattern specification tool that eases the process of writing the patterns for verification.
3. A user interface module that allows the operator to control the whole process.

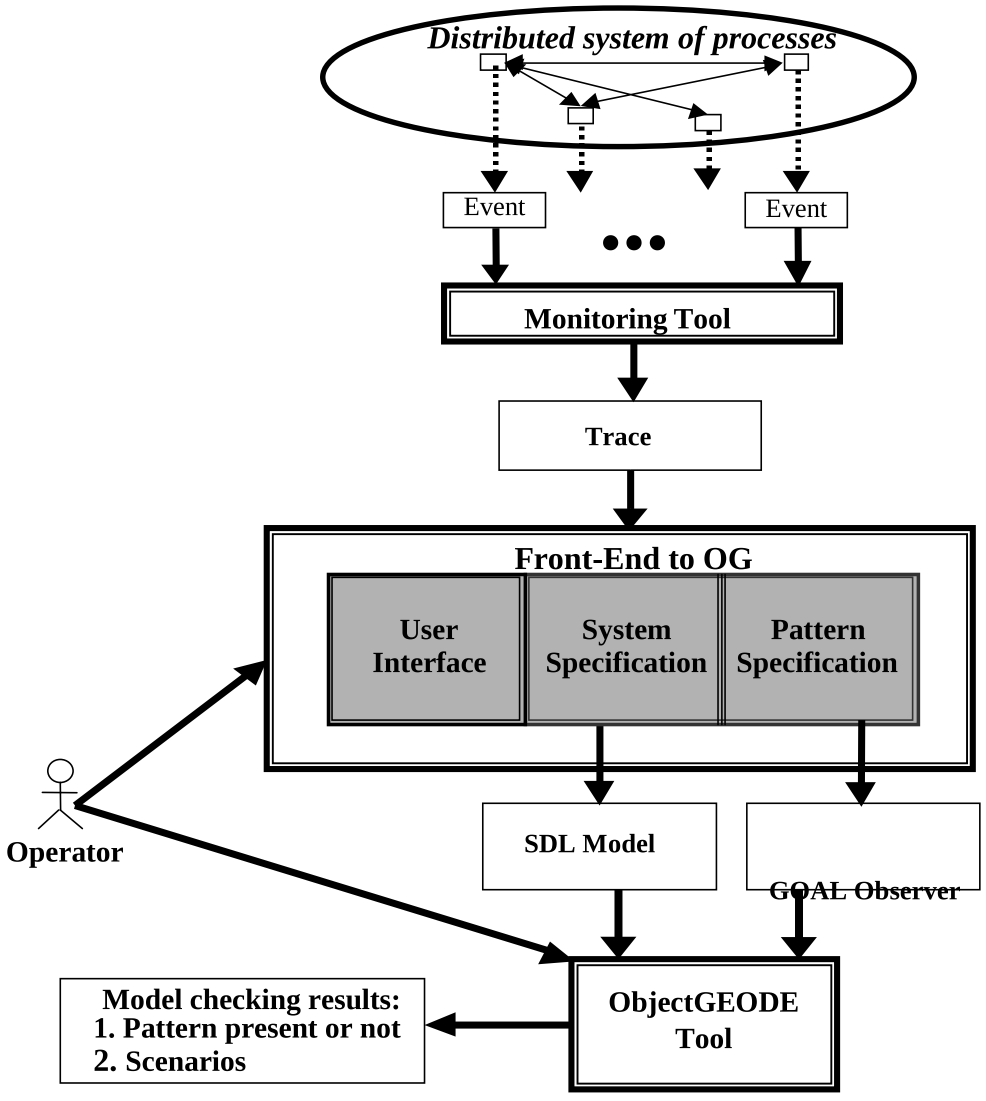


Figure 1: The diagram of the workflow.

In order to model the behavior of the SUT in SDL, as it is recorded in the execution trace, we have to:

a) Match each receive event with its corresponding send event.
b) Preserve in the SDL system's behavior the local order of events in each process.

3.2 Matching Receive with Send Events

The problem of matching receive and send events is crucial to determine the partial order of events and eventually build an adequate (SDL) model from the recorded trace. Much depends on how the distributed system is instrumented and what exactly is monitored. In this paper, we assume that each event in the trace collected by the monitoring system carry the following information:

1. Name and type of the event: Send, Receive, or Local.
2. ID of the issuing process.
3. The local ordinal number of the event in the process.
4. The source process (for Receive).
5. The destination process(es) (for Send).
6. The message parameters of the event: a list of typed parameters that includes a message name and other message attributes (for Send and Receive events).
7. The local parameters of the issuing process: a list of typed parameters that reflect its current state.

We assume that in a pair of source and destination processes, each Receive event matches with a single Send event in the sense that the values of all the message parameters coincide. Moreover, we take for granted that in the given trace, each Send event matches with at least one Receive event. The local ordinal number of events allows us to compute a total order of events in each process. Such total order has to be preserved in the partial order of the events in the whole system.

3.3 Preservation of the Event Orders

To illustrate a potential problem that is related to the violation of a local order by an SDL model directly deduced from a given trace, we consider the trace represented by the MSC in Figure 2. Process $P1$ issues a "Send" event $S1$, which reflects sending the message $m1$ to $P2$. Upon receiving the message, process $P2$ issues the "Receive" event $R1$. Similar events are generated when $P3$ sends the message $m2$ to $P2$. Notice that events $S1$ and $S2$ are independent while $R1$ and $R2$ depend on $S1$ and $S2$, respectively. The trace indicates that $R1$ occurs before $R2$ regardless of the order between $S1$ and $S2$. In other words, it can be easily deduced from the trace that $S1 \leq R1$, $S2 \leq R2$, $R1 \leq R2$, where $\leq$ is the "happened before" relation.

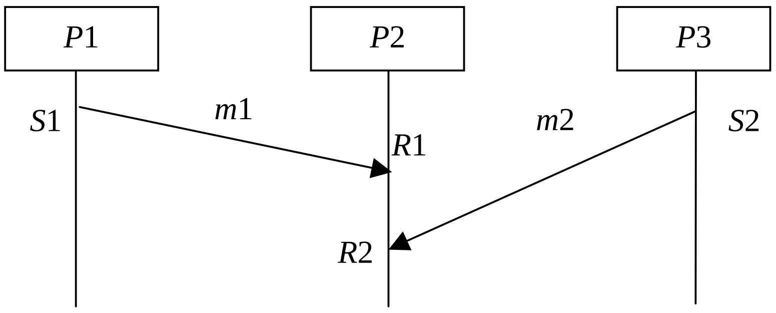


Figure 2: MSC of the sample trace.

A simple SDL process corresponding to $P1$ should just output a single output signal $m1$. The same applies to $P3$. $P2$ in turn should just receive two signals, namely, $m1$ followed by $m2$. When the OG simulator executes such SDL system, it treats $S1$ and $S2$ as concurrent events, i.e., firing them in all possible orders ([$S1$, $S2$] and [$S2$, $S1$]). This means the OG simulator reaches a global state where the queue of $P2$ contains the signals $m2$ and $m1$ received in the reversed order, i.e., $m2$ precedes $m1$, contrary to what the trace prescribes. In this state, the SDL specification of $P2$ foresees the reception of $m1$ only; this process considers $m2$ as an implicit input and discards it. The resulting global state graph would then contain an execution that violates the local order perceived by $P2$.

It turns out that SDL offers a simple, elegant solution to this problem. To prevent a signal loss, SDL contains the SAVE construct that prohibits the process in its current state from consuming the declared signals from the queue. The saved signal is kept for future consumption. In our example, the use of SAVE in the state with the input of signal $m1$ would prevent the SDL process from consuming $m2$ first and thus prevents the OG simulator from building executions that contradict to the given trace.

Therefore, we use the SAVE construct in each state of each SDL process to store all the unexpected signals that the OG simulator may try to put into the queue. Adding Save completes the construction of the model of the system that can be obtained from the given trace. The resulting SDL model allows us to verify any system property that can be specified in the Goal language.

4 Properties in Execution Traces

Here, we discuss the properties of distributed systems that can be verified with the approach. To address this issue we turn to the properties that are believed to be mostly used in practical applications. The existing research in the field has explored a wide range of properties that can be sought in distributed systems. These properties can be classified into two types: state based and event based.

Event based patterns allow detecting simple atomic or composite events in the behavior of the SUT. State based patterns, on the other hand, formulate assertions on the state variables of the processes in the SUT. The work of Dwyer et al. at Kansas State University to build a repository of specification patterns (similar to design patterns) [7], [11] shows an effort to cover both types of properties. For our project, we plan to eventually build a repository of typical and frequently used specification property templates in GOAL.

To make the specification of patterns portable, a mapping to several formalisms (LTL, CTL, GIL, QRE, and INCA Queries) has already been provided in [11]. There are two classes of patterns in the repository:

1. Occurrence patterns. They include patterns that express universality, existence, absence, and bounded existence of an event (or state) or sequence of events (or states).

2. Order patterns. These express relations between events (or states) or sequences of events (or states); and they include precedence, response, chain precedence, and chain response.

Each pattern is defined over a scope that expresses the extent of the execution, over which the pattern must hold. Five basic kinds of scopes exist in the repository. A scope of a pattern can be: *global* (the complete execution), *before* (up to a given state/event in the execution), *after* (the part of the execution after a given state/event), *between* (any part of the execution from one given state/event to another given state/event), or *after-until* (like *between* but the designated part of the execution continues even if the second state/event does not occur). In the repository, each scope is determined by specifying state/event delimiters for the pattern: the scope consists of all states/events beginning with the starting state/event and up to but not including the ending state/event [11].

We present here two illustrative examples, the universality and existence properties.

Universality: We consider the global universality of a predicate *P*, where *P* represents an assertion on events or state formula. This pattern can be stated in CTL as: AG(*P*). The corresponding observer is shown in Figure 3.

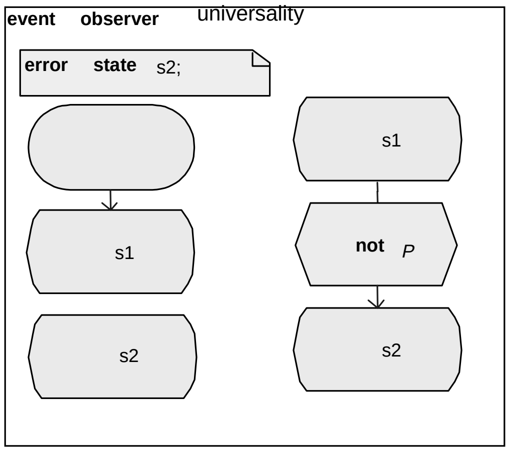


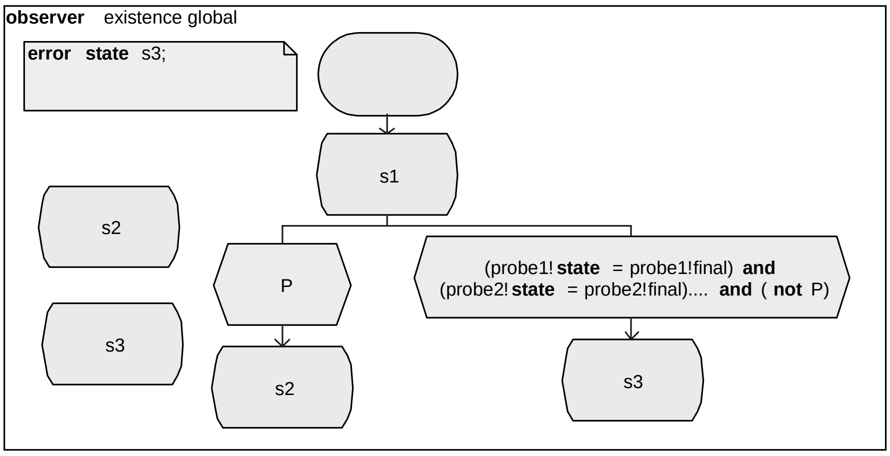
Figure 3: Observer for global universality.

To check that the property *P* holds in a given system, we should detect whether the negation of *P* ever occurs in it. In the case of events, the negation would mean the occurrence of any event other than in *P* (*P* might be a disjunction of events). In the case of a state formula, the negation of the Boolean predicate *P* is required. The corresponding observer in GOAL uses the WHEN construct to describe the universality property. The property holds if the observer does not terminate in the error state s2.

Existence: Here, we also consider the global existence of *P*. This pattern is stated in CTL as: AF(*P*). The corresponding observer is shown in Figure 4.

Clearly, this observer has to verify *P* on all the paths of the state graph built from the trace. To do so, we let the observer watch for *P* while concurrently checking the termination of the involved processes using assertions on the state of each process.

When all the processes reach their final states, and *P* has not been detected yet, the observer enters an error state. Note that the negation of the predicate itself should hold true when the processes reach the final states to account for the case when *P* holds in the final state of any of the processes.



observer existence global
error state s3;
s1
s2
s3
P
s2
(probe1! **state** = probe1!final) **and** (probe2! **state** = probe2!final).... **and** (**not** P)
s3

Figure 4: Observer for global existence of *P*.

The presented examples confirm that GOAL allows the expression of a wide range of patterns.

5 Example

We use an example to further illustrate our approach. Figure 5 shows an event sequence diagram deduced from a hypothetical trace. Here we do not discuss its structure to keep the example simple and transparent. In this trace, the two processes communicate between each other and issue local events. All Send, Receive and Local events are considered to be observable by the monitor.

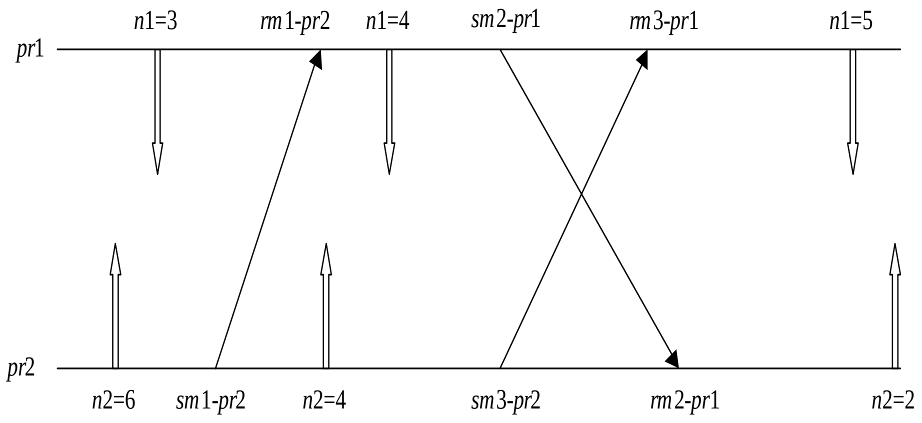


Figure 5: The event sequence of the collected trace, where hanging arrows correspond to local events, and sloped arrows to communications.

The two processes *pr*1 and *pr*2 have local variables *n*1 and *n*2, respectively, that are updated during the execution of the two processes. The values of these variables are sent in Local events only. From the trace in Figure 5, the representation of both processes can be extracted as a sequence of events belonging to either process.

Process *pr*1 is represented in the trace by the following event sequence:

1. Local event with *n*1 instantiated to 3, *n*1=3.
2. Receive event, denoted *sm*1_*pr*2, this indicates that the message #1 is received; it is a message from *pr*2.
3. Local event *n*1=4.
4. Send event, denoted s*m*2_*pr*1, this indicates that the message #2 is sent by *pr*1 to *pr*2.
5. Receive event, denoted *rm*3_*pr*2, this indicates that the message #3 is received; it is a message from *pr*2.
6. Local event *n*1=5.

Similarly, the second process *pr*2 is represented in the trace by the following event sequence:

1. Local event *n*2=6.
2. Send event *sm*1_*pr*2.
3. Local event *n*2=4.
4. Send event *sm*3_*pr*2.
5. Receive event *rm*2_*pr*1.
6. Local event *n*2=2.

The block diagram and the corresponding processes of the SDL model are shown in Figures 6, 7, and 8, respectively. Communication between the two processes is carried out through a two-way channel. Note that, in Figures 7 and 8, we model the Local events of each process using the TASK construct of SDL. On the other hand, we use the message names to represent the communication events in the SDL model.

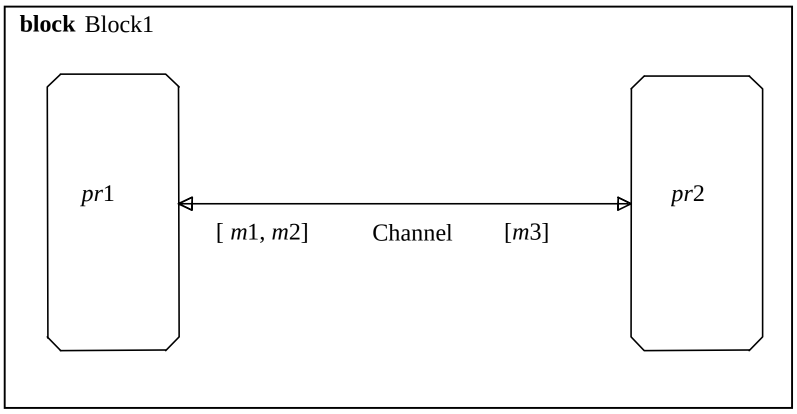


Figure 6: Block diagram of the system model in SDL.

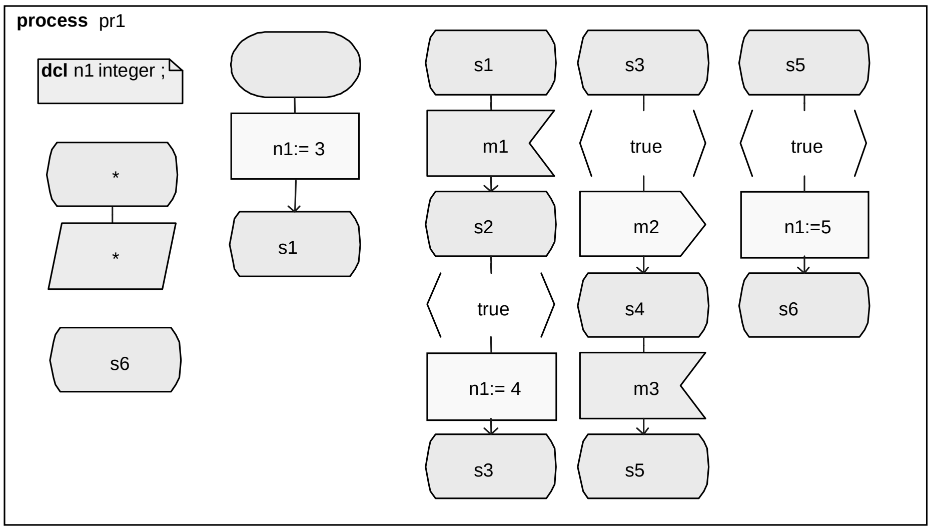


Figure 7: The SDL model for the first process.

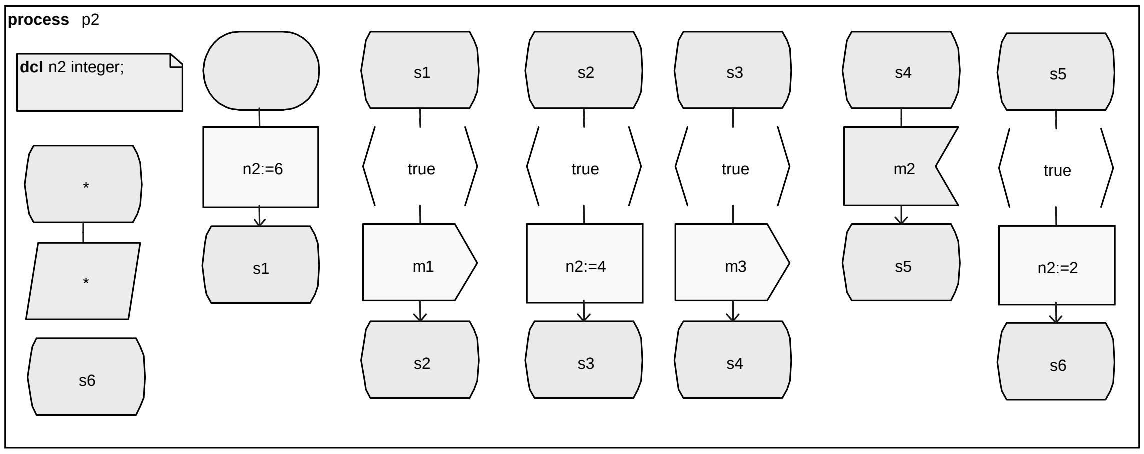


Figure 8: The SDL model for the second process.

The OG simulator returns the state graph with 27 states and 40 transitions described in textual representation. The lattice itself is shown in Figure 9.

The transitions are labeled with the names of the corresponding events. As for the states, they express local states of the processes and the corresponding values of the variables. Here, we only show the local variables of the processes. We use them to formulate the patterns we would like to recognize with the model checker and to demonstrate that they are eventually identified correctly. The sign "?" indicates the fact that the value of the corresponding variable is unknown in the corresponding state as there was no report from the process on the value of this variable yet. As a result, the detection of a pattern formulated in terms of both variables cannot be accurately performed in any of the five global states, where the value of either variable is still unknown. Note that the OG simulator assumes zero values for these variables by default.

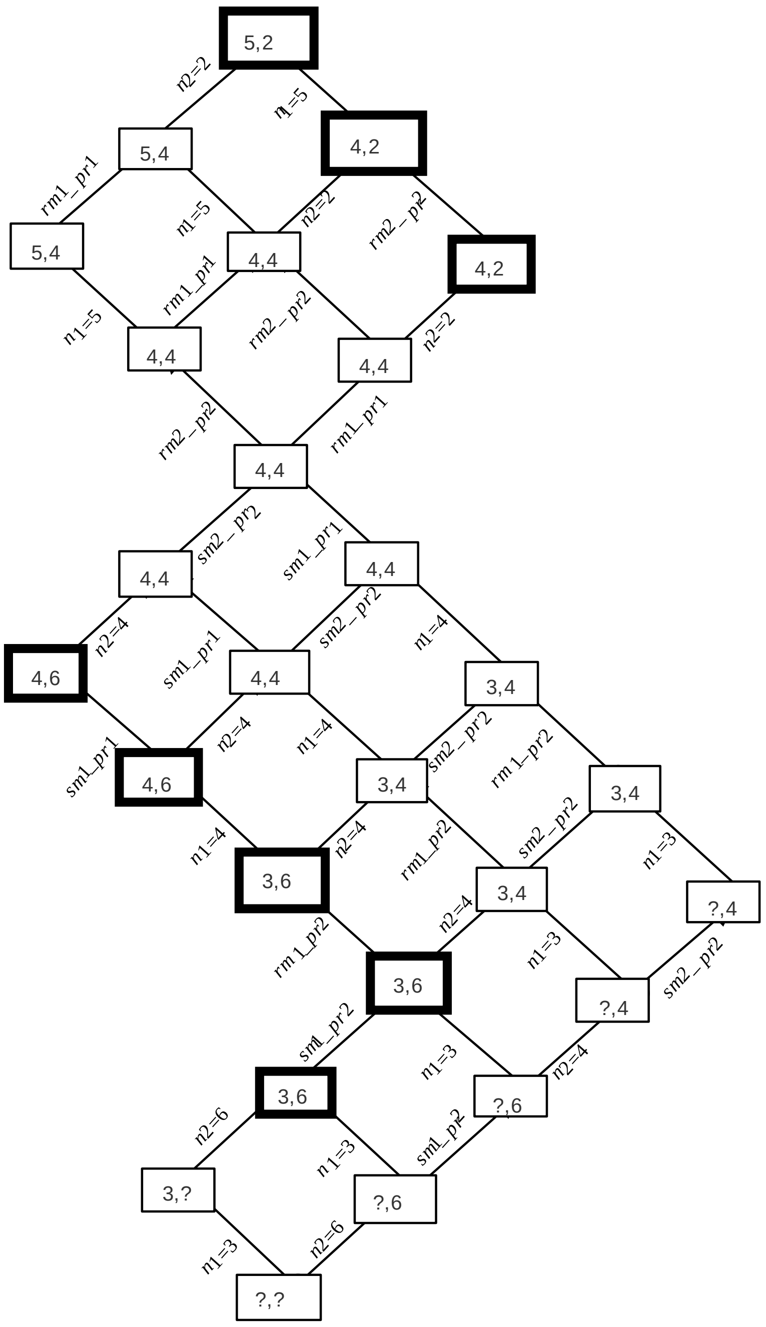
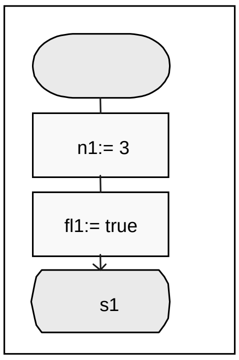
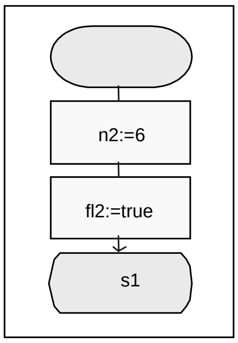


Figure 9: The lattice of the collected trace of events. (*sm*1_*pr*2 stands for send *m*1 by process *pr*2, and *rm*1_*pr*2 stands for receive *m*1 from process *pr*2, and local events are indicated by the variables and their values)

To fix this problem, we introduce two auxiliary Boolean variables in the SDL models of the processes that act as flags. After the first assignment on a variable (*n*1 or *n*2) is done, the corresponding flag is set. This amounts to modifying only the transitions that first affect the variables. The modified transitions of *pr*1 and *pr*2 are shown in Figure 10 a) and b), respectively. The settings of the flags are then used in order to initialize any observers that watch the values of the variables.



(a)



(b)

Figure 10: The modified transitions of *pr*1 (a) and *pr*2 (b).

For our example, we consider the following two patterns. The patterns use the values of the two variables, *n*1 and *n*2.

Pattern 1: It is possible that the variable *n*2 exceeds the variable *n*1 by 2 or more, i.e., $n2 \geq n1 + 2$, and later *n*1 exceeds *n*2 by 2 or more, i.e., $n1 \geq n2 + 2$.

Note that the pattern expresses a temporal property. It can be thought of as an instantiation of the existence pattern with scope "*after*". We check the occurrence of the state formula $P = (n1 \geq n2 + 2)$ after the state formula $Q = (n2 \geq n1 + 2)$. By direct inspection of the graph in Figure 9, one can check that this pattern is present. Clearly, there are five adjacent global states of the system (shown in bold frames), where the condition *Q* holds. From either state, the system can reach the other three states (also shown in bold frames) where *P* holds.

In order to illustrate a pattern that is not present in the given trace, we reverse the order of the two predicates in Pattern 1.

Pattern 2: It is possible that the variable *n*1 exceeds the variable *n*2 by 2 or more, i.e., $n1 \geq n2 + 2$, and later *n*2 exceeds *n*1 by 2 or more, i.e., $n2 \geq n1 + 2$.

Obviously, this pattern is not in the observed behavior of the system.

We build two GOAL observers that monitor the values of *n*1 and *n*2, and implement the patterns. The two observers for Pattern 1 and Pattern 2 are shown in Figure 11 on the left and right side, respectively. Note how the flags are used to initialize the observers. The observation of the values of the variables does not begin until the flags are set.

The exhaustive simulation of the SDL models together with the observers allows us to verify each property in the trace. The results of the exhaustive simulation (Figure 12) show that there are three scenarios that verify the existence of Pattern 1. On the other hand, Pattern 2 is not detected in the collected trace; no scenario leads to the success state of the observer. These results could be verified by inspecting Figure 9.

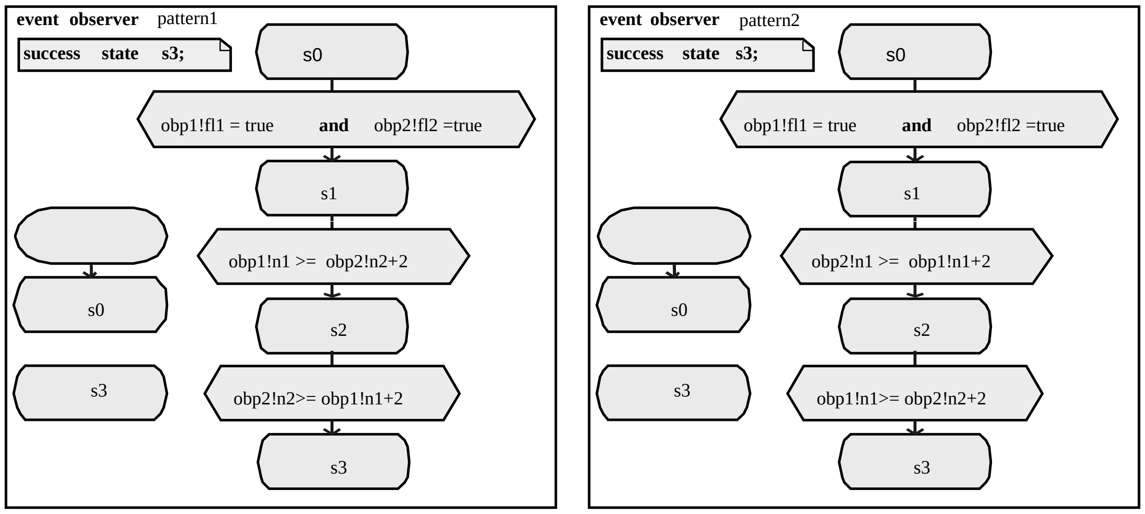


Figure 11: The observer for patterns 1 (left) and 2 (right).

```
Number of states : 44
Number of transitions : 64
Maximum depth reached : 13
duration : 0 mn 1 s
Number of exceptions : 0
Number of deadlocks : 4
Number of stop conditions : 0
Transitions coverage rate : 100.00 (0 transitions not covered)
States coverage rate : 100.00 (0 states not covered)
Basic  blocks  coverage  rate  :  100.00  (0  basic  blocks  not
covered)
Number of errors : 0
Number of success : 3
  observer ant1: 0 errors, 3 success
  observer ant2: 0 errors, 0 success
```

Figure 12: Verification results for the two patterns.

6 Conclusions

We proposed an approach to property verification of execution traces from distributed systems that is based on SDL and uses the commercial off-the-shelf model checker, the ObjectGEODE simulator. The approach assists in checking whether the distributed system satisfies certain properties and in pinning down errors to their origins.

The suggested approach takes into account the usual lack of formal design specifications in practice. It requires merely that the aspects of the distributed system the designer or tester is interested in be modeled as patterns. The approach relies on the availability of a tracing and monitoring tool that must be integrated with the distributed system to obtain an execution trace as the basis for analysis. It turns out that in practice this requirement is not hard to meet since developers usually implement their own tracing mechanisms to help them debug the system. Moreover, non-intrusive methods exist already for a number of operating systems that do not require changes of the original source code to collect trace data.

The validity of our approach depends on two main factors: correct matching of communication events and preservation of the local orders of events as stated in the collected traces. Concerning the first factor, we stated the required assumptions on the trace to guarantee correct matching. As for the second factor, we use the SDL special construct "SAVE" to preserve the local orders of events.

In addition, we have considered the problem of specifying the desired properties. For this purpose, we used a repository of specification patterns [9], and we demonstrated how typical patterns are mapped into GOAL observers. Moreover, an example has been used to illustrate the approach.

Finally, this ongoing work includes implementing the front-end tool that automates the process of extracting the SDL models and supports the specification of properties in GOAL. Currently, the finished parts of the tool allow us to build SDL models from execution traces of real systems.

Acknowledgements. Siemens AG, Germany, financially supports this research project. Fruitful discussions on observers and Goal language with R. Groz and D. Vincent are greatly appreciated.

References

[1] B. Algayres, Y. Lejeune, E. Hugonnet, "GOAL: Observing SDL behaviors with GEODE", *in SDL'95 with MSC in CASE (ed. R. Braek, A. Sarma), Proc. of the 7th SDL Forum*, Oslo, Norway, September, 1995, Elsevier Science Publishers B. V. (North Holland), pp. 359-372.

[2] J. P. Black, M. H. Coffin, D. J. Taylor, T. Kunz, A. A. Basten, "Linking Specifications, Abstraction, and Debugging", *CCNG Technical Report E-232, Computer Communications and Network Group*, University of Waterloo, November 1993.

[3] K. M. Chandy, L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems", *ACM Transactions on Computing Systems* 3(1), pp. 63-75, February 1985.

[4] S. Cheung, K. Levitt, "A Formal-Specification Based Approach for Protecting the Domain Name System", *In Proc. of the Workshop on Depend Despite Malicious Faults*, New York, June 2000.

[5] P. Dauphin, M. Kienow, A. Quick, "Model-Driven Validation of Parallel Programs Based on Event Traces", *In Proc. of Programming Environments for Parallel Computing*, Edinburgh, 1992.

[6] F. Dietrich, X. Logean, S. Koppenhoefer, J.-P. Hubaux, "Testing Temporal Logic Properties in Distributed Systems", *In Proc. of the 11th International Workshop on Testing of Communicating Systems*, Tomsk, Russia, August 1998.

[7] M. Dwyer, G. Avrunin, and J. Corbett, "Patterns in Property Specifications for Finite-state Verification", *In Proc. 21st International Conference on Software Engineering*, May 1999.

[8] M. Dwyer, L. Clarke, "Data Flow Analysis for Verifying Properties of Concurrent Programs", *In Proc. of ACM SIGSOFT'*94, New Orleans, LA, USA, 1994.

[9] E. Fromentin, M. Raynal, V. Garg, and A. Tomlinson, "On the Fly Testing of Regular Patterns in Distributed Computations", *Internal Publication* # 817, IRISA, Rennes, France, 1994.

[10] R. Groz, "Unrestricted Verification of Protocol Properties on a Simulation Using an Observer Approach", *Protocol Specification, Testing and Verification,* VI, Montréal, Canada, North-Holland, 1986, pp. 255-266.

[11] http://www.cis.ksu.edu/santos/spec-patterns.

[12] C. E. Jackl, "Event-Predicate Detection in the Debugging of Distributed Applications", *Master's Thesis*. Department of Computer Science, University of Waterloo, 1996.

[13] C. Jard, T. Jeron, G. V. Jourdan, and J. X. Rampon, "A General Approach to Trace-checking in Distributed Computing Systems", *In Proc. IEEE Int. Conf. on Distributed Computing Systems*, Poznan, Poland, June 1994.

[14] S. Kumar, E. Spafford, "An Application of Pattern Matching in Intrusion Detection", *Technical Report* 94-013, Purdue University, Department of Computer Sciences, March 1994.

[15] D. C. Luckham and B. Frasca, "Complex Event Processing in Distributed Systems", *Stanford University Technical Report* CSL-TR-98-754, March 1998, 28 pages.

[16] B. Miller, J. Choi, "Breakpoints and Halting in Distributed Programs", *In Proc. of the 8th IEEE Int. Conf. on Distributed Computing Systems*, San Jose, July 1988.

[17] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoaglan. K. Levitt, C. Wee, R. Yip, D. Zerkle, "GrIDS- A Graph Based Intrusion Detection System for Large Networks", *In Proc. of National Information Systems Security Conference*, Baltimore, MD, October 1996.

[18] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoaglan. K. Levitt, C. Wee, R. Yip, D. Zerkle, "The Design of GrIDS: A Graph-Based Intrusion Detection System", *Technical Report*, Department of Computer Science, University of California at Davis, January 1999.

[19] P. A. S. Ward, "A Framework Algorithm for Dynamic Centralized Dimension-Bounded Timestamps", *In Proc. of CASCON* 2000, Mississauga.

A formal approach to practical test selection

Tibor Csöndes*
Tibor.Csondes@eth.ericsson.se
Conformance Lab, Ericsson Hungary Ltd.,
1037 Budapest, Laborc u. 1., Hungary

Balázs Kotnyek
B.Kotnyek@lse.ac.uk
London School of Economics and Political Science

Abstract

This paper presents a formal approach to the practical test selection problem that arises in conformance test laboratories when the question is which test cases from a given test suite should be executed. The existing standard ([1]), which provides a formal framework to conformance testing, does not deal with this aspect, although it is an important part of the test laboratories' activity. Our goal is to work out a theoretical background to formal methods tackling the test selection problem. To this aim, we define the relevant notions unambiguously.

Keywords: Telecommunication Systems, Conformance Testing, Test Selection, Formal Approach

1 Introduction

The reduction of the time or effort put into conformance testing while keeping the test coverage under control is very important for those who perform conformance testing, usually in a test laboratory. If the time for conformance testing is limited, i.e., there is no time to execute all the test cases, then testers have to select the most efficient test cases from the whole test suite in order to make testing possible within a shorter period of time. A good selection has important economic advantages. Test laboratories can complete testing faster or they can achieve higher coverage in a given time. Therefore, a method which can help them in making the selection more efficient has considerable practical advantages.

Such a method should be efficient, it should make the test process faster by losing only a relatively small part of coverage. It also has to be applicable to different

*Corresponding author. Fax: +36 1 437 7767 types of real-life protocols as the laboratories have to test several implementations of different vendors. The method has to be flexible, meaning that special preferences of the test laboratories and their knowledge of the specific protocol can be incorporated. An important requirement of any kind of testing is that it has to be reproducible, that is, the selection can be repeated if it is needed and the same input data should result in the same selected test set, to ensure objective decisions.

To satisfy these requirements formal methods have to be used. But formal methods need formal notations. The main purpose of this paper is to present a formal description of the test selection problem by introducing a mathematical model. We first define the notions and relations relevant to test selection formally, then using these definitions we translate the test selection to two optimization problems.

The existing methods do not handle the problem from the practical point of view. Their aim is either to find efficient test generation methods (see e.g. [2]), the purpose of which is to produce an optimal test suite, or to provide a generic framework to test selection. The most elaborated method of this latter approach is due to Tretmans ([3]), which bases on the Labeled Transition System description of protocols. Another theoretical test selection method is the metric based selection ([4, 5, 6]), which uses execution sequences to describe the protocol's behaviour. Despite the unarguable theoretical advantages of these methods (including test generation), their main drawback is that they cannot easily be applied in real-life situations. This is mainly because they use formal notations that are not available (and cannot be expected that they will be available in the near future) for any real protocol.

Our approach, however, follows the practice by selecting test cases from the Abstract Test Suite (ATS) of a protocol. Besides the fact that it is the ATS that is used in test laboratories, another advantage of using it as a basis for selection is that it is the only available standardized form in conformance testing. To achieve minimal testing time for a given lower bound of coverage or maximal coverage for an upper bound of the cost, our method determines which test cases have to be selected for execution. The method is protocol independent, so it can be used in the testing of any protocol.

In what follows, we give a short introduction to conformance testing, listing the most important notions that are relevant to test selection. Then, in Section 3, we define these notions formally, and introduce a new concept, the subpurposes. Section 4 presents the mathematical model of test selection. In Section 5 we show how the theory can be put in practice.

2 Conformance testing

Conformance testing is an important step in the life-cycle of telecommunication protocols ([7]). Its purpose is to check whether the implemented protocol is conform to standards, and by that to increase the probability that different implementations are able to interwork. Conformance testing involves verifying that the external behaviour of the implementations comply with the requirements contained in their relevant protocol specifications ([8]).

As a background to our method, in this section we introduce the most important and relevant notions of conformance testing:

Conformance requirements During conformance testing, a protocol is tested to ensure that it meets the *conformance requirements* contained informally in the relevant protocol specification ([8]). Basically, a protocol specification is made up of the conformance requirements.

Test purposes, test cases The standard ([9]) defines a *Test Purpose* (TP) as "a prose description of a well defined objective of testing, focusing on a single conformance requirement or a set of related conformance requirements as specified in the appropriate OSI specification (e.g.: verifying the support of a specific value of a specific parameter)".

Standardization institutes collect the conformance requirements of a protocol specification and produce test purpose documents. These standardized documents contain several test purposes in a well-defined form. Different standardization institutes use different forms. For example, within ETSI this task is described in the "Test Purpose style guide" ([10]), guiding the document writers in preparing a test purpose document.

A test purpose is usually based on a single conformance requirement. In practice, however, test purposes may well concern a set of conformance requirements. The "Test Purpose style guide" lists the following cases when this is allowed: When the test case concerns an aspect specific to a profile, or when the size of the ATS demands a limitation in the number of test purposes ([10]).

Based on the test purpose document, *Test Cases* (TCs) are derived: one test case for each test purpose. This derivation can be done automatically or manually. The automatic test case derivation is based on test generation methods (see e.g.: [2]). The manual test case writing focuses on the protocol specification and the informally given conformance requirements collected in the test purpose document.

The Abstract Test Suite In order to get a reliable and repeatable verdict of conformance testing, a standardized test suite, the *Abstract Test Suite* (ATS) has to be used. The ATS contains a considerable number of test cases, which focus on the specific parts of the protocols' behaviour described in their purposes, and which can be executed independently from each other. The test suite is called abstract because it is independent from the implementation. This implies that an ATS is not directly executable, as it needs further parameterization and compilation.

Ideally, an ATS should be *complete*, that is, an implementation passes it if and only if the implementation is correct. Since it is generally not possible to construct a finite complete test suite, an ATS is required to be *sound,* meaning that all implementations that do not pass it are not correct. A sound test suite is complete if and only if it is also *exhaustive*; that is, if all passing implementations are conforming ([1]).

Tree and Tabular Combined Notation (TTCN) Test suites must be specified according to a test notation. A good test notation is well-defined, independent of the implementation, and is generally accepted. The standard ISO 9646 recommends a semi-formal language, the Tree and Tabular Combined Notation (TTCN).

TTCN can be given in two forms: a *graphical form,* denoted as TTCN.GR, and a *machine-processable form*, denoted as TTCN.MP. The graphical form is defined using tabular proformas, thus it is suitable for human reading or visual interpretation and it is easy to understand. TTCN.MP is suitable for transmission of TTCN descriptions between different machines and is possibly suitable for other automated processing. Consequently, TTCN is defined in such a way that its automatic execution is possible. In fact, the two different representations of an ATS in graphical and machine processable format are equivalent. Moreover, using a commercial TTCN editor, it is easy to convert a TTCN.GR to TTCN.MP and vice versa ([7, 9, 11]).

Test selection The methodology of conformance testing ([9]) defines test selection as a selection process of test cases from an Abstract Test Suite based on specific parameter values contained in the Protocol Implementation Conformance Statement (PICS) and the Protocol Implementation eXtra Information for Testing (PIXIT). The test laboratory selects the set of test cases the corresponding protocol behavior of which are actually implemented in the IUT. This is usually an automatic selection and keeps the mandatory capabilities, as well as the optional and conditional capabilities in view during the selection process ([7]).

In this paper, however, we will use the phrase test selection in a different context. In our terms, and in this we follow [3], test selection means selecting test cases from a given test suite in order to get a test set which is executable within limited resources. During the conformance testing process, the preparation for testing and the test operation is time intensive and expensive. For this reason, test laboratories usually have to omit some test cases and execute only certain ones ([12]). By this selection, the soundness of the test suite is kept, though its error-detecting capability may decrease.

Coverage The coverage ([1]) is a widely used, though not exactly defined, metric to measure the completeness of a set of test cases with respect to checking the conformance of a protocol. Generally speaking, the coverage of a test suite quantifies the percentage the protocol behaviour is 'covered'. In other words, the coverage is a measurement of the error-detecting capabilities of a test suite. The coverage can be used to compare test suites; high coverage expresses high quality test suite.

As the whole ATS is usually not complete, the requirements it tests may not cover the entire behaviour of the protocol. However, since the task is to select test cases of a given ATS, the coverage of a set of test cases can be viewed as its relative completeness with respect to the whole ATS. That is, it measures how much the requirements of the test suite are covered by the selected test cases. In other words, we can assume the coverage of the whole ATS to be 100%.

3 Formal description

In the previous sections we outlined conformance testing and introduced the test selection problem. We listed the requirements an efficient test selection method has to fulfill: it has to be efficient and reproducible. Furthermore, an efficient test selection method has to stand alone, as in that it has to involve as little human intervention as possible. These requirements need the automation of the test selection process. On the other hand, as we are going to apply a mathematical model to test selection, we have to formalize it.

In this section we give a formal description of the basic notations of conformance testing presented previously, and introduce a new notation: the subpurposes. They are mathematically well defined parts of a protocol which are automatically detectable in the ATS. Finally, we present how the data elements can be used in the test selection method.

3.1 Definition of subpurposes

Let us suppose that the standardized test suite $TS = \{t_1, t_2, \ldots, t_n\}$ consisting of test cases $t_1, t_2, \ldots, t_n$ and the set of corresponding test purposes $TP = \{p_1, p_2, \ldots, p_n\}$ are given. As we explained in Section 2, there is a one-to-one connection between the test cases and test purposes: p_i corresponds to t_i $(i = 1, 2, \ldots, n\ )$.

Let $REQ = \{r_1, r_2, \ldots, r_m\}$ denote the set of conformance requirements. As defined in Section 2, test purposes are related to a set of conformance requirements, and a test case is written to check the requirements involved in the corresponding test purpose. The following relation describes this connection between the test cases and the conformance requirements:

$$t_i \ \underline{check} \ r_j \overset{def}{\Leftrightarrow} t_i \text{ is able to check } r_j \tag{1}$$

Although test purposes describe conformance requirements, this description is prose and informal. To apply formal methods, we need a formal representation of test purposes. To this aim, let us introduce the *abstract test purpose* P_{t_i} for each test case t_i as a set of conformance requirements that the test case is able to check:

$$P_{t_i} \overset{def}{=} \{r \in REQ \mid t_i \, \underline{check} \, r\} \quad (i = 1, 2, \ldots, n)\,.$$

Note that in the ETSI terminology ([10]) "formal test purpose" is used to denote a special format of test purposes. This format is a structured description, but it is not suitable for automated processing. On the other hand, conformance requirements, which describe the basic features of the protocol behaviour that have to checked to ensure the conformance of the protocol, are not given in an appropriate, automatically processable or even formal format.

Hence, neither the test purposes nor the conformance requirements are available in a format that is suitable for automated test selection. That is why we introduce subpurposes.

Definition 1. The *subpurposes* are automatically detectable approximations of the conformance requirements.

Figure 1 illustrates the relationship between conformance requirements, test purposes and subpurposes. Test purposes are made up of conformance requirements, while the requirements can be associated with a set of subpurposes.

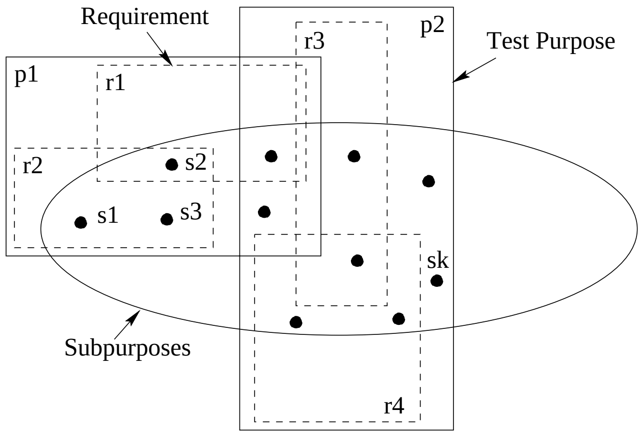


Figure 1: Requirements, test purposes and subpurposes

Conformance requirements provide a subdivision of the protocol behaviour: every relevant aspect of the protocol is described with one or more requirements. Subpurposes provide another subdivision. As both the entire set of conformance requirements and the subpurposes cover the protocol behaviour, the building blocks of the original subdivision (the conformance requirements) can be described by the building blocks of the second subdivision (the subpurposes).

To formulate this relationship, let $SP = \{s_1, s_2, \ldots, s_k\}$ be the set of subpurposes and for each requirement we associate a subset of SP:

$$r_j \longmapsto SP_{r_j} \subseteq SP \text{ for } j = 1, 2, \ldots, m \tag{2}$$

We can extend relation $\underline{check}$ to subpurposes by defining the set of subpurposes that are checked by test case t_i $(i = 1, 2, \ldots, n\ )$:

$$\begin{gathered} t_i \longmapsto S_{t_i} \subseteq SP \text{ where } S_{t_i} \overset{def}{=} \{s \in SP \mid \exists\, r \in REQ,\, r \in P_{t_i},\, s \in SP_r\} \\ t_i\ \underline{check}\ s_j \overset{def}{\Leftrightarrow} s_j \in S_{t_i} \end{gathered} \tag{3}$$

So far we have described the theoretical way of introducing subpurposes and defining sets of S_{t_i}. In practice, the requirements and therefore the sets P_{t_i} are not given, so we have to define the sets S_{t_i} directly and skipping the association described in (2). We discuss this method in the next section.

As a summary, we classified the most important notions of test selection with respect to their properties (Table 1). Test cases and test purposes are available in standardized test documents, while subpurposes and requirements are usually not. The difference between requirements (given or not given) and subpurposes is that subpurposes are automatically detectable by definition.

Table 1: The classification of the most important notions of test selection

	Automatically detectable	Not automatically detectable
Given	test case	test purpose
Not given	subpurpose	requirement

3.2 Determination of subpurposes

In the previous section we introduced subpurposes as automatically detectable parts of the protocol behaviour which can approximate the conformance requirements. In this section we present a possible way to determine subpurposes.

The conformance requirements have four components ([13]):

- Messages and their parameters
- Time periods
- States
- Protocol mechanisms

Furthermore, according to the standard ([9]), OSI protocol specifications define dynamic conformance requirements in terms of Protocol Data Units (PDUs) and Abstract Service Primitives (ASPs), that is, data elements.

On the other hand, messages and their parameters from the list above, in other words the data, play an increasingly important role in recent protocols. Take for example the modern WWW protocols (HTTP1.1), or the latest internet protocol (IPv6).

For these reasons, we make the data elements equivalent with the subpurposes. That is, we relate the purpose of a test case to the set of data elements sent or received in the test case. Of course, this way we get only an approximation of the real purpose of the test case but as experimental results show, this approach can be justified.

On the other hand, data elements used in a test case can be detected automatically with the help of the TTCN machine processable format of an ATS.

To make it formal, let $DATA = \{d_1, d_2, \ldots, d_q\}$ be the data elements used by the protocol. To every test case t_i we assign the protocol data elements sent or received in t_i:

$$D_{t_i} \overset{def}{=} \{d \in DATA \mid d \text{ used in } t_i\}$$

Now, as mentioned above, let us map the data elements onto subpurposes, i.e., $SP := DATA$ by identifying the set of subpurposes checked by a test case with the set of data elements used in it:

$$S_{t_i} := D_{t_i}, \quad (i = 1, 2, \ldots, n) \tag{4}$$

3.3 Abstract data levels

Data elements are not well-defined notions in conformance testing, for different reasons. First, because protocols are different with respect to data content. There are protocols where the data have much more importance than the behaviour control part, while in other protocols there is more emphasis on the control. In this latter case, the variety of data is usually poor and its characterizing part lies 'deeper', for example, in the parameters of the messages.

Secondly, different standardization institutes and test suite writers issue ATSs written in different ways. One prefers to use few, highly parameterized ASPs and PDUs, while the other employs more ASP and PDU constraints.

To get a generic view of data elements, let us now examine what is common in the data of every ATS. Evidently, there is always a hierarchy in data elements. There are data elements on higher or lower level. Thus, let us define the *abstract data levels* as a generic frame to describe this hierarchy. A simple possible set of abstract data levels is the following:

1. level of *data type*
2. level of *data*
3. level of *data parameter*

This model is the simplest possible one, as further levels, such as the level of the parameters' parameters can be included, depending on the structure of the data.

To determine the subpurposes, we have to decide which level is the most important in the ATS. The level which expresses most of the protocol behaviour will play the role of subpurposes. This decision is made intuitively, based on the protocol and the way the ATS was written.

If the ATS is written in TTCN, the data levels are the following:

1. ASP and/or PDU type level
2. ASP and/or PDU constraint level
3. parameters of ASPs and/or PDUs: simple types, structured types, etc...

In 2000, ETSI standardized the new version of TTCN (TTCN version 3 [14]). The TTCN-3 does not distinguish between ASP and PDU types, so it is necessary to change the above structure:

1. message type level
2. template level
3. parameters of template level

Naturally, by the abstract data levels this hierarchy can be applied to any other testing language.

4 Mathematical model for test selection

In the previous section we gave formal definition of the relations relevant to test selection. Now we show that using this formal description, a mathematical model of the test selection problem can be developed. In that we will use the theory of mathematical programming by introducing two optimization problems. We also formally define the cost and the coverage of a test set.

We partly presented these ideas in [15], [17] and [18]. Here we show how the formal description of the previous section helps in unambiguous definitions.

4.1 Problem formulation

Let us recall that $TS = \{t_1, t_2, \ldots, t_n\}$ and $SP = \{s_1, s_2, \ldots, s_k\}$ denote the set of test cases and subpurposes related to a test suite. We define a positive *cost* function $c : TS \to \mathbb{R}_+$ measuring the amount of resources the execution of a test case requires. The relative importance of the subpurposes with respect to the correct behaviour of the protocol is represented by the *weight* function $w : SP \to \mathbb{R}_+$. In Section 5.2 we will discuss how these functions can be determined.

In (3) we defined S_{t_i}, the subset of subpurposes test case t_i is able to check. From another point of view, a subset of test cases, T_{s_i} is associated with each subpurpose s_i $(i = 1, 2, \ldots, k)$ containing the test cases which are able to check the subpurpose:

$$T_{s_i} \overset{def}{=} \{t \in T_s \mid t \, \underline{check} \, s_i\} \quad (i = 1, 2, \ldots, n) \,.$$

We will call these tests *related to the subpurpose.* For the sake of simpler notations, let b_i denote the number of test cases related to subpurpose s_i, namely let $b_i = |T_{s_i}|$ for each subpurpose.

Following the definitions presented above, we can define the **subpurpose-test case incidence matrix** which we will denote as A. Subpurposes $(s_1, s_2, \ldots, s_k)$ are placed in the rows, test cases $(t_1, t_2, \ldots, t_n)$ are placed in the columns. Matrix A has the following entries:

$$A\left(s_i, t_j\right) = \begin{cases} 1 & \text{if } t_j \, \underline{check} \, s_i \\ 0 & \text{otherwise} \end{cases}$$

Let T be an arbitrary set of the test cases, $T \subseteq TS$. The cost of this test set is defined as the sum of the costs of the test cases belonging to T:

$$c(T) = \sum_{t \in T} c(t) \tag{5}$$

Let $cov(T)$ denote the coverage of test set T, that is $cov(T)$ measures how large part of the protocol is tested by T (below, in (8), we give a formal definition as well).

Two optimization problems can be defined according to our purposes and limitations.

Minimal cost problem: Given a lower bound (K) for the coverage. Find the set of test cases that satisfies this bound with minimal cost.

$$\begin{array}{rl} \min & c(T) \\ \text{subject to} & cov(T) \geq K \\ & T \subseteq TS \end{array} \tag{6}$$

Maximal coverage problem: Given an upper bound (L) for the cost. Find the subset of test cases the cost of which is not more than L and checks as large part of the protocol (i.e. has as big coverage) as possible.

$$\begin{array}{rl} \max & cov(T) \\ \text{subject to} & c(T) \leq L \\ & T \subseteq TS \end{array} \tag{7}$$

Subpurposes represent separate parts of the protocol behaviour and their importance is denoted by their weights. This implies that the coverage of a test set can be expressed as the weighted sum of the individual coverages of the subpurposes.

$$cov(T) = \sum_{s_i \in SP} w(s_i) \cdot purpcov(s_i, T) \tag{8}$$

where $purpcov(s_i, T)$ measures in what proportion T covers the basic conformance feature represented by subpurpose s_i. Naturally, $0 \le purpcov(s_i, T) \le 1$ for all $s_i \in SP$ and $T \subseteq TS$. It is assumed that $purpcov(s_i, T)$ is a function of the number of test cases that are related to s_i and belong to test set T:

$$purpcov(s_i, T) = f_i(|T_{s_i} \cap T|) \quad (i = 1, 2, \ldots, k)$$

for some $f_i : \{0, 1, \ldots, |T_{s_i}|\} \to [0, 1]$. The exact values of these functions depend on the methodology the construction of the test suite followed. We presented possible alternatives in [15], where we called them *coverage models* as they determine the coverage of a test set. We will use this name in this paper too.

5 Test selection process

In this section we show how the theory can be turned into practice. We give a possible approach to the test selection process using the previously presented mathematical apparatus.

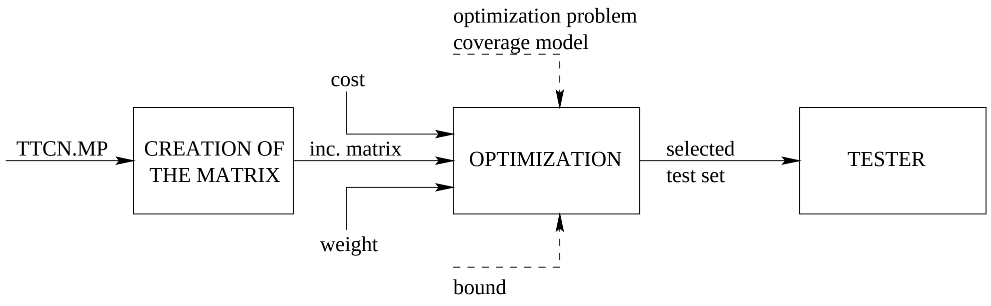


Figure 2: The process of selection

Figure 2 illustrates the main phases of the test selection process. Its steps are:

1. Creating the subpurpose-test case incidence matrix manually or automatically.
2. Determining the cost and weight functions (c and w), deciding which optimization problem (minimal cost or maximal coverage) and coverage model will be used, inputting the cost or coverage bound (L or K), and choosing a suitable optimization algorithm.

3. Loading the selected test cases into the tester.

For a given ATS of a protocol, the first step (i.e. the creation of the incidence matrix) has to be done only once. This step can be the most time consuming of the test selection process and it is where the most human interface is necessary. (If done with the manual matrix filling method, described later.) The fact that the same incidence matrix can be used for subsequent selections means that the efficiency of test selection can be strongly improved once the incidence matrix is available from a previous session.

Furthermore, if the incidence matrix is available, we can try to carry out the optimization for different cost, weight functions, optimization problems or different coverage models and bounds in order to find the best test set that meet our aims.

5.1 Creation of the subpurpose-test case incidence matrix

In the test selection process, the most crucial step is the creation of the subpurpose-test case incidence matrix. The incidence matrix contains the most crucial information about the protocol, that is why it is so important to fill this matrix precisely. On the other hand, this matrix filling can be done only once for a protocol, and it is possible to use it in any optimization method later as many times as needed.

5.1.1 Manual matrix filling

There are protocols (see e.g.: [16]) in which conformance requirements are given in the ATS standard accompanied by the test cases which can be used to check them. In this case we do not need to deal with deriving subpurposes because we can easily construct a matrix which describe the connection between the requirements and test cases.

In other cases, when the requirements are not given explicitly, they have to be determined, or approximated by subpurposes. If the ATS is not given in a machine processable form, (i.e.: in TTCN.MP) the incidence matrix can be filled only manually. In this case we cannot speak about subpurposes as they have to be automatically detectable by definition. What we do instead is finding the conformance requirements implicitly given in the protocol description and the test cases and fill the requirement-test case incidence matrix by examining which test case checks which requirement. This is a difficult task and requires deep knowledge of the chosen protocol and test description, as well as considerable time for real-life protocols. On the upside, the matrix filled by an expert is the one which is the closest to the theoretical conformance requirement-test case incidence matrix.

An example when manual matrix filling is needed is *function testing*, a test methodology widely used in e.g. Ericsson. This test method uses neither the conformance testing methodology nor the TTCN. For this reason it is impossible to create the incidence matrix automatically based on TTCN.MP and the manual matrix filling has to be used.

5.1.2 Automatic matrix filling

When a machine processable form of the ATS (usually TTCN.MP) is given, then the cumbersome manual matrix filling can be avoided by using subpurposes. Subpurposes are defined as automatically detectable parts of the protocol's behaviour (see Section 3.1), so automatic methods can be used to fill the subpurpose-test case incidence matrix.

A good subpurpose definition result in closer approximation of the theoretical requirement-test case incidence matrix, so one has to pay due attention to this step. On the other hand, the determination of the subpurposes needs only a good general view of the structure of the ATS and the behaviour of the protocol. It does not require detailed examination and comparison of test cases as in the case of the manual matrix filling where the conformance requirements and their relation to the test cases have to be fully detected.

We have to decide first what segment of the protocol is going to play the role of subpurposes, in other words, following Section 3.2, we have to choose an abstract data level (see Section 3.3) which will represent the subpurposes. Although PDUs are the most natural candidates for subpurposes, sometimes it is worth stepping one layer up or down, and choosing the PDU types or the parameters of the PDUs respectively to be subpurposes, if it results in more efficient selection. The most appropriate choice of subpurposes depends on the structure of the ATS.

We implemented a PERL program that is able to find the PDUs used in the test cases and fill the incidence matrix independently of the size of the ATS. This program can also detect PDUs that are used in the test steps or defaults appearing in the test case. So the PDUs related to a test case are sent either in its main body or in a test step or default used in the test case. By this the dependencies of he test cases are handled. For example, when a test case appears as the preamble of another test case, the incidence matrix will reflect this fact, consequently this dependency is taken into account in the optimization. The program also deletes the all zero rows if there are any (i.e.: eliminate the PDUs not used in the ATS), and merge equal rows, because in this case the PDUs related to these rows can be together viewed as one subpurpose.

5.2 Creation of cost and weight vectors

Determining the costs of the tests and the weights of the subpurposes is up to the test laboratories' individual preferences, but we think that choosing them to be the same for each test case and subpurpose, respectively is the most natural solution. As for the coverage, this is because at first sight there is no theoretical basis to distinguish between the subpurposes. The costs can be chosen to be the same because the most time consuming task during the conformance assessment process is preparation and parameterization, the time of which is in direct proportion to the number of the executed tests. Therefore, in most cases all-one vectors can be used as cost and weight vectors. In our experiments we also used randomly generated vectors as well as costs and weights based on the estimation of experts.

5.3 Optimization

Having the subpurpose-test case incidence matrix, the cost and weight vectors at hand we have to decide on the optimization problem, coverage model and upper or lower bound. Then the optimization problem should be solved to achieve an optimal test set which can be executed. There are several possible methods to find optimal or very good quality suboptimal solutions of a mathematical programming problem.

We can transform them to an Integer Linear Programming (ILP) problem and solve with efficient optimization algorithms (e.g.: Branch and Bound) using commercial ILP solving softwares. We published this way in [17]. Another possibility is to directly solve the optimization problems with heuristic algorithms, as presented in [18]. Anyway, the result of the optimization is a test set which is the best possible subject to the constraint, and which, because it was selected from the ATS, can be directly loaded into the tester.

6 Conclusion

We have introduced the test selection problem arising in the practice of conformance test laboratories, where the task is to select test cases for execution from the Abstract Test Suite. We have given a formal description of the problem, on the basis of which a mathematical model of test selection can be given. We have presented a possible approach which translates test selection to mathematical programming problems. We have also shown how this model fits in the practice of conformance testing, making it more efficient.

In particular, we defined subpurposes, an automatically detectable approximation of the conformance requirements and the abstract data levels, which help in finding good subpurposes in a protocol.

The presented approach, possibly with slight modifications, can be applied to other kinds of testing, e.g. function or regression testing.

References

[1] ITU-T Recommendation Z.500, Framework on Formal Methods in Conformance Testing, 1997.

[2] X. Sun, Y. Shen, C. Feng and F. Lombardi, *Protocol Conformance Testing Using Unique Input/Output Sequences*, World Scientific, Singapore, 1997.

[3] J. Tretmans, *A Formal Approach to Conformance Testing*, PhD Thesis, University of Twente, The Netherlands, 1992.

[4] S.T. Vuong and J. Alilovic-Curgus, On Test Coverage Metrics for Communication Protocols in J. Kroon, R.J. Heijink, E. Brinksma (Eds.), *Protocol Test Systems*, IV, Elsevier, 31-45, 1992.

[5] S.T. Vuong, J. Zhu and J. Alilovic-Curgus, Sensitivity Analysis of the Metric Based Test Selection in M. Kim, S. Kang, K. Hong (Eds.) *Testing of Communicating Systems*, Vol. 10, Chapman & Hall, 1997.

[6] J. Zhu and S.T. Vuong, Generalized Metric Based Test Selection and Coverage Measure for Communication Protocols in T. Mizuno, N. Shiratori, T. Higashino, A. Togashi (Eds.), *Formal Description Techniques and Protocol Specification, Testing and Verification*, Chapman & Hall, 299-314, 1997.

[7] B. Baumgarten and A. Giessler, *OSI Conformance Testing Methodology and TTCN*, Elsevier, Amsterdam, 1994.

[8] M. Bush, K. Rasmussen and F. Wong, Conformance Testing Methodologies for OSI Protocols, *AT&T Technical Journal*, 84-100, January/February 1990.

[9] ITU-T Recommendation X.290-X.296 - ISO/IEC 9646, Information Technology - Open Systems Interconnection - Conformance Testing Methodology and Framework, 1994.

[10] ETSI ETR 266; Methods for Testing and Specification (MTS); Test Purpose style guide, 1996.

[11] R.L. Probert and O. Monkewich, TTCN: The International Notation for Specifying Tests of Communications Systems, *Computer Networks and ISDN Systems* **23**, 417-438, 1992.

[12] R.J. Heijink, FAITH, a General Purpose Test System for ISDN, *Computer Networks and ISDN Systems* **26**, 1581-1593, 1994.

[13] K. Tarnay, *Protocol Specification and Testing*, Akadémiai Kiadó, Budapest, 1991.

[14] ETSI DES/MTS-00063-1 v1.0.10; Methods for Testing and Specification (MTS); The Tree and Tabular Combined Notation version 3; TTCN-3: Core Language, 2000.

[15] T. Csöndes, B. Kotnyek, Automated Test Case Selection Based on Subpurposes, in Gy. Csopaki, S. Dibuz, K. Tarnay (Eds.) *Testing of Communicating Systems, Methods and Applications,* Kluwer Academic Publishers, 251-265, IFIP TC6 12th International Workshop on Testing of Communicating Systems (IWTCS'99), Budapest, Hungary, September 1-3, 1999.

[16] ETSI Draft prTBR4 Integrated Services Digital Network (ISDN); Attachment requirements for terminal equipment to connect to an ISDN using ISDN primary rate access, 1994.

[17] T. Csöndes, S. Dibuz, B. Kotnyek, Test Suite Reduction in Conformance Testing, *Acta Cybernetica,* Vol. 14 No. 2, 229-238, 1999.

[18] B. Kotnyek, T. Csöndes, Heuristic Methods for Conformance Test Selection, *European Journal of Operational Research,* submitted, August 2000.

Recent BRICS Notes Series Publications