Burkhart Wolff
Fatiha Zaïdi (Eds.)

LNCS 7019

Testing Software and Systems

23rd IFIP WG 6.1 International Conference, ICTSS 2011
Paris, France, November 2011
Proceedings

ifip

Springer

Burkhart Wolff Fatiha Zaïdi (Eds.)

Testing Software and Systems

23rd IFIP WG 6.1 International Conference, ICTSS 2011
Paris, France, November 7-10, 2011
Proceedings

Springer

Volume Editors

Burkhart Wolff
Fatiha Zaïdi
Université Paris-Sud
LRI UMR 8623 CNRS
91405 Orsay Cedex, France
E-mail: {burkhart.wolff, fatiha.zaidi}@lri.fr

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-24579-4 e-ISBN 978-3-642-24580-0
DOI 10.1007/978-3-642-24580-0
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011937951

CR Subject Classification (1998): D.2, D.2.4, F.3, D.1, D.3, F.4.1, K.6

LNCS Sublibrary: SL 2 – Programming and Software Engineering

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Testing has steadily become more and more important within the development of software and systems, motivating an increasing amount of research aimed at trying to solve both new challenges imposed by the advancement in various areas of computer science and long-standing problems. Testing has evolved during the last decades from an ad-hoc and under-exposed area of systems development to an important and active research area.

The 23rd International Conference on Testing Software and Systems (ICTSS) involved the merger of two traditional and important events which have served the testing community as important venues for discussing advancements in the area. Those events, namely, TestCom (the IFIP TC 6/WG 6.1 International Conference on Testing of Communicating Systems), and FATES (International Workshop on Formal Approaches to Testing of Software), together form a large event on testing, validation, and specification of software and systems. They have a long history. TestCom is an IFIP-sponsored series of international conferences, previously also called International Workshop on Protocol Test Systems (IWPTS) or International Workshop on Testing of Communicating Systems (IWTCS). It is devoted to testing communicating systems, including testing of communication protocols, services, distributed platforms, and middleware. The previous events were held in Vancouver, Canada (1988); Berlin, Germany (1989); McLean, USA (1990); Leidschendam, The Netherlands (1991); Montreal, Canada (1992); Pau, France (1993); Tokyo, Japan (1994); Evry, France (1995); Darmstadt, Germany (1996); Cheju Island, South Korea (1997); Tomsk, Russia (1998); Budapest, Hungary (1999); Ottawa, Canada (2000); Berlin, Germany (2002); Sophia Antipolis, France (2003); Oxford, UK (2004); Montreal, Canada (2005); and New York, USA (2006). Fates, Formal Approaches to Testing of Software, is a series of workshops devoted to the use of formal methods in software testing. Previous events were held in Aalborg, Denmark (2001); Brno, Czech Republic (2002); Montreal, Canada (2003); Linz, Austria (2004); Edinburgh, UK (2005); and Seattle, USA (2006). From 2007 on, TestCom and Fates have been jointly held in Tallinn, Estonia (2007), Tokyo, Japan (2008), and Eindhoven, The Netherlands (2009).

This book constitutes the refereed proceedings of the 23rd IFIP International Conference on Testing Software and Systems (ICTSS 2011) held on November 7–10, 2011 in Paris. There were 40 submissions. Each submission was reviewed by at least 3, and on the average 3.6, Program Committee members. After intensive discussions, the committee decided to accept 13 papers, among them 2 papers from the industrial track. The program also included 2 invited talks: Marie-Claude Gaudel with an overview of random-based test generation algorithms and the latest applications to LTL model-checking, and Manuel Núñez on problems in testing combinations of timed and probabilistic finite state machine formalisms. The conference was preceded by a tutorial day consisting of presentations of industrial tools relevant for the field; the speakers were Bruno Legeard, Jean-Pierre Schoch, and Nikolai Tillmann.

We would like to thank the numerous people who contributed to the success of ICTSS 2011: the Steering Committee, the Program Committee, and the additional reviewers for their support in selecting papers and composing the conference program, and the authors and the invited speakers for their contributions without which, of course, these proceedings would not exist. The conference would not have been possible without the contributions and the support of the following organizations: Microsoft Research, the Institut Henri Poincaré (that generously offered us the facilities for the conference location), INRIA, the LRI of Université de Paris-Sud, the Digiteo Foundation, and the CNRS Scientific Research Group GDR GPL. Moreover, all members of the ForTesSE team helped with the local arrangements of the conference.

August 2011

Burkhart Wolff
Fatiha Zaïdi

Organization

Program Committee

Nina Yevtushenko	Tomsk State University
Fatiha Zaïdi	University of Paris-Sud, LRI

Additional Reviewers

Akhin, Marat
Bessayah, Fayçal
Bogdanov, Kirill
Bouchy, Florent
Du Bousquet, Lydie
Dury, Arnaud
Gromov, Maxim
Herbold, Steffen
Hwang, Iksoon
Lalanne, Felipe
Makedonski, Philip
Mouttappa, Pramila
Shabaldin, Andrey
Shabaldina, Natalia
Streader, David
Taylor, Ramsay
Weise, Carsten
Werner, Edith
Xiao, Xusheng

Table of Contents

Counting for Random Testing

Marie-Claude Gaudel

Université Paris-Sud 11, LRI, Orsay, F-91405,
and CNRS, Orsay, F-91405
mcg@lri.fr
http://www.lri.fr/~mcg

Abstract. The seminal works of Wilf and Nijenhuis in the late 70s have led to efficient algorithms for counting and generating uniformly at random a variety of combinatorial structures. In 1994, Flajolet, Zimmermann and Van Cutsem have widely generalised and systematised the approach. This paper presents several applications of these powerful results to software random testing, and random model exploration.

Keywords: Software testing, random walks, combinatorics.

1 Introduction

In the area of analytic combinatorics, the seminal works of Wilf and Nijenhuis in the late 70s have led to efficient algorithms for counting and generating uniformly at random a variety of combinatorial structures [13,19]. In 1994, Flajolet, Zimmermann and Van Cutsem have widely generalised and systematised the approach [7]. The recent book by Flajolet and Sedgewick [8] presents a detailed survey of this corpus of knowledge. These works constitute the basis of powerful tools for uniform random generation of complex entities, such as graphs, trees, words, paths, etc.

This paper summarises several applications of these powerful results to software random testing, and random model exploration.

Random methods look attractive for testing large programs or checking large models. However, designing random methods that have a good and assessable fault detection power is not so easy: the underlying probability distribution of inputs or paths must be carefully designed if one wants to ensure a good coverage of the program or model, or of potential fault locations, and to quantify this coverage.

This paper is organised as follows: Section 2 recalls some basic facts on software random testing and random walks; Section 3 briefly presents methods for uniform generation of bounded paths that are completely described in [17] and [5]; Section 4 shows how to take into account other coverage criteria, and gives a definition of randomised coverage satisfaction [5] ; finally Section 5 addresses the uniform coverage of lassos, which are the basis of LTL model-checking [18].

2 Some Preliminaries on Random Testing and Random Walks

There are three main categories of methods for software random testing: those based on the input domain of the system under test, those based on some knowledge of its environment, and those based on some model of its behaviour.

We focus on the third case, where some graphical description of the behaviour of the system under test is used. Random walks [1] are performed on the set of paths of this description. Classical random walk methods, sometimes called isotropic, progress from one state by drawing among its successors uniformly at random. The big advantage of such methods is that they are easy to implement and only requires local knowledge of the model. A serious drawback is that in case of irregular topology of the underlying graph, uniform choice of the next state is far from being optimal from a coverage point of view: some examples are given in [4] and [5]. Similarly, getting an estimation of the coverage obtained after one or several random walks would require some complex global analysis of the topology of the model.

The works presented in this paper aim at improving the quality of random walks with respect to various coverage criteria: bounded paths coverage, transitions/branches coverage, states/statements coverage, lassos. There is a price to pay: some non-local knowledge of the models is required, based on counting the elements to be covered accessible from each successor of the current state. Thank to the powerful results mentioned above, and to sophisticated implementation methods, it is possible to get good compromises between memory requirement, efficiency of drawing, and quality of the achieved coverage.

All the works below rely on combinatorial algorithms, based on the formalisation of models or programs as some automaton or as some product of several automata, synchronised or not. The basic algorithms are implemented and available in the RUKIA C++ library (http://rukia.lri.fr/en/index.html).

3 Improvements of Recursive Uniform Path Generation

3.1 The Classical Recursive Method

This classical method was first presented in [19,7]. Let us consider a deterministic finite automaton $\mathcal{A}$ with q states $\{1, 2, \ldots, q\}$ among which are distinguished an initial state and some final states. For each state s, let $l_s(n)$ be the number of paths of length n starting from s and ending at a terminal state. Such values can be computed with the following recurrences on n (where $\mathcal{F}$ denotes the set of final states in $\mathcal{A}$):

$$\begin{cases} l_s(0) = 1 & \text{if } s \in \mathcal{F} \\ l_s(0) = 0 & \text{if } s \notin \mathcal{F} \\ l_s(i) = \sum\limits_{s \to s'} l_{s'}(i-1) & \forall i > 0 \end{cases} \quad (1)$$

We note $L_n = \langle l_1(n), l_2(n), \ldots, l_q(n)\rangle$. The principle of the recursive method is in two steps:

– Compute and store vector L_k for all $1 \leq k \leq n$. This calculation is done starting from L_0 and using equations (1).
– Generate a path of length n by choosing, when the current state is s and the path has already $n-m$ states, successor s_i with the probability:

$$\mathbb{P}(s_i) = \frac{l_{s_i}(m-1)}{l_s(m)}. \tag{2}$$

When using floating point arithmetic as in [6], the complexity of the algorithm is in $\mathcal{O}(qn)$ space and time for the preprocessing stage and $\mathcal{O}(n)$ for the generation, where n denotes the length of the path to be generated, and q denotes the number of states.

For big models and long paths, this method does not scale up well. This was the motivation for two pieces of work presented below.

3.2 A Dichotomic Algorithm for the Uniform Random Generation of Paths

In [17] Oudinet et al. have presented the so-called *dichopile* method, which is based on a divide-and-conquer approach, avoids numerical instability and offers an excellent compromise in terms of space and time requirements.

Note that to choose the successor of the initial state, only L_n and L_{n-1} are needed. Then, L_{n-1} and L_{n-2} allow to choose the next state and so on. Thus, if we had a method that compute efficiently $L_n, L_{n-1}, \ldots, L_0$ in descending order, we could store the two last vectors only and reduce space complexity. This *inverse* approach constitutes the principle of Goldwurm's method [10]. However, in [16], Oudinet showed that this method is numerically instable, thus forbidding the use of floating-point arithmetics, which is essential given the big numbers involved.

The idea of the *dichopile* algorithm is as follows. Compute the number of paths of length n from the number of paths of length 0 while saving in a stack a logarithmic number of intermediate steps: the number of paths of length $n/2$, of length $3n/4$, of length $7n/8$, etc. For computing the number of paths of length $n-i$, it is computed again from the intermediate stage that is at the top of the stack. Recall that L_j denotes the vector of q numbers of paths of length j, that is the $l_s(j)$'s for all states s.

Unlike the classical recursive method, there is no preprocessing phase. In [17] it is proved that using floating-point numbers with a mantissa of size $\mathcal{O}(\log n)$, bit complexities of drawing are $\mathcal{O}(q \log^2 n)$ in space and $\mathcal{O}(dqn \log^2 n)$ in time, where d stands for the maximal out-degree of the automaton.

The classical recursive method is much faster after the preprocessing stage, but it is unusable for long paths and large models due to its space requirement. *dichopile* is an excellent compromise when considering both space and time complexities. In our experiments with automata from the VLTS benchmark suite (Very Large Transition Systems, http://tinyurl.com/yuroxx), examples of limits for the recursive method were 8879 states, 24411 transitions and paths of length 64000, or 10849 states 56156 transitions and paths of length 16000, where dichopile was able to generate paths of length 128000 and more. *dichopile* was able to deal with 12323703 states, 27667803 transitions and paths of length 8000. Both methods are implemented in the RUKIA library.

3.3 Uniform Path Exploration in Very Large Composed Models

Fortunately, huge models are rarely stated from scratch. They are obtained by composition of smaller ones, the main source of state number explosion being parallel compositions.

When there is no synchronisation, the parallel composition of r models Mod_1, $\ldots, Mod_r$ is the product of the underlying automata [2]. A brute force method to uniformly drawing paths is to build the product and to use the methods above. Since it is possible for moderate sizes only we have developed an alternative method that avoids the construction of the global model. This method is presented in detail in [4] [14] and [5]. We sketch it below.

- Given n the length of the global path to be drawn
- Choose some lengths $n_1, \ldots, n_r$ such that $\sum_{i=1,\ldots,r} n_i = n$, with adequate probabilities (see below)
- For each Mod_i, draw uniformly at random some path w_i of length n_i
- Interleave the r w_i in a randomised way that ensures uniformity among interleavings.

Let $\ell(n)$ be the number of global paths of length n, and $\ell_i(k), i = 1, \ldots, r$ the number of paths of length k in Mod_i. The choice of the $n_1, \ldots, n_r$ should be done with the probability below:

$$\Pr(n_1, \ldots, n_r) = \frac{\binom{n}{n_1, \ldots, n_r} \ell_1(n_1) \ldots \ell_r(n_r)}{\ell(n)} \tag{3}$$

where the numerator is the number of interleavings of length n that can be built with r local paths of lengths $n_1, \ldots, n_r$. Since computing the exact value of $\ell(n)$ would require the construction of the global model and of the corresponding tables, we use the following approximation from [8]:

$$\ell(n) \sim C\omega^n. \tag{4}$$

where C and ω are two constants. A sufficient, but not necessary, condition for this approximation to hold is aperiodicity and strong connectivity of the automaton, which is satisfied by any LTS with a reset. Details of weaker conditions can be found in [8]. This approximation is precise enough even for small values of n since $C\omega^n/\ell(n)$ converges to 1 at exponential rate.

Using the same approximations for the $\ell_i(n_i), i = 1, \ldots, r$, we get (see [4]):

$$\ell(n) \sim C_1 \ldots C_r(\omega_1 + \ldots + \omega_r)^n \tag{5}$$

and then

$$\Pr(n_1, \ldots, n_r) \sim \frac{\binom{n}{n_1, \ldots, n_r} \omega_1^{n_1} \omega_2^{n_2} \ldots \omega_r^{n_r}}{(\omega_1 + \omega_2 + \ldots + \omega_r)^n} . \tag{6}$$

This avoids the computation of $\ell(n)$, C and ω. The constants $\omega_i, i = 1, \ldots, r$ are computable in polynomial time with respect to the size of the Mod_i. It means that the complexity is dependent on the size of the components only, and not of the size of the global model.

In [4], we provide an algorithm for drawing $n_1, \ldots, n_r$ with this probability without computing it: draw a random sequence of n integers in $\{1, \ldots, r\}$, with the probability to choose i equal to $\Pr(i) = \frac{\omega_i}{\omega_1+\omega_2+\ldots+\omega_r}$; then take as n_i the number of occurrences of i in this sequence.

This concludes the issue of the choice of the $n_1, \ldots, n_r$. Then a classical randomised way of interleaving r w_i of lengths n_i that ensures uniformity is used.

This method is available in the RUKIA library. Experiments have been successfully led on models with 10^{34} states, with performances that show that the approach makes it possible to uniformly explore even larger models [14].

The generalisation to one synchronisation is given in [9]. The case of several synchronisations is studied in Oudinet's Ph.D. thesis [15] It turns out be practicable in the case of a small number of synchronisations only. The number of synchonisations can be increased by considering partial order reduction, i.e. by collapsing interleavings. Besides, in presence of many synchronisations, the synchronised product is smaller and a brute-force method, where it is constructed and used for uniform drawings, may become feasible. Actually, practical solutions are probably combinations of these two approaches depending on the architecture of the global system.

4 Randomised Coverage of States, Transitions, and Other Features

Path coverage is known to be too demanding, due to the path number explosion. Thus it is of interest to consider other coverage criteria. In [11] [3] [5], we have defined a notion of randomised coverage satisfaction for random testing methods.

What does it mean for a random exploration method to take into account a coverage criterion? Let $E_C(\mathcal{G})$ be the set of elements characterising a coverage criterion C for a given graph $\mathcal{G}$, for instance the vertices or the arcs of $\mathcal{G}$, or some subset of them.. The satisfaction of this coverage criterion C by some random exploration of the graph $\mathcal{G}$ can be characterised by the minimal probability $q_{C,N}(\mathcal{G})$ of covering any element of $E_C(\mathcal{G})$ when drawing N paths. $q_{C,N}(\mathcal{G})$ can be easily stated as soon as $q_{C,1}(\mathcal{G})$ is known.One has $q_{C,N}(\mathcal{G}) = 1-(1-q_{C,1}(\mathcal{G}))^N$.

Given a coverage criteria and some given random testing method, the elements to be covered have generally different probabilities to be reached by a test. Some of them are covered by all the tests. Some of them may have a very weak probability, due to the structure of the graph or to some specificity of the testing method.

Let $E_C(\mathcal{G}) = \{e_1, e_2, ..., e_m\}$ and for any $i \in (1..m)$, let p_i the probability for the element e_i to be exercised during the execution of a test generated by the considered random testing method. Then

$$q_{C,N}(\mathcal{G}) = 1 - (1 - p_{min})^N \tag{7}$$

where $p_{min} = min\{p_i | i \in (1..m)\}$. This definition corresponds to a notion of *randomised coverage satisfaction*. It makes it possible to assess and compare random exploration methods with respect to a coverage criterion.

Conversely, the number N of tests required to reach a given probability of satisfaction $q_C(\mathcal{G})$ is

$$N \geq \frac{log(1 - q_C(\mathcal{G}))}{log(1 - p_{min})} \tag{8}$$

By definition p_{min} gives $q_{C,1}(\mathcal{G})$. Thus, from the formula above one immediately deduces that for any given $\mathcal{G}$, for any given N, maximising the quality of a random testing method with respect to a coverage criteria C reduces to maximising $q_{C,1}(\mathcal{G})$, i. e. p_{min}. Note that uniform drawing of bounded paths, as presented in Section 3 maximises p_{min} on the set of paths to be covered.

A more developed discussion of these issues can be found in [5], together with the treatment of state coverage and transition coverage: some methods for computing their probabilities for a given model are given . These methods were first developed, implemented and experimented for C programs in [11] [3].

5 Uniformly Randomised LTL Model-Checking

The big challenge of model-checking is the enormous sizes of the models. Even when the best possible abstractions and restrictions methods have been applied, it may be the case that the remaining size is still significantly too large to perform exhaustive model explorations. Giving up the idea of exhaustivity for model-checking leads to the idea of using test selection methods for limiting the exploration of models.

One of these methods is randomisation of the search algorithm used for model exploration. A first introduction of randomisation into model-checking has been described and implemented in [12] as a Monte-Carlo algorithm for LTL model-checking. The underlying random exploration is based on a classical uniform drawing among the transitions starting from a given state. As said in Section 2 the drawback of such random explorations is that the resulting distribution of the exploration paths is dependent on the topology of the model, and some paths may have a very low probability to be traversed.

In [18], we have studied how to perform uniform random generation of lassos, which are the kind of paths of interest for LTL model-checking. This implies counting and drawing elementary circuits, which is known as a hard problem. However, efficient solutions exist for specific graphs, such as reducible data flow graphs which correspond to well-strucured programs and control-command systems. An immediate perspective is to embed this method in an existing model-checker such as SPIN or CADP, with the aim of developing efficient randomised methods for LTL model-checking with as result a guaranted probability of satisfaction of the checked formula.

The interest of this approach is that it maximises the minimal probability to reach a counter-example, and makes it possible to state a lower bound of this probability after N drawings, giving an assessment of the quality of the approximation.

6 Conclusion

In this set of works, we study the combination of coverage criteria with random walks. Namely, we develop some methods for selecting paths at random in a model. The selection is biased toward a coverage criterion. We have introduced a notion of randomised coverage satisfaction of elements of the model such as states, transitions, or lassos which are of interest for checking or testing LTL formulas.

We use methods for counting and generating combinatorial structures, presenting several original applications of this rich corpus of knowledge. They open numerous perspectives in the area of random testing, model checking, or simulation of protocols and systems.

Acknowledgments. The works reported here have been led in the RASTA working group (RAndom System Testing and Analysis), which involves members of various research teams of LRI (Algorithms and Complexity, Bioinformatics, Formal Testing and System Exploration) and of the Equipe de Logique Mathématique at University Paris 7. The current and past members are: Alain Denise, Marie-Claude Gaudel, Sandrine-Dominique Gouraud, Richard Lassaigne, Johan Oudinet, and Sylvain Peyronnet.

References

1. Aldous, D.: An introduction to covering problems for random walks on graphs. J. Theoret Probab. 4, 197–211 (1991)
2. Arnold, A.: Finite Transition Systems. Prentice-Hall, Englewood Cliffs (1994)
3. Denise, A., Gaudel, M.C., Gouraud, S.D.: A generic method for statistical testing. In: IEEE Int. Symp. on Software Reliability Engineering (ISSRE), pp. 25–34 (2004)
4. Denise, A., Gaudel, M.C., Gouraud, S.D., Lassaigne, R., Peyronnet, S.: Uniform random sampling of traces in very large models. In: 1st International ACM Workshop on Random Testing, pp. 10–19 (July 2006)
5. Denise, A., Gaudel, M.C., Gouraud, S.D., Lassaigne, R., Oudinet, J., Peyronnet, S.: Coverage-biased random exploration of large models and application to testing. STTT, International Journal on Software Tools for Technology Transfer Online First, 26 pages (2011)
6. Denise, A., Zimmermann, P.: Uniform random generation of decomposable structures using floating-point arithmetic. Theoretical Computer Science 218, 233–248 (1999)

7. Flajolet, P., Zimmermann, P., Cutsem, B.V.: A calculus for the random generation of labelled combinatorial structures. Theoretical Computer Science 132, 1–35 (1994)
8. Flajolet, P., Sedgewick, R.: Analytic Combinatorics. Cambridge University Press, Cambridge (2009)
9. Gaudel, M.C., Denise, A., Gouraud, S.D., Lassaigne, R., Oudinet, J., Peyronnet, S.: Coverage-biased random exploration of large models. In: 4th ETAPS Workshop on Model Based Testing. Electronic Notes in Theoretical Computer Science, vol. 220(1), 10, pp. 3–14 (2008), invited lecture
10. Goldwurm, M.: Random generation of words in an algebraic language in linear binary space. Information Processing Letters 54(4), 229–233 (1995)
11. Gouraud, S.D., Denise, A., Gaudel, M.C., Marre, B.: A new way of automating statistical testing methods. In: IEEE International Conference on Automated Software Engineering (ASE), pp. 5–12 (2001)
12. Grosu, R., Smolka, S.A.: Monte-Carlo Model Checking. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 271–286. Springer, Heidelberg (2005)
13. Nijenhuis, A., Wilf, H.S.: The enumeration of connected graphs and linked diagrams. J. Comb. Theory, Ser. A 27(3), 356–359 (1979)
14. Oudinet, J.: Uniform random walks in very large models. In: RT 2007: Proceedings of the 2nd International Workshop on Random Testing, pp. 26–29. ACM Press, Atlanta (2007)
15. Oudinet, J.: Approches combinatoires pour le test statistique à grande échelle. Tech. rep., LRI, Université Paris-Sud 11, Ph. D. thesis,118 pages (November 2010), `http://www.lri.fr/~oudinet/en/research.html#publications`
16. Oudinet, J.: Random exploration of models. Tech. Rep. 1534, LRI, Université Paris-Sud XI, 15 pages (June 2010)
17. Oudinet, J., Denise, A., Gaudel, M.C.: A new dichotomic algorithm for the uniform random generation of words in regular languages. In: Conference on random and exhaustive generation of combinatorial objects (GASCom), Montreal, Canada, 10 pages (September 2010) (to appear)
18. Oudinet, J., Denise, A., Gaudel, M.C., Lassaigne, R., Peyronnet, S.: Uniform Monte-Carlo model checking. In: Giannakopoulou, D., Orejas, F. (eds.) FASE 2011. LNCS, vol. 6603, pp. 127–140. Springer, Heidelberg (2011)
19. Wilf, H.: A unified setting for sequencing, ranking, and selection algorithms for combinatorial objects. Advances in Mathematics 24, 281–291 (1977)

Formal Testing of Timed and Probabilistic Systems*

Manuel Núñez

Departamento de Sistemas Informáticos y Computación,
Universidad Complutense de Madrid, Madrid, Spain
mn@sip.ucm.es

Abstract. This talk reviews some of my contributions on formal testing of timed and probabilistic systems, focusing on methodologies that allow their users to decide whether these systems are *correct* with respect to a formal specification. The consideration of time and probability complicates the definition of these frameworks since there is not an obvious way to define correctness. For example, in a specific situation it might be desirable that a system is as fast as possible while in a different application it might be required that the performance of the system is exactly equal to the one given by the specification. All the methodologies have as common assumption that the system under test is a black-box and that the specification is described as a timed and/or probabilistic extension of the finite state machines formalism.

1 Introduction

Testing was classically considered an informal discipline. Actually, it was assumed that formal methods and testing were orthogonal lines of research. Quoting from Edsger W. Dijkstra's ACM Turing Lecture:

> *Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence. The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness.*

However, early work already showed that it is possible to successfully combine formal methods and testing [21,6] and during the last 20 years there has been a vast amount of work on formal testing [9]. In particular, several workshops are exclusively devoted to the topic and formal testing has a strong presence in testing conferences, including this one, and in scientific journals.

Formal testing techniques provide systematic procedures to check systems in such a way that the coverage of their critical parts/aspects depends less on the intuition of the tester. While the relevant aspects of some systems only concern

* This research has been partially supported by the Spanish MICINN project TESIS (TIN2009-14312-C02-01) and by the Santander-UCM Programme to fund research groups (GR35/10-A-910606).

what they do, in some other systems it is equally relevant *how* they do what they do. Thus, after the initial consolidation stage, formal testing techniques started to deal with properties such as the probability of an event to happen, the time that it takes to perform a certain action, or the time when a certain action happens.

The work on formal testing of timed systems has attracted a lot of attention during the last years. Most work considers that time is *deterministic*, that is, time requirements follow the form “after/before t time units...” Even though the inclusion of time allows to give a more precise description of the system to be implemented, there are frequent situations that cannot be accurately described by using this notion of deterministic time. For example, in order to express that a message will arrive at any point of time belonging to the interval $[0, 1]$ we will need, in general, infinite transitions, one for each possible value belonging to the interval. In this case, it would be more appropriate to use time intervals to describe the system. Let us consider now that we have to simulate the performance of a petrol station. Since the arrival of cars follows a Poisson distribution, we would need again to use an infinite number of transitions. Moreover, if we have to use a time interval we would be very imprecise since all that we could say is that the next car will arrive in the interval $[0, \infty)$. Thus, it would be very useful to have a mechanism allowing to express that a time constraint is given by using a random variable that follows a precise probability distribution function.

In addition to consider the temporal behavior of systems, it is also interesting to study their probabilistic behavior. The use of probabilities allows to quantify the non-deterministic choices that a system may undertake. For example, instead of just specifying that a dice can non-deterministically return a value between 1 and 6, we can give more information by pointing out that the probability associated with each of these values is equal to $\frac{1}{6}$. In order to introduce probabilities, we consider a variant of the *reactive* model [7]. A reactive model imposes a probabilistic relation among transitions departing from a given state and labeled by the same action but choices between different actions are not quantified. In our setting, we express probabilistic relations between transitions outgoing from a state and having the same input action (the output may vary). Technically, for each input and state, the addition of the probabilities associated with transitions departing from a given state and labeled with that input is equal to either 0 (if there are no transitions) or to 1.

2 Outline of the Talk

The bulk of the talk is devoted to present formal testing methodologies where the temporal behavior of systems is taken into account and it is based on a joint work with Mercedes G. Merayo and Ismael Rodríguez [16]. The last part of the talk shows how the timed framework can be extended with probabilistic information. This part of the talk is based on a joint work with Ana Cavalli, Iksoon Hwang and Mercedes G. Merayo [13].

During this talk, the considered formalisms are always simple extensions of the classical concept of *Finite State Machine.* Intuitively, transitions in finite state machines indicate that if the machine is in a state s and receives an input i then it will produce an output o and it will change its state to s'. An appropriate notation for such a transition could be $s \xrightarrow{i/o} s'$. If we consider a timed extension of finite state machines, a transition such as $s \xrightarrow{i/o}_t s'$ indicates that if the machine is in state s and receives the input i, it will perform the output o and reach the state s' after t time units. Similarly, if we consider that time is defined in stochastic terms, a transition as $s \xrightarrow{i/o}_\xi s'$ indicates that if the machine is in state s and receives the input i, it will perform the output o and reach the state s' after a certain time t with probability $F_\xi(t)$, where F_ξ is the probability distribution function associated with ξ. Finally, in a model where non-determinism is probabilistically quantified and time is defined in stochastic terms, a transition such as $s \xrightarrow{i/o}_{p,\xi} s'$ indicates that if the machine is in state s and receives the input i then with probability p the machine emits the output o and it moves to state s' before time t with probability $F_\xi(t)$.

If time is expressed in stochastic terms, then the black-box assumption complicates the work of the tester. In this case, testers cannot compare in a direct way timed requirements of the *real* implementation with those established in the model. The idea is that we can *see* the random variable associated with a given transition in the model, but we cannot do the same with the corresponding transition of the implementation, since we do not have access to it. Thus, in contrast with approaches considering fix time values, to perform a transition of the implementation once does not allow the tester to obtain all the information about its temporal behavior. Therefore, the tester must induce the system to perform the same transition several times to collect different time values.

Implementation relations are used to relate systems under test and specifications. It is very helpful to start by considering an implementation relation where time is not taken into account. In this case, the idea is that the implementation I does not *invent* anything for those inputs that are *specified* in the model. In order to cope with time, we do not take into account only that a system may perform a given action but we also record the amount of time that the system needs to do so. Several timed conformance relations can be defined according to the interpretation of *good* implementation and the different time domains. Time aspects add extra complexity to the task of defining these relations. For example, even though an implementation I had the same traces as a formal model S, we should not consider that I conforms to S if I is always *slower* than S. Moreover, it can be the case that a system performs the same sequence of actions for different times. These facts motivate the definition of several conformance relations. For example, it can be said that an implementation conforms to a formal model if the implementation is always *faster*, or if the implementation is at least as *fast* as the worst case of the model.

With respect to the application of tests to implementations, the above mentioned non-deterministic temporal behavior requires that tests work in a specific manner. For example, if we apply a test and we observe that the implementation takes less time than the one required by the formal model, then this single application of the test allows us to know that the implementation *may* be faster than the model, but not that it *must* be so.

3 Other Work on Testing of Timed and Probabilistic Systems

In addition to the work presented in this talk, I have also participated in the development of other frameworks to test timed and probabilistic systems that I would like to briefly review. First, it is worth mentioning that the presented framework is general enough so that it can be easily modified to deal with other formalisms such as timed variants of stream X-machines [14]. The timed framework presented in this talk allows their users to express temporal requirements concerning the time elapsed between the reception of an input and the production of an output but it does not deal with *timeouts*. Therefore, this work [16] was extended to add timeouts: if after a certain amount of time the system does not receive and input, then a timeout will be invoked and the system will change its state [15]. Another interesting line of work consists in considering that the time information might not exactly reflect reality (e.g. due to bad equipment to measure time). Therefore, the tester might assume that small errors can be acceptable [17].

The previous approaches consider *active* testing, that is, the tester provides inputs to the system under test and analyzes the received outputs. However, in certain circumstances the tester cannot interact with the system and has to analyze observations of this system that are not controlled by him: he becomes a *passive* tester. During the last years, I was interested on the definition of a passive testing methodology for timed systems [3] and in the application of the methodology to real systems [2,1].

A different line of research in testing of timed systems consists in using genetic algorithms to select *better* test cases among the (possibly infinite) test cases that can be derived from a given specification [4,5].

Concerning testing of probabilistic systems, it is interesting to test the probabilistic behavior of systems with distributed ports since it is challenging to establish probabilistic relations between ports and between actions in the same port [10].

In some situations it is difficult to precisely specify the probabilities governing the behavior of the system. Therefore, instead of specifying that a certain event will happen with probability p, it is more appropriate to say that it will happen with a probability belonging to the interval $[p - \epsilon, p + \delta]$. The definition of a testing methodology for this kind of systems was quite interesting [12].

The previous approaches considered extensions of the **ioco** implementation relation [22]. A more theoretical line of work consists in defining timed and probabilistic extensions [19,20,11] of the classical de Nicola & Hennessy testing framework [18,8].

4 Future Work

I continue working on probabilistic and/or timed extensions of formal testing frameworks. Currently, I am very interested on testing systems with distributed ports. One line of work consists in using (probabilistic and distributed) schedulers to solve some of the problems detected on previous work [10]. Another line of work considers the analysis of time while testing systems with distributed ports. Using this additional information can help to fix the order in which events at different ports were performed. I am also working on applying our frameworks to test real systems. Specifically, the timed framework presented in this talk [16] is being use to test the temporal behavior of major household appliances while the probabilistic and timed framework presented during the talk [13] is being applied to analyze wireless communication protocols.

Acknowledgements. First, I would like to thank Burkhart Wolff and Fatiha Zaïdi, Chairs of the conference, for their kind invitation to be an invited speaker in ICTSS 2011. I would also like to thank all my colleagues with whom I have had the privilege of working on the study of timed and probabilistic aspects: César Andrés, Ana Cavalli, Karnig Derderian, David de Frutos, Carlos Gregorio, Rob Hierons, Iksoon Hwang, Luis Llana, Natalia López, Stéphane Maag, Mercedes G. Merayo, Pedro Palao, Fernando L. Pelayo, Ismael Rodríguez and Fernando Rubio.

References

1. Andrés, C., Maag, S., Cavalli, A., Merayo, M., Núñez, M.: Analysis of the OLSR protocol by using formal passive testing. In: 16th Asia-Pacific Software Engineering Conference, APSEC 2009, pp. 152–159. IEEE Computer Society, Los Alamitos (2009)
2. Andrés, C., Merayo, M., Núñez, M.: Applying formal passive testing to study temporal properties of the stream control transmission protocol. In: 7th IEEE Int. Conf. on Software Engineering and Formal Methods, SEFM 2009, pp. 73–82. IEEE Computer Society, Los Alamitos (2009)
3. Andrés, C., Merayo, M., Núñez, M.: Formal passive testing of timed systems: Theory and tools. Software Testing, Verification and Reliability (2012) (accepted for publication)
4. Derderian, K., Merayo, M., Hierons, R., Núñez, M.: Aiding test case generation in temporally constrained state based systems using genetic algorithms. In: Cabestany, J., Sandoval, F., Prieto, A., Corchado, J.M. (eds.) IWANN 2009. LNCS, vol. 5517, pp. 327–334. Springer, Heidelberg (2009)
5. Derderian, K., Merayo, M., Hierons, R., Núñez, M.: A case study on the use of genetic algorithms to generate test cases for temporal systems. In: Cabestany, J., Rojas, I., Joya, G. (eds.) IWANN 2011, Part II. LNCS, vol. 6692, pp. 396–403. Springer, Heidelberg (2011)
6. Gaudel, M.C.: Testing can be formal, too! In: Mosses, P.D., Nielsen, M. (eds.) CAAP 1995, FASE 1995, and TAPSOFT 1995. LNCS, vol. 915, pp. 82–96. Springer, Heidelberg (1995)

7. Glabbeek, R.v., Smolka, S., Steffen, B.: Reactive, generative and stratified models of probabilistic processes. Information and Computation 121(1), 59–80 (1995)
8. Hennessy, M.: Algebraic Theory of Processes. MIT Press, Cambridge (1988)
9. Hierons, R., Bogdanov, K., Bowen, J., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Luettgen, G., Simons, A., Vilkomir, S., Woodward, M., Zedan, H.: Using formal methods to support testing. ACM Computing Surveys 41(2) (2009)
10. Hierons, R., Núñez, M.: Testing probabilistic distributed systems. In: Hatcliff, J., Zucca, E. (eds.) FMOODS 2010. LNCS, vol. 6117, pp. 63–77. Springer, Heidelberg (2010)
11. Llana, L., Núñez, M.: Testing semantics for RTPA. Fundamenta Informaticae 90(3), 305–335 (2009)
12. López, N., Núñez, M., Rodríguez, I.: Specification, testing and implementation relations for symbolic-probabilistic systems. Theoretical Computer Science 353(1-3), 228–248 (2006)
13. Merayo, M., Hwang, I., Núñez, M., Cavalli, A.: A statistical approach to test stochastic and probabilistic systems. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 186–205. Springer, Heidelberg (2009)
14. Merayo, M., Núñez, M., Hierons, R.: Testing timed systems modeled by stream X-machines. Software and Systems Modeling 10(2), 201–217 (2011)
15. Merayo, M., Núñez, M., Rodríguez, I.: Extending EFSMs to specify and test timed systems with action durations and timeouts. IEEE Transactions on Computers 57(6), 835–848 (2008)
16. Merayo, M., Núñez, M., Rodríguez, I.: Formal testing from timed finite state machines. Computer Networks 52(2), 432–460 (2008)
17. Merayo, M., Núñez, M., Rodríguez, I.: A formal framework to test soft and hard deadlines in timed systems. Software Testing, Verification and Reliability (accepted for publication, 2011), doi: 10.1002/stvr.448
18. de Nicola, R., Hennessy, M.: Testing equivalences for processes. Theoretical Computer Science 34, 83–133 (1984)
19. Núñez, M.: Algebraic theory of probabilistic processes. Journal of Logic and Algebraic Programming 56(1-2), 117–177 (2003)
20. Núñez, M., Llana, L.: A hierarchy of equivalences for probabilistic processes. In: Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K. (eds.) FORTE 2008. LNCS, vol. 5048, pp. 267–282. Springer, Heidelberg (2008)
21. Sidhu, D., Leung, T.K.: Formal methods for protocol testing: A detailed study. IEEE Transactions on Software Engineering 15(4), 413–426 (1989)
22. Tretmans, J.: Model based testing with labelled transition systems. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST. LNCS, vol. 4949, pp. 1–38. Springer, Heidelberg (2008)

Improved Usage Model for Web Application Reliability Testing

Gregor v. Bochmann, Guy-Vincent Jourdan, and Bo Wan

School of Information Technology & Engineering, University of Ottawa, Ottawa, Canada
`{bochmann,gvj,bwan080}@ site.uottawa.ca`

Abstract. Testing the reliability of an application usually requires a good usage model that accurately captures the likely sequences of inputs that the application will receive from the environment. The models being used in the literature are mostly based on Markov chains. They are used to generate test cases that are statistically close to what the application is expected to receive when in production. In this paper, we study the specific case of web applications. We present a model that is created directly from the log file of the application. This model is also based on Markov chains and has two components: one component, based on a modified tree, captures the most frequent behavior, while the other component is another Markov chain that captures infrequent behaviors. The result is a statistically correct model that exhibits clearly what most users do on the site. We present an experimental study on the log of a real web site and discuss strength and weakness of the model for reliability testing.

Keywords: Web applications, Usage models, Reliability testing, Markov chains.

1 Introduction

Many formal testing techniques are directed towards what is sometimes called "debug techniques": the goal is to fulfill some given criteria (branch coverage, all-uses coverage, all paths, all code and many others), or uncover every fault[1] using some restricted fault models (checking experiments). However, in practice, non-trivial applications are simply not expected to ever be failure-free, thus the purpose of a realistic testing campaign cannot be to find all the faults. Given that only some of the failures will be uncovered, it only makes sense to question which ones will be found by a testing method. Note that in a realistic setting, failures are always ranked by importance.

In this paper, we are interested in testing the reliability of an application. For a material system, reliability is usually defined by the expected time of operation after

[1] In this document, a "fault" in the application source code leads to a "failure" at execution time. A "test sequence" is a sequence of interactions between the testing environment (e.g. the tester) and the tested application. "Test input data" is the data that is input during the execution of the test sequence. A test sequence, valued with test input data, is a "test case". A test case may uncover a failure, due to one (or more) fault.

which the system will fail. In the case of a software system, it can be defined by the expected number of usages before it will fail. A usage, in this context, may be a request provided by the environment, or a complete usage session, for instance in the case of an application with an interface to a human user. Clearly, the occurrence of a failure of a software system is dependent on the input provided. In order to test the reliability of an application, it is therefore important to apply inputs that reflect the behavior of the environment of the application in the normal operating conditions. This is sometimes called "operational testing". In this context, it is important to test first those behavior patterns that occur most frequently under normal operating conditions. This idea has been applied with great success on certain large software projects: Google was for example able to deliver an internet browser, Chrome, that was remarkably reliable from its first release, not necessarily because it was tested against more web pages than the other browsers, but because it was tested against the web pages that Google knew people were most looking at[2].

There are essentially two methods for obtaining a realistic model of the behavior of the environment of the application to be tested for reliability:

1. Environment model based on the application model: If the functional behavior requirements of the application are given in the form of an abstract model, for instance in the form of a UML state machine model, this model can be easily transformed into a model of the environment by exchanging input and output interactions. However, for obtaining an environment model useful for reliability testing, this functional model must be enhanced with statistical performance information about the frequency of the different inputs applied to the application in the different states of the environment. This may be formalized in terms of a Markov model based on the states of the abstract functional application model.
2. Environment model extracted from observed execution traces in a realistic environment: Independently of any model of the application that may be available, a model of the dynamic behavior of the environment may be extracted from the observation of a large number of execution traces that have occurred in a realistic setting.

In this paper, we pursue the second approach. We assume that the application to be tested is a Web application. We make the assumption that after each input by the user, the response from the web server provides information about the functional state of the application. In the case of traditional web applications, the state information is given by the URL of the page that is returned. In the case of Rich Internet Applications (RIA), we consider that the content of the returned web page, that is the DOM of this page, represents the application state, assuming that there is no hidden state information stored in the server.

In previous work on reliability testing, the user model is usually either given in the form of a tree of possible execution sequences with associated probabilities for each branching point [1][2], or in the form of a Markov model [3],[4],[5],[6]. We show in this paper how one can extract, from a given set of execution sequences, a user model that is a combination of an execution tree and a traditional Markov model. We first

[2] See `http://www.google.com/googlebooks/chrome/`, page 10 for a graphical illustration.

construct the execution tree from the given set of execution sequences. The upper branches of the tree have usually been executed a large number of times which means that good statistical information is available for the branching probabilities. This part of our model is called the "upper tree". For the lower branches of the tree, however, there are usually only one or a few executions that have been observed; therefore the statistical information about branching probabilities is very weak. We therefore remove these lower branches and combine them into a Markov model for which the branching probabilities are obtained from the union of all the lower branches. This part of our model is called "lower Markov model". Our resulting user model is therefore a Markov model which contains two parts, the "upper tree" and the "lower Markov model".

The "lower Markov model" is a traditional (first-order) Markov model where each state of the model corresponds to a state of the application. However, the "upper tree" is a higher-order Markov model which may contain several different Markov states corresponding to the same application state; this is the case when the behavior of the user does not only depend on the current application state, but also on the path which was taken to get to this state. In contrast to other statistical modeling methods starting out with observed execution sequences that can only model dependencies on previous states up to a limited number of interactions [7],[8] our "upper tree" can model dependencies on previous application state for arbitrarily long state sequences.

This paper is structured as follows. Section 2 presents an overview of Markov usage models and their application in reliability testing, followed by a brief review of previous work on Markov usage models in Web applications. Section 3 presents a detailed description of our hybrid Markov usage model. In Section 4, we present the results of experiments conducted with real data. In Section 5, we give our conclusions and present our plans for future research.

2 Review Markov Usage Model

Markov models are commonly used to model usage patterns and to establish reliability estimations because they are compact, simple to understand and based on well-established theory. K. Goseva-Popstojanova and K. S. Trivedi used Markov renewal processes to estimate software reliability [9],[10], but they did not use the usage model. Markov chains have been used extensively over the past two decades in the domain of statistical usage testing for software. In 1994, Whittaker and Thomason [3] explained how to use a Markov-chain-based model of software usage to perform statistical testing. Random walks on the model are performed to generate test sequences. These test sequences are applied to the implementation and the test experiment is run until enough data is gathered for the implementation under test. More recently, MaTeLo, an industrial tool also used Markov chains to model usage profiles, generate test cases, debug and estimate software reliability [4], [5], [11].

In Web applications reliability testing, a Markov chain model can be constructed from log files. When users visit a Web site, Web servers record their interactions with the Web site in a log file. The log file usually contains data such as the user's IP address, viewing time, required page URL, status, and browser agent. After some massaging of the data, it is possible to infer from the log files a reliable set of user sessions (see e.g. [12],[13],[14] for detailed explanations of how to obtain sessions from Web log files). Figure 1(a) illustrates the principle of building a Markov chain from log files. In this example, the Web pages being visited (identified in practice by their URLs) belong to the set {1, 2, 3, 4, 5}. In the following, we call such pages "application states". From the log files, visiting sessions have been reconstructed. Artificial starting state S and terminating state T are added to theses sessions for simplicity. Some of the reconstructed sessions may be identical if several users have followed the same sequence of pages on the Web site. We combine such sessions going through the same sequence of application states, to obtain what we call "application state sequences". In the figure, the column Nb shows how many times each application state sequence was followed. Figure 1.b presents the traditional Markov chain model for these sessions. In the Markov chain, the edges are labeled with probabilities representing the distribution of the user's choice for the next state from the current one.

Application state sequences	Nb.
S-1-3-5-T	4
S-3-4-T	2
S-2-3-4-T	1
S-2-3-5-T	3

(a)

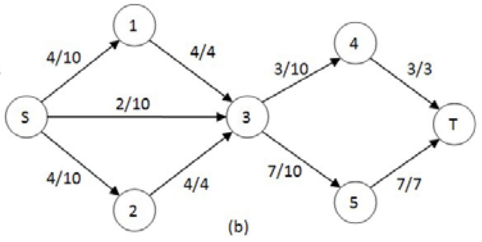


Fig. 1. An example of a traditional Markov Chain model: (a) a collection of application state sequences and (b) the corresponding traditional Markov chain model

Traditional Markov models are simple and compact, but they have also limitations when used to model usage profiles. In Web applications, a traditional Markov model, sometimes called *first-order* Markov model, captures the page-to-page transition probabilities: $p(x2|x1)$ where $x1$ denotes the current page and $x2$ denotes one of pages reachable from $x1$. Such low order Markov models cannot capture behavior where the choice of the next page to be visited depends on "history", that is, on how the current application state was reached. For example, in an e-commerce site, after adding an item to the shopping card, the user would typically either "proceed to checkout" or "continue shopping". The probability of doing one or the other is certainly not identical after adding one item, after adding two items etc. Another example is shown in Figure 1: In the snippet of a traditional Markov usage model (b), we see that there are three ways to reach state 3 (from state 1, from state 2 and from state S), and that from state 3, there is 30% chance to go to state 4, and 70% chances to go the state 5. However, looking at the provided application state sequences, we can see that users reaching state 3 from state 1 never go to state 4 afterwards. What is shown in the traditional Markov chain is misleading. Since it is reasonable that most Web applications involve such history-dependent behavior, accurate models of user behavior cannot be obtained with first order Markov chains [15]. The same problem is also discussed by Deshpande and Karypis [16]. Thus, a good usage model requires higher-order Markov chains.

A higher-order Markov model has already been explored by Borges and Levene in 2000 to extract user navigation patterns by using a Hypertext Probabilistic Grammar model structure (HPG) and N-grams [7]. In their work, an N-gram captures user behavior over a subset of N consecutive pages. They assume that only the N-1 previous pages have a direct effect on the probability of the next page selected. To capture this, they reuse the concept of "gram" taken from the domain of probability language learning [17]. Consider, for example, a web site composed of six states {*A1, A2, A3, A4, A5, A6*}. The observed application state sequences are given in Table 1 (Nb denotes the number of occurrences of each sequence).

Table 1. A collection of application state sequences

Application State Sequences	Nb
A1-A2-A3	3
A1-A2-A4	1
A5-A2-A4	3
A5-A2-A6	1

A bigram model is established using first-order probabilities. That is, the probability of the next choice depends only on the current position and is given by the frequency of the bigram divided by the overall frequency of all bigrams with the same current position. In the example of Table 1, if we are interested in the probabilities of choices from application state A2, we have to consider bigrams (sequences including two application states) that start with state A2. This includes the following: Segment A2-A3 has a frequency of 3, and other bigrams with A2 in their current position include the segments A2-A4 and A2-A6 whose frequency are 4 and 1, respectively; therefore, *p(A3|A2)=3/(3+4+1)=3/8*. It is not difficult to see that the 2-gram model is a first-order Markov chain, the traditional Markov usage model. The second-order model is obtained by computing the relative frequencies of all trigrams, and higher orders can be computed in a similar way. Figure 2 shows the 3-gram model corresponding the sessions in Table 1.

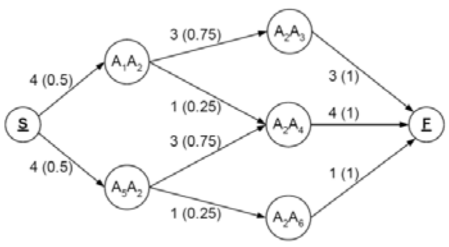


Fig. 2. 3-gram model corresponding to the sessions given in Table 1

Subsequently, the same authors showed in 2004 how to use higher-order Markov models in order to infer web usage from log files [8]. In this paper, they propose to duplicate states for which the first-order probabilities induced by their out-links diverge significantly from the corresponding second-order probabilities. Take Table 1 again as example. Consider state 2 and its one-order probability *p(A3|A2)=3/8*, and its two-order probability *p(A3|A1A2)=3/4*. The large difference between *p(A3|A2)* and *p(A3|A1A2)* indicates that coming from state *A1* to state *A2* is a significant factor on the decision to visit *A3* immediately afterwards. To capture this significant effect, they split state *A2* as illustrated in figure 3. A user-defined threshold defines how much the first and second order probabilities must differ to force a state splitting. A k-means clustering algorithm is used to decide how to distribute a state's in-links between the split states.

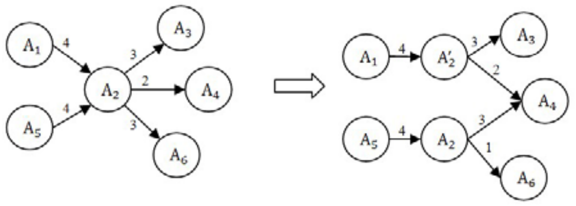


Fig. 3. An example of the cloning operation in dynamic clustering modeling

All these approaches focus on the last N-1 pages and will thus ignore effects involving earlier pages visited. In addition, the fixed-order Markov model also has some limitations in accuracy as pointed out by Jespersen, Pedersen and Thorhauge [18].

3 Hybrid Tree-Like Markov Usage Model

In this section, we introduce a new method to infer a probabilistic behavioral model from a collection of sessions extracted from the logs of a Web application. The model draws from both a traditional Markov chain usage model and a tree of application state sequences which is introduced in the next Section. The new usage model contains a modified tree of state sequences that captures the most frequent behaviors, and a traditional Markov chain model recording infrequent behavior.

Table 2. A collections of Application state sequences

Application State Sequence	Nb
S-1-1-3-5-T	1
S-1-3-2-1-2-4-T	4
S-1-3-2-2-4-T	9
S-2-3-4-2-2-4-T	4
S-2-3-4-4-T	21
S-2-3-4-2-3-4-T	14
S-3-3-4-2-4-T	23
S-3-3-4-2-T	4
S-3-3-4-4-T	33
S-3-2-2-3-4-T	4
S-3-2-2-5-T	4
S-3-2-4-5-T	4
S-3-2-4-3-5-T	4
S-3-T	2

The usage model is built from a collection of user sessions, to which we add a common starting and terminating state. Again, different users can go over the same application states during their sessions. We group sessions in “application state sequences”, as discussed above. Table 2 shows an example, along with the number of times each state sequence occurs in the log files.

3.1 Building the Tree of Sequences

The tree of sequences (TS for short) is constructed from the given state sequences by combining their longest prefix. Each node in the tree, called model state, corresponds to an application state (e.g. the URL of the current page), but each application state has in general several corresponding model states. The tree captures the application state sequence that was followed to reach a given model state; this “history” corresponds to the path from the root of the tree to the model state. This is unlike the traditional Markov chain usage model where there is a one-to-one mapping between application states and model states. In addition, each edge of the tree is labeled with the number of application state sequences that go over this particular branch. Figure 4 shows the TS model built from the state sequences listed in Table 2. One major advantage of the TS model is that it can be used to see the conditional distribution of the next state choice based on the full history. For example, the probability of choosing state 2 from state 4 after the state sequence 2-3-4 is *p(2|2-3-4)=18/39* while the probability of choosing state 4 under the same circumstances is *p(4|2-3-4)=21/39.*

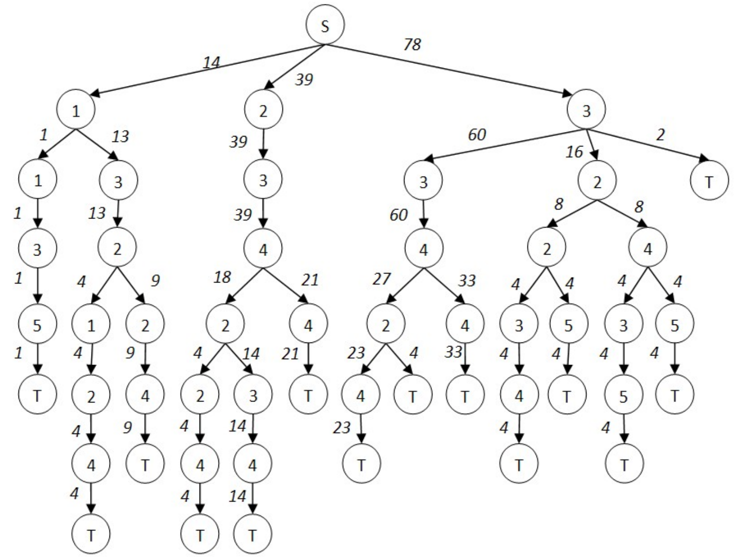


Fig. 4. The tree of sequences captured from Table 2

Despite its strengths, the TS model has many weaknesses. One major disadvantage is the fact that the probabilities calculated for each transition might not be reliable if the state sequences have not been followed very often. Since the tree tends to be very wide there are many such sequences. And long sequences tend to be less and less representative the longer they go (that is, many users may have followed a prefix of the sequence, but few have followed it to the end). In addition, we mention that (a) it tends to be fairly large, (b) it is not adequate for creating new test cases, and (c) it does not pinpoint common user behavior across different leading state sequences.

3.2 Frequency-Pruned TS Model

To overcome some of the problems of the TS model, we first apply a simple technique that we call "frequency pruning". This is based on the observation that model states that occur with low frequency in the application state sequences do not carry reliable information from a statistical point of view. We note that, for such states, the estimation of the conditional probabilities will not be reliable [16]. Consequently, these low frequency branches can be eliminated from the tree without affecting much the accuracy of the model. However, just applying such pruning would impact the coverage that can be inferred from the model, since it removes some low frequency but still very real branches. To avoid this problem, we do not discard these pruned branches, instead we include the corresponding state sequences in the "lower Markov model" (introduced below), which is a traditional Markov usage model.

The amount of pruning in the TS model is controlled by a parameter, called "frequency threshold" θ. When the calculated conditional probability of a branch is lower than the frequency threshold, the branch is cut. Figure 5 shows the result of pruning of the TS model of figure 4, with θ set at 10%. For example, we have $p(1|1)=1/14 < \theta$, therefore the branch 1-3-5-T is cut from the tree and will be used when building the "lower Markov model". The grey nodes in figure 5 represent access point from the TS model to this Markov model.

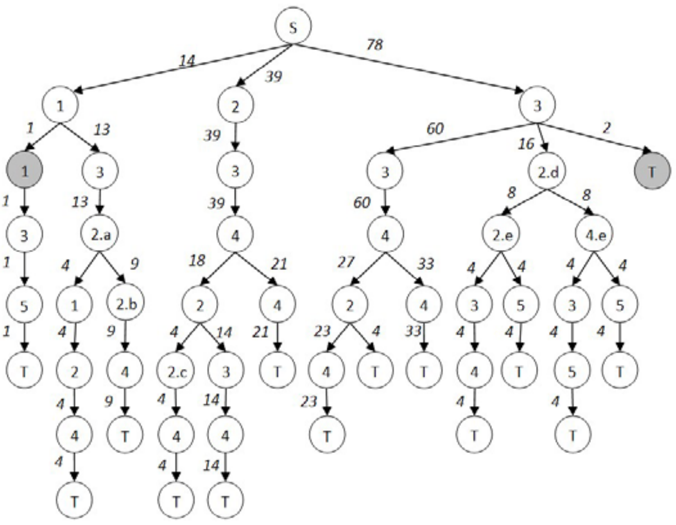


Fig. 5. Frequency-pruned TS Model

3.3 Hybrid Markov Usage Model

Our goal is to strike the right balance between the traditional Markov chain model and the TS model. We want to have separate model states for the instances of applications states when the user behavior is statistically different, and we want to merge them into a single model state when the user behavior cannot be statistically distinguished (be it that the users behaves identically or that we do not have enough information to make the difference). Working from the two previous models, one could start from the traditional Markov usage model and “split” states for which a statistically different behavior can be found depending on the history, or one could start from the TS model and “merge” states that are instances of the same application state for which no significant behavior difference can be found (or even iterate on merges and splits).

In this paper, we work from the frequency-pruned TS model and merge states. The goal is thus to look at different model states which represent the same application state and decide whether the recorded user behavior is statistically significantly different. If so, the states must be kept apart, and otherwise the states are candidates to be merged.

3.4 Independence Testing and State Merging

As shown in the example of Figure 7, the TS model contains in general many model states that correspond to the same application state. For instance, the states 4.a, 4.b and 4.c all correspond to the application state 4. To simplify the user behavior model, we would like to combine such states in order to reduce the number of states in the model. However, this should only be done if the user behavior is the same (or very similar) in the different merged states. We therefore have to answer the following question for any pair of model states corresponding to the same application state: Is the recorded user behavior statistically significantly different on these two states? In other words, is the users’ behavior dependant on how they have reached this application state? – If the answer is yes, then the model states should be kept separated, otherwise they should be merged. An example is shown Figure 6 (a) where the statistical user behavior is nearly identical in the two states 1.a and 1.b, and these two states could therefore be merged leading to Figure 6 (b).

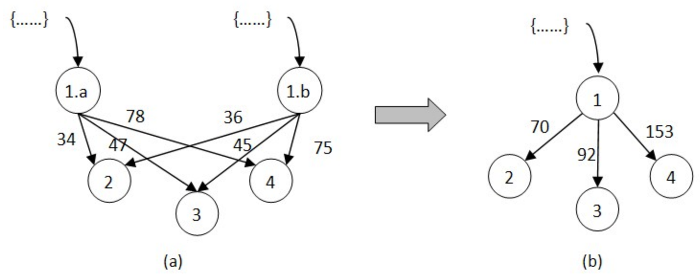


Fig. 6. An example of merging two model states

We note that the model states that are the successors of the states to be merged must be identical for the two states to be merged, as shown by the example of

Figure 6. This implies that the merging operations must be applied from the bottom-up through the original tree-like TS model. We note that the terminal states labeled T can be merged (because they have the same user behavior). However, the model states that precede the final state, and many of the other preceding states, have only few occurrences in the application state sequences inferred from the log files. Therefore the statistical significance of these occurrences is not so strong; therefore a decision for merging is difficult to make.

There are a number of statistical methods to answer to these questions. We will use in the following the so-called "test of independence", itself based on the chi-square test (see for instance [19]). However, we note that the test of independence gives only reliable answers when there is enough statistical information. The so-called "Cochran criterion" states that in order to apply the test of independence, at most 20% of the possible alternatives should have fewer than six instances in the sample set. As discussed above, many of the model states in the lower part of the TS model will fail the Cochran criterion and thus cannot be used for the test of independence since they do not carry enough information to be statistically significant.

We therefore propose to merge into a single model state all TS model states that correspond to a given application state and do not satisfy the Cochran criterion. These merged states form what we called "lower Markov model" in the Introduction. For our running example of Figure 5, we obtain after the application of the Cochran criterion the model of Figure 7. The grey nodes form the "lower Markov model". For example, the model states 2.a, 2.b, 2.c, etc of Figure 5 were merged into the state 2.e of Figure 7. For state 2.d in Figure 5, for instance, there are two choices to go to state 2.e or to state 4.e with the frequencies of 8 and 8 respectively. They are represented in Figure 7 as state 2.d to state 2.e or state 4.e with frequencies 8 and 8, respectively.

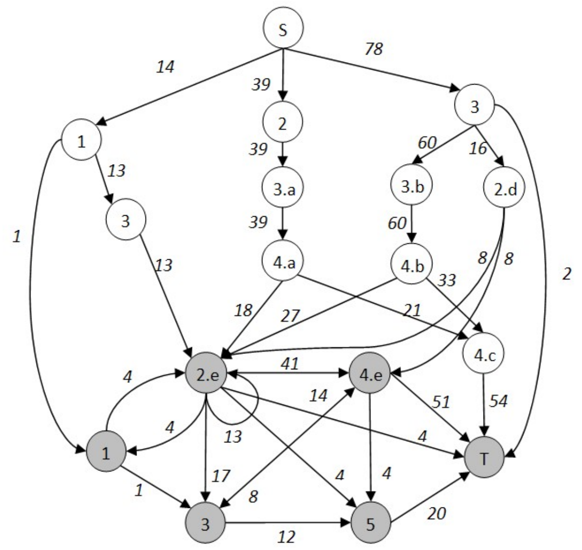


Fig. 7. The model after pruning based on Cochran criterion

We note that the final "lower Markov model" will also include the application state sequences of the tree branches that were pruned during the frequency-pruning phase described in Section 3.2. The frequency-pruning steps is applied first to avoid running into situations in which a model state traversed by a large number of application state transitions still fails the Cochran criterion because a few very infrequent behavior have been observed there (for example because of sessions created by web crawler).

After applying the Cochran criterion and constructing the "lower Markov model", we check the remaining model states in the "upper tree" for the possibility of merging by applying the chi-square-based independence test in a bottom-to-top order. We apply this test pairwise, even if there are more than two model states corresponding to the same application state.

The value of chi-square indicates how good a fit we have between the frequency of occurrence of observations in an observation sample and the expected frequencies obtained from the hypothesized distribution. Assuming that we have k possible observations and have observed $o_i(i = 1, \dots k)$ occurrences of observation i while the expected frequencies are $e_i(i = 1, \dots k)$, the value of chi-square is obtained by Formula (1)

$$\chi^2 = \sum_{i=1}^{k} \frac{(o_i - e_i)^2}{e_i} \tag{1}$$

χ^2 is a value of a random variable whose sampling distribution is approximated very closely by the chi-square distribution for (k-1) degrees of freedom [19].

Let us consider the example shown in Table 3 below. It shows the observed choices to application states 2 and 4 from the model states 4.a and 4.b (see Figure 7). If we assume that these two model states can be merged, that is, the branching probabilities to states 2.c and 4.c is almost identical, we can calculate these branching probabilities by considering the union of all observed sessions going through states 4.a and 4.b (see last row in the table). This leads to the expected number of choices indicated in the last two columns of the table. For instance, the probability of choosing application state 2 is 45/99, and therefore the expected number of choices of state 2 from model state 4.a is 45/99 * 39 = 17.73.

Table 3. Example of chi-square calculation

	Observed occurrences			Expected occurrences	
Next State	4.a	4.b	total	4.a	4.b
2	18	27	45	17.73	27.27
4	21	33	54	21.27	32.73
total	39	60	99	39	60

Then we use the numbers in the table to calculate χ^2 according to formula (1) for model states 4.a and 4.b and take the average. The result is the χ^2 value that we can use to determine whether our hypothesis is valid for a given confidence level, using a table of the chi-square distribution for one degree of freedom. In the case of our example, we get a χ^2 value of 0.0124. Since this value is smaller than $\chi^2_{0.05}$= 3.841 we can say with confidence level of 95% that the model states 4.a and 4.b represent the same user behavior, and the states can be merged, as shown in Figure 8.

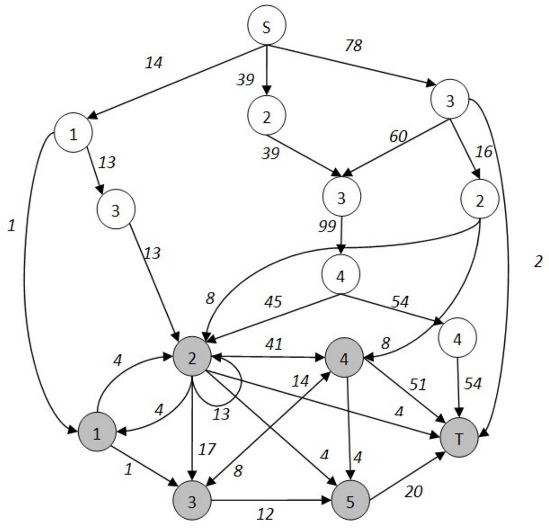


Fig. 8. Hybrid Markov model constructed by sessions in table 2

We apply such merging tests to all pairs of model states that correspond to the same application state and that have outgoing transitions to the same set of model states[3]. This is done from the bottom of the “upper tree” towards its root. The states at the bottom have transitions that lead to states of the “lower Markov model”, which are already merged, thus the test can always be applied to these bottom states. If these bottom states are merged, then the test can be applied to their parents and this keeps going until the test fails. As already explained, one example of two states that can be merged is states 4.a and 4.b (see Figure 7 and Figure 8). Once they are merged, some states higher in the tree may become candidates for merging. In this example, the parent nodes of 4.a and 4.b, namely nodes 3.a and 3.b, respectively, can also be merged (because they have only one successor which is the same). Once all candidates for merging have either been merged or are determined not to satisfy the merging condition, then we obtain our final performance model, as shown for our example in Figure 8.

4 Experiment

We experimented our approach on a web site called Bigenet (http://www.bigenet.org). Bigenet is a genealogy web site allowing access to numerous registers – birth, baptism, marriage, death and burials – in France. Two international exchange students, Christophe Günst and Marie-Aurélie Fund, developed a tool which is able to generate a list of visiting sessions from the access log files of the web server and the functional model of the application. The tool follows the approach presented in [12].

[3] If some model states are reached by only one of the two states being tested, we assume that the other state also reaches to the same states but with a probability 0.

We had at our disposal the access log files for the period from September 2009 to September 2010. Table 4 presents a summary of the characteristics of the visiting sessions during this period.

Table 4. Summary Statstic for the data set from Bigenet

Characteristics	Bigenet
Num. of Application States	30
Num. of Request	900689
Num. of Sessions	108346
Num. of Application State Sequences	27778
Ave. Session length	8.3132
Max. Session length	301

We have developed a second tool that implements the model construction approach described in Section 3. It creates the TS model from the list of application state sequences inferred from the reconstructed sessions, and then performs the pruning, Cochran merging and state merging based on independence tests. The TS model constructed from the whole year of visiting sessions is very large, containing 348391 nodes in the tree. Figure 9 shows the TS model based on 1000 visiting sessions.

Table 5 shows the results of our analysis of these 108346 user sessions (see column labeled "All"). The following parameters were used during our analysis: (a) the pruning threshold was 5%; (b) the confidence level for the independence test was 95%.

Table 5. Summary of experimental results

	All	Set-1	Set-2	Set-3	Set-4
Num. of states in TS	348391	38652	38463	38336	42039
Num. of states after frequency pruning	81364	12493	11796	12438	13066
Num. of states in "lower Markov model"	30	30	29	30	29
Num. of states in "upper tree" before merging	426	78	79	76	82
Num. of states in "upper tree" after merging	337	65	68	65	68
Num. of independence tests applied	108	17	15	15	18
Num. of mergings performed	89	13	11	11	14
Execution time without optimization[4]	3937ms	313ms	328ms	297ms	328ms

[4] We coded all algorithms in NetBeans 6.9.1 and performed experiments on a 2.53GHz Intel Core 2 P8700 laptop computer with 2.93 GB RAM.

We note that frequency-pruning and Cochran criteria leads to a user performance model that has a very much reduced “upper tree” and a “lower Markov model” that corresponds to the states of the application which in this case contains 30 states. The merging of non-independent states in the “upper tree” leads in our case to a further reduction of 20% of the model states. Most of the applied tests for merging succeeded. The “upper tree” is shown Figure 10.

Table 5 also shows similar results for several smaller sets of user sessions that were selected randomly from the original visiting sessions. Each subset has 10000 sessions. The results obtained for the different subsets of sessions are very similar to one another. The number of model states in the “upper tree” is smaller than in the case of all sessions, since the Cochran criterion removes more states from the TS model because of the lower number of observations. As can be seen, the model that we obtain is quite stable with different sets of sessions. The number of states in the obtained user models varies little for the different subsets of sessions. Since the “upper tree” part of the model is the most interesting, we show in Figure 11 the “upper trees” for the sets of sessions Set-1 and Set-2. One can also see that these trees closely resemble the upper part of the “upper tree” for all sessions, as shown in Figure 10. Due to the difficulty of defining a feature space and measurement method, we do not discuss the similarity between the re-constructed web sessions and the real data in this paper.

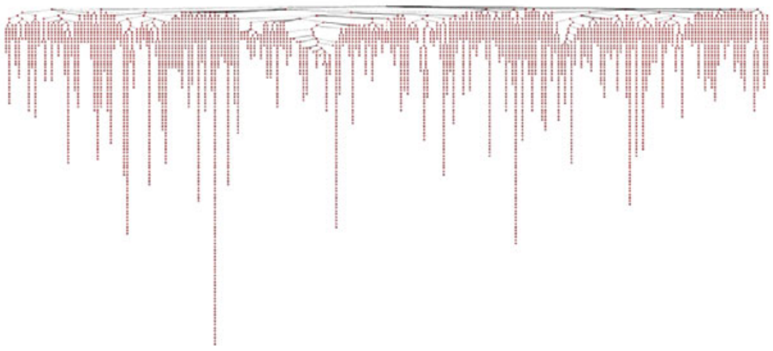


Fig. 9. The TS model created by 1000 visiting sessions

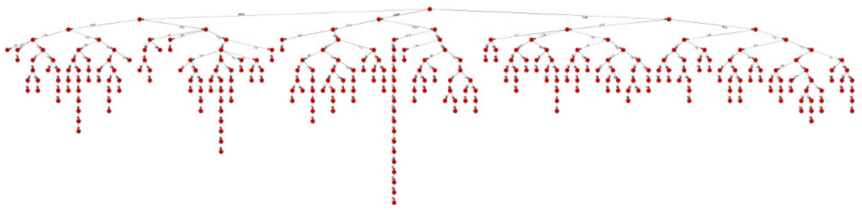


Fig. 10. The upper tree part of usage model, down from originally 348,391 states

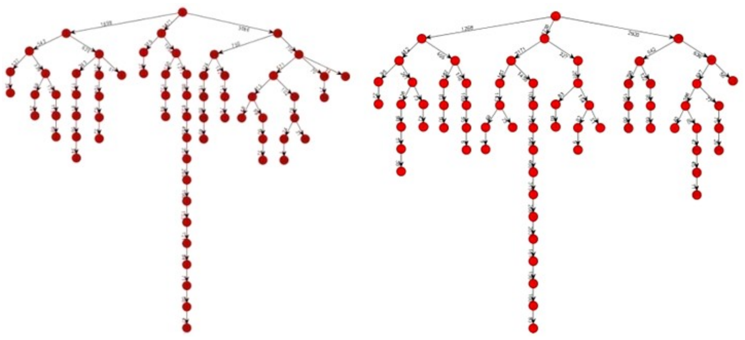


Fig. 11. Upper-tree part of usage models generated from Set-1 and Set-2

5 Conclusion and Future Work

In this paper, we have presented a method that can be used to create an accurate statistical usage model for Web applications. This model is created from the application log file, and can be used for reliability testing. Our method uses a tree structure to preserve statistically significant information on user behavior, as gathered from the log files. The initially very large tree is reduced in three steps: first, frequency pruning removes the branches that are almost never followed. Then, a test called Cochran criterion is used to remove states that do not carry reliable statistical information. States removed during these two steps are merged into a traditional Markov chain model (the "lower Markov chain") that captures infrequent behaviors. The pruned tree is further reduced through merging of model states corresponding to the same application states and on which user behavior is statistically similar. The test for similarity is a classical test of independence, and the resulting "tree", which we call the "upper tree", contains the most frequent behaviors, which are statistically significant. In our experiments, the resulting hybrid Markov usage model is drastically smaller than the original tree of sequences, but still contains all the significant behavioral and coverage information.

This improves on lower-order Markov usage models that contain usually all the application states but cannot capture the user behavior accurately since they have no concept of history. In our model, in the upper tree the entire history is preserved. Other history-preserving higher-order Markov models, such as N-grams, exist but come with their own set of limitations. For N-Grams, they do retain history of length N-1, but cannot capture sessions of length less than N [16].

Our model still has shortcomings. A main one is its inability to identify some of the common behavior, if the behavior occurs on branches that must be kept apart because they lead to statistically different behavior lower down in the tree. Indeed, our state merging process tends to merge states that are mostly toward the bottom of the tree. To overcome this, we are planning to use either extended Finite State Machine models or hierarchical models, in order to merge parts that are statistically identical but are included inside larger sequences that are not. One may also improve the model by introducing some history dependence in the "lower Markov model" by using, for instance, the N-gram approach. The other major shortcoming of our current model is that user inputs are not captured. The model must be enhanced to accommodate for a statistically accurate representation of the inputs and their relation with the followed path.

Acknowledgments. The authors wish to thank Marie-Aurélie Fund and Christophe Günst for their work on this project. This work was supported in part by grants from the Natural Sciences and Engineering Research Council of Canada.

References

1. Vilkomir, S.A., Parnas, D.L., Mendiratta, V.B., Murphy, E.: Segregated Failures Model for Availability Evaluation of Fault-Tolerant Systems. In: 29th Australasian Computer Science Conference, vol. 48 (2006)
2. Wang, W., Tang, M.: User-Oriented Reliability Modeling for a Web System. In: Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE 2003), pp. 1–12 (2003)
3. Whittaker, J.A., Thomason, M.G.: A Markov Chain Model for Statistical Software Testing. IEEE Trans. Software Eng. 20(10), 812–824 (1994)
4. Le Guen, H., Marie, R., Thelin, T.: Reliability Estimation for Statistical Usage Testing using Markov Chains. In: ISSRE 2004: Proceedings of the 15th International Symposium on Software Reliability Engineering, pp. 54–65. IEEE Computer Society, Washington, DC (2004)
5. Dulz, W., Zhen, F.: MaTeLo—statistical usage testing by annotated sequence diagrams, Markov chains, and TTCN-3. In: Proceedings of Third International Conference On Quality Software (QSIC 2003). IEEE, Los Alamitos (2003)
6. Sayre, K.: Improved Techniques for Software Testing Based on Markov Chain Usage Models. PhD thesis, University of Tennessee (1999)
7. Borges, J.: A Data Mining Model to Capture User Web Navigation. PhD thesis, University College London, London Uiversity (2000)
8. Borges, J., Levene, M.: A dynamic clustering-based Markov model for web usage mining. In: CoRR:the computing research repository. cs.IR/0406032 (2004)
9. Goseva-Popstojanova, K., Trivedi, K.S.: Failure Correlation in Software Reliability Models. IEEE Trans. on Reliability 49, 37–48 (2000)
10. Goseva-Popstojanova, K., Hamill, M.: Estimating the Probability of Failure When Software Runs Are Dependent: An Empirical Study. In: 20th International Symposium on Software Reliability Engineering, ISSRE, pp. 21–30 (2009)
11. Feliachi, A., Le Guen, H.: Generating transition probabilities for automatic model-based test generation. In: Third International Conference on Software Testing, Verification and Validation, pp. 99–102 (2010)
12. Cooley, R., Mobasher, B., Srivastava, J.: Data Preparation for Mining World Wide Web Browsing Patterns. Knowledge and Information Systems 1(1), 5–32 (1999)
13. Pei, J., et al.: Mining Access Patterns Efficiently from Web Logs. In: Proc. Pacific-Asia Conf. on Knowledge Discovery and Data Mining, pp. 396–407. Springer, New York (2000)
14. Miller, K.W., et al.: Estimating the Probability of Failure When Testing Reveals No Failures. IEEE Transactions on Software Engineering 18, 33–42 (1992)

15. Pirolli, P.L.T., Pitkow, J.E.: Distributions of surfers' paths through the world wide web: Empirical characterizations. World Wide Web, 29–45 (1999)
16. Deshpande, M., Karypis, G.: Selective Markov Models for Predicting Web-Page Accesses. In: Proc. of the 1st SIAM International Conference on Data Mining (2001)
17. Charniak, E.: Statistical Language Learning. The MIT Press, Cambridge (1996)
18. Jespersen, S., Pedersen, T.B., Thorhauge, J.: Evaluating the markov assumption for web usage mining. In: Proceeding of the Fifth International Workshop on Web Information and Data Management (WIDM 2003), pp. 82–89 (2003)
19. Walpole, R.E., Myers, R.H.: Probability and Statistics for Engineers and Scientists, 5th edn. Macmillan Publishing Company, Basingstoke (1993)

Measuring Test Properties Coverage for Evaluating UML/OCL Model-Based Tests

Kalou Cabrera Castillos[1], Frédéric Dadeau[1], Jacques Julliand[1], and Safouan Taha[2]

[1] LIFC / INRIA CASSIS Project – 16 route de Gray - 25030 Besançon cedex, France
{kalou.cabrera,frederic.dadeau,jacques.julliand}@lifc.univ-fcomte.fr
[2] SUPELEC Systems Sciences (E3S) 3 rue Joliot-Curie, 91192 Gif-sur-Yvette, France
safouan.taha@supelec.fr

Abstract. We propose in the paper a test property specification language, dedicated to UML/OCL models. This language is intended to express temporal properties on the executions of the system, that one wants to test. It is based on patterns, specifying the behaviours one wants to exhibit/avoid, and scopes, defining the piece of execution trace on which a given pattern applies. Each property is a combination of a scope and a pattern, providing a means for a validation engineer to easily express temporal properties on a system, without using complex formal notations. Properties have the semantics of an event-based transition system whose coverage can be measured so as to evaluate the relevance of a given test suite. These principles aim at being used in the context of a research project, in which the security properties are expressed on an industrial case study of a smart card operating system. This approach makes it possible to assist the Common Criteria evaluation of the testing phase, that requires evidences of the extensiveness of the testing phase of a security product.

Keywords: Model-Based Testing, UML/OCL, temporal property, coverage, model animation.

1 Introduction and Motivations

Critical software validation is a challenge in software engineering and a convenient context for Model-Based Testing [4]. Indeed, the cost of writing a formal model to support the test case generation phase is made profitable by the necessity of increasing the confidence in the safety and the security of the system. MBT is well-suited to conformance testing, as the model describes the expected behavior of a system. The system under test is then checked against the model on specific execution traces called test cases. A conformance relationship, usually based on the observation points provided by the SUT, is then used to establish the test verdict.

This work is done in the context of the ANR TASCCC project[1], we are interested in the validation of smart card products security by means of model based

[1] Funded by the French National Research Agency ANR-09-SEGI-014 – http://lifc.univ-fcomte.fr/TASCCC

tests. The tests are produced by the CertifyIt tool, provided by the Smartesting company[2]. This test generator takes as input model based tests in UML/OCL and generated tests aiming at the structural coverage of the OCL code describing the behaviors of the class operations. CertifyIt is an automated test generator, in the sense that, apart from the model, no further information is required to generate the tests.

We propose to consider user-defined test properties to express test patterns associated to the requirements of the software. The contribution is twofold. First, we propose a test property language based on Dwyer's property patterns [9] and applied to UML/OCL models. In this context, the considered events are either controllable (the invocation of an operation) or observable (a state predicate becomes satisfied at a given state). The properties describe the apparition of these events, in given scopes. Second, we propose to assist the validation engineer, by measuring the coverage of the property. Indeed, a property exhibits a set of model executions that are authorized, expressed as an automaton. In order to evaluate the exhaustiveness of the testing phase, we measure and report the coverage of the underlying automaton. In addition, uncovered parts of the property indicate which part has not been tested and, as a consequence, on which part the tester should focus his efforts.

The remainder of the paper is organized as follows. Section 2 presents the test generation process of the CertifyIt tool based on the structural coverage of the OCL code. Then, the property language is defined in Sec. 3 and its semantics is provided in Sec. 4. Section 5 defines the notion of property coverage and explains how this action is processed. Finally, we conclude and present the related and future works in Sec. 6.

2 Test Generation from UML/OCL Models

We present here the test generation principles of the CertifyIt test generation tool. First, we introduce the subset of UML/OCL that is considered and we illustrate it with a simple running example. Then, we present the test generation strategy of CertifyIt.

2.1 Considered Subset of UML/OCL

The model aims at being used by the CertifyIt tool, commercialized by the Smartesting company. This tool generates automatically model-based tests from a UML model [5] with OCL code describing the behaviors of the operations. CertifyIt does not consider the whole UML notation as input, it relies on a subset named UML4ST (UML for Smartesting) which considers class diagrams, to represent the data model, augmented with OCL constraints, to describe the dynamics of the system. It also requires the initial state of the system to be represented by an object diagram. Finally, a statechart diagram can be used to complete the description of the system dynamics.

[2] `www.smartesting.com`

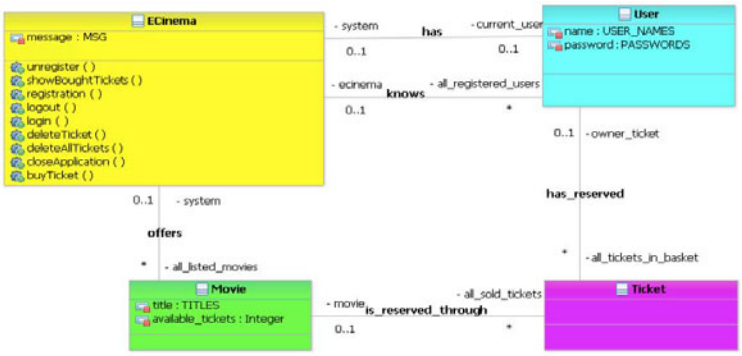


Fig. 1. Class Diagram of the eCinema Model

Concerning modelling, some restrictions apply on the class diagram model and OCL constraints that can be written. The system under test (SUT) has to be modelled by a class, which carries all the operations representing the API provided by the SUT. CertifyIt does not allow inheritance, nor stereotypes like *abstract* or *interface* on the classes. Objects can not be created when executing the model. As a consequence, the object diagram, representing the initial state, has to provide all the possible class instances, possibly isolated (i.e., not associated to the SUT object or any other object) if they are not supposed to exist at the initial state.

OCL provides the ability to navigate the model, select collections of objects and manipulate them with universal/existential quantifiers to build boolean expressions. Regarding the OCL semantics, UML4ST does not consider the third logical value *undefined* that is part of the classical OCL semantics. All expressions have to be defined at run time in order to be evaluated. CertifyIt interprets OCL expressions with a strict semantics, and raises execution errors when encountering null pointers. The overall objective is to dispose of an executable UML/OCL model. Indeed, the test cases are produced by animating the model in order to satisfy a given coverage criterion. Before describing this process, we first introduce a simple running example.

2.2 Running Example

We illustrate the UML/OCL models that are considered using a simple model of a web application named eCinema. This application provides a means for registered users to book tickets for movies that are screened in a cinema.

The UML class diagram, depicted in Fig. 1 contains the classes of the application: `ECinema`, `Movie`, `Ticket` and `User`. The `ECinema` class models the system under test (SUT) and contains the API operations offered by the application. Several requirements are associated to the system, for example: *the user must be registered and connected to access the proposed services*, *the registration is valid* *only if the user's name and password are valid and if the user is not already registered*, *the user must be connected in order to buy tickets*, etc.

The initial state contains a single instance of the system under test class eCinema, identified by `sut`, two instances of movies linked with the `sut` instance (instanciating the `offers` association), twenty-five isolated instances of tickets, and two users, one registered (i.e. linked to the SUT using the `knows` association) and one not registered (isolated instance).

This model contains several operations whose meaning are straightforward: `unregister`, `showBoughtTickets`, `registration`, `logout`, `login`, `deleteTicket`, `deleteAllTickets`, `closeApplication`, `buyTicket`.

Figure 2 shows the OCL code of the `buyTicket` operation used by an authenticated user to buy a ticket. This operation can only be invoked with a valid movie title and if all the tickets are not already assigned to users. To be successful, the user has to be authenticated, and at least one tickets for the movie should be available. Otherwise, error messages report the cause of the failure. Upon successful execution, an available ticket is chosen and assigned to the user for the corresponding movie, and the number of available seats is decremented.

In the OCL code, there are non-OCL annotations, inserted as comments, such as `---@AIM: id` and `---@REQ: id`. The `---@AIM:id` tags denotes test targets while the `---@REQ: id` tags mark requirements from the informal specifications. These tags are used by CertifyIt to know which tags were covered during the execution of the model, and, consequently, inside the test cases.

2.3 CertifyIt Test Selection Criterion

Smartesting CertifyIt is a functional test generator that aims at exercising the atomic transitions of the model, provided by the class operations. The CertifyIt

```
context ECinema::buyTicket(in_title : ECinema::TITLES): oclVoid
pre:
   self.all_listed_movies->exists(m : Movie | m.title = in_title) and
   Ticket.allInstances()->exists(t : ticket | t.owner_ticket.oclIsUndefined())
post :
   ---@REQ: BASKET_MNGT/BUY_TICKETS
   if self.current_user.oclIsUndefined() then
     message = MSG::LOGIN_FIRST      ---@AIM: BUY_Login_Mandatory
   else
      let target_movie: Movie = self.all_listed_movies->any(m: Movie | m.title = in_title) in
      if target_movie.available_tickets = 0 then
         message= MSG::NO_MORE_TICKET     ---@AIM: BUY_Sold_Out
      else
         let avail_ticket: Ticket =
            (Ticket.allInstances())->any(owner_ticket.oclIsUndefined()) in
         self.current_user.all_tickets_in_basket->includes(avail_ticket) and
         target_movie.all_sold_tickets->includes(avail_ticket) and
         target_movie.available_tickets = target_movie.available_tickets - 1 and
         message= MSG::NONE     ---@AIM: BUY_Success
      endif
   endif
```

Fig. 2. OCL code of the `buyTicket` operation

test generation strategy works in two steps. First, it identifies test targets and, second, it builds a test case that is a trace, composed of successive operation calls, reaching the considered target.

In the first part of the test generation process, the test targets are computed by applying a structural test selection criterion on the OCL code of the class operations (Decision Coverage criterion). Each test target is thus associated to a predicate that describes the set of possible concrete states from which the operation can be invoked. Each test aims at covering a specific target (possibly characterized by a set of `@AIM` tags). There are three test targets for the `buyTicket` operation, one for each available `@AIM` tags.

The second part of the test generation process consists in performing an automated state exploration, from the initial state, in order to reach a state satisfying the state predicate associated to the test target. This sequence is called a *preamble*. The preamble computation is based on a Breadth First Search algorithms that stops when the targeted state is reached. To obtain the test case, the preamble is concatenated with the invocation of the targeted operation with the appropriate parameter values.

The test selection criterion ensures that one test is built for each target. Nevertheless, some targets may be covered several times, if they are found in the preambles of others targets. Table 1 shows the tests generated for the operation `buyTicket` displayed in Fig. 1.

Notice that the undetermined state of the test $BTTest_3$ declared by CertifyIt refers to internal limitations of the tool in terms of depth search bound. Indeed, in our initial state, we specified that the two movies could deliver 20 tickets each, which needs to build a test sequence in which all 20 tickets are already bought. Since this configuration could be reached (the number of ticket instances is set to 25), this message does not conclude on the general unreachability of the test targets.

Since CertifyIt is a functional test generator, it is not intended to cover specific sequences of operations, or states. Nevertheless, the tool provides a means to complete automatically generated tests with test scenarios, built using the simulator (the internal model animator) and exported as test cases.

3 Test Property Language

The Object Constraint Language is quite similar to first-order predicate logic. OCL expressions are used in invariants, pre- and postconditions. They describe a single system state or a one-step transition from a previous state to a new state upon the call of some operation.

Table 1. Generated tests for the `buyTicket` operation

Test name	Test sequence	Target
BTTest$_1$	`init ; sut.login(REGISTERED_USER,REGISTERED_PWD) ; sut.buyTicket(TITLE1) ;`	`@AIM: BUY_Success`
BTTest$_2$	`init ; sut.buyTicket(TITLE1) ;`	`@AIM: BUY_Login_Mandatory`
BTTest$_3$	`declared as ‘‘Undetermined’’`	`@AIM: BUY_Sold_out`

Several OCL extensions already exist to support temporal constraints [8,11,16]. They only add to OCL unary and binary temporal operators (e.g., always, next and until) in order to specify safety and liveness properties. Unfortunately, most developers are not familiar with temporal logics and this is a serious obstacle to the adoption of such OCL extensions. We propose to fill in this gap by adding to OCL a pattern-based temporal layer to ease the specification of temporal properties.

3.1 A Temporal Extension to UML/OCL

For specifying the temporal aspects of system properties, we adopt the work of Dwyer et al. [9] on specification patterns for temporal properties. Although formal methods are largely automated today, most engineers are not familiar with formal languages such as linear temporal logic (e.g. LTL) or tree logic (e.g. CTL). The effort required to acquire a sufficient level of expertise in writing these specifications represents a serious obstacle to the adoption of formal methods. Therefore, Dwyer et al. have introduced a new property specification language based on patterns in which a temporal property is a combination of one pattern and one scope.

Patterns. There are 8 patterns organized under a semantic classification. We distinguish occurrence patterns from order patterns.

Occurrence patterns are: (*i*) *Absence*: an event never occurs, (*ii*) *Existence*: an event occurs at least once, (*iii*) *Bounded Existence* has 3 variants: an event occurs k times, at least k times or at most k times, and (*iv*) *Universality*: an event/state is permanent.

Order patterns are: (*v*) *Precedence*: an event P is always preceded by an event Q, (*vi*) *Response*: an event P is always followed by an event Q, (*vii*) *Chain Precedence*: a sequence of events $P_1, \ldots, P_n$ is always preceded by a sequence $Q_1, \ldots, Q_m$ (it is a generalization of the Precedence pattern), (*viii*) *Chain Response*: a sequence of events $P_1, \ldots, P_n$ is always followed by a sequence $Q_1, \ldots,$ Q_m (it is a generalization of the Response pattern).

Scopes. A scope is the discrete time interval over which the property holds. There are five kinds of scopes, illustrated on Fig.3 (taken from [9]):

(*a*) `globally` covers the entire execution, (*b*) `before` Q covers the system's execution up to the first occurrence of Q, (*c*) `after` Q covers the system's execution after the first occurrence of Q, (*d*) `between` Q `and` R covers time intervals of the system's execution from an occurrence of Q to the next occurrence of R, (*e*) `after` Q `until` R is the same as the `between` scope in which R may not occur. Dwyer et al. provide a complete library[3] mapping each pattern/scope combination to the corresponding formula in many formalisms (LTL, CTL, μ-calculus,

[3] `http://patterns.projects.cis.ksu.edu`

etc.). For example, one entry of this library that maps the *Response* pattern S `follows` P to LTL formula for different scopes is given in Tab. 2.

The work of Dwyer et al. on such patterns dramatically simplifies the specification of temporal properties, with a fairly complete coverage. Indeed, they collected hundreds of specifications and they observed that 92% fall within this small set of patterns/scopes [9]. For these reasons, we adopt this pattern-based approach for the temporal part of our OCL extension. We now present its syntax.

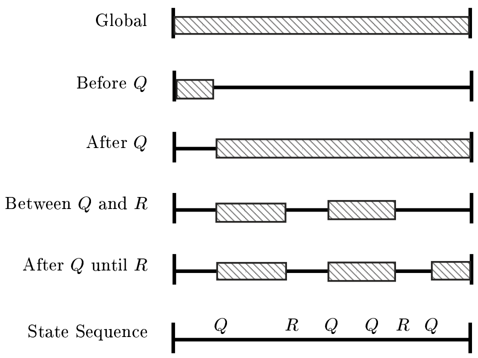


Fig. 3. Property scopes

Table 2. LTL mapping of the S `follows` P pattern

Scope	LTL
`globally`	$\Box(P \Rightarrow \Diamond S)$
`before` R	$\Diamond R \Rightarrow (P \Rightarrow (\neg R \,\mathsf{U}\, (S \wedge \neg R)))\,\mathsf{U}\, R$
`after` Q	$\Box(Q \Rightarrow \Box(P \Rightarrow \Diamond S))$
`between` Q `and` R	$\Box((Q \wedge \neg R \wedge \Diamond R) \Rightarrow (P \Rightarrow (\neg R \,\mathsf{U}\, (S \wedge \neg R)))\,\mathsf{U}\, R)$
`after` Q `until` R	$\Box(Q \wedge \neg R \Rightarrow ((P \Rightarrow (\neg R \,\mathsf{U}\, (S \wedge \neg R)))\,\mathsf{W}\, R)$

3.2 Syntax

We extended the OCL concrete grammar defined within the OMG standard [15] in order to express temporal properties that will provide our test properties.

The syntax of our language is summarized in Fig. 4. In this figure, non-terminals are designated in *italics*, terminals are underlined and construct (. . .)?

```
TempExpr ::= TempPattern TempScope

TempPattern ::= always OclExpression
              | never Event
              | eventually Event (Times)?
              | Event (directly)? precedes Event
              | Event (directly)? follows Event

TempScope ::= globally
            | before Event
            | after Event
            | between Event and Event
            | after Event until Event

Event ::= ChangeEvent ( || Event )?
        | CallEvent ( || Event )?

ChangeEvent ::= becomesTrue(OclExpression)

CallEvent ::= isCalled( (name::)? name
                  (, pre: OclExpression)?
                  (, post: OclExpression)?
                  (, TagList)? )

TagList ::= including: { Tags}
          | excluding: { Tags}

Times ::= integer times
        | at least integer times
        | at most integer times

Tags ::= @REQ: name (, Tags)?
       | @AIM: name (, Tags)?
```

Fig. 4. Syntax of our temporal property extension

designates an optional part. Terminal name designates an identifier that represents the name of a temporal property, an instance name, an operation name (when separated by `::`) or tag names (`REQ` or `AIM`, as shown in Sec. 2.3). The OCLExpression terminal designates an OCL predicate according to the supported syntax of UML4ST (as explained in Sec. 2.1).

As explained before, a temporal property *TempExpr* is a combination of a pattern *TempPattern* and a scope *TempScope* whose respective meanings have been informally described before. Events are of two kinds. A *ChangeEvent* is parameterized by an OCL predicate P, and designates a step in which P now becomes true, i.e. P was evaluated to false in the preceding step. This event represents an observable event, that is triggered after the execution of an operation of the system (but it is not possible to know *a priori* which operation will cause this event). A *CallEvent* represents the invocation of an operation on a given instance. Optional field `pre` provides a precondition, namely an OCL predicate that has to be true before the invocation of the operation (or at the beginning of the invocation if the expression refers to input parameter values). Optional field `post` provides a postcondition that has to be true after the execution of the operation. Finally, optional field `including` (resp. `excluding`) provides the set of tags for which at least one has to be covered (resp. none shall be covered) by the execution of the operation. For example, event `isCalled(sut::buyTicket, pre: in_title=TITLES::Tron and not self.current_user.oclIsUndefined(),including:{@AIM:BUY_Success})` is triggered when operation `buyTicket` is invoked on the `sut` instance, with parameter `in_title` representing a given movie title (provided as an enumeration class `TITLES`), when a user is logged on the system and the operation terminates by a successful buying of a ticket for this movie.

Example 1 (Temporal property). Let us consider the example of the eCinema application described in Sec. 2.2. We can formalize the following test requirements of the application as temporal properties. *Users can only buy tickets when logged on the system.* This statement can be expressed as a test property using a `between` scope as follows:

```
eventually isCalled(buyTicket, including:{@AIM:BUY_Success})
           at least 0 times
   between becomesTrue(not(self.current_user.isOclUndefined()))
   and becomesTrue(self.current_user.isOclUndefined()).
```

Even though the presence of `at least 0` in the pattern may be strange, it describes the optional occurrence of the event C. But, this statement may also be expressed as a robustness test property:

```
never isCalled(buyTicket, including:{@AIM:BUY_Success})
   after becomesTrue(self.current_user.isOclUndefined())
   until isCalled(login, including:{@AIM:LOGIN_Success}).
```

4 Semantics of the Test Property Language

In this section, we formally define *substitution automata* that are used to describe the semantics of the properties, and we give a definition of the substitution process. The motivation behind the use of such automata is to have a compositional semantics, so as to be able to build an automaton representing a test property. The latter is thus a combination of a scope and a pattern. Each of them is formalized using substitution automata defined in Def. 1. The resulting automaton will capture all the executions of the system and highlight specific transitions representing the events used in the property. We first define the substitution automata modelling scopes and patterns. Then we present the labels of the transitions representing the events in properties. Finally, we give the definition of the substitution that applies.

4.1 Substitution Automata

Substitution automata are labelled automata where the labels are defined from the events (see *Event* in Fig. 4). The states in S are substitution states that represent a property provided with generic patterns. They will be replaced by an automaton defining a particular pattern. For some property, such as *Pattern* `between` E_1 `and` E_2, it is necessary to avoid that E_2 is triggered in the automaton of the pattern. For that, we formalise this restriction (R) labelling the substitution state by a set of labels. These labels are events that will not be triggered by the internal transitions of the pattern automaton.

Definition 1 (Substitution automaton). *Let Σ be the set of labels. A substitution automaton is a 6-tuple $a = \langle Q, F, q_0, S, R, T\rangle$ where: Q is a finite set of states, F is a set of final states ($F \subseteq Q$), q_0 is an initial states ($q_0 \in Q$), S is a set of substitution states ($S \subseteq Q$), R is a function that associates a set of labels to any substitution state ($R \in S \to \mathcal{P}(\Sigma)$), T is a set of transitions ($T \in Q \times \mathcal{P}(\Sigma) \times Q$) labelled by a set of labels.*

Graphically, substitution states will be depicted as squares (instead of regular circles). If the substitution state presents a restriction on the alphabet, the restricted elements are written as state labels.

Scopes and patterns will be formalized as substitution automata whose labels are events that occur in the scopes and patterns. More precisely, a scope is modelled as a substitution automaton with one substitution state, whereas a pattern is modelled as a substitution automaton without substitution states.

Example 2 (Substitution Automaton for Scopes and Patterns). Figures 5 and 6 respectively give the graphical representation of the automata of the temporal property TP `between` A `and` B in which the square state represents the generic pattern TP and pattern `eventually` C `at least` 0 `times`. For the latter, the automaton clearly identifies a reflexive transition that represents the occurrence of C. Its presence originates from the motivation of coverage measure, and will make it possible to see if C has been called, or not.

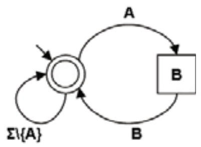


Fig. 5. `between` A `and` B

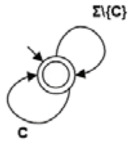


Fig. 6. `eventually` C `at least` 0 `times`

Before describing the substitution operation, we now present the alphabet of labels that is used in our automata.

4.2 Events and Labels

The events triggered in a property description and described by the rules *CallEvent* and *ChangeEvent* in the grammar described in Fig. 4 are represented by labels in automata.

The *isCalled* structure can be divided in four parts: the operation name, the precondition for this call to happen, the postcondition that has to be satisfied after the call, and a list of tags that may be activated. Thus, we will use the following notation to denote the alphabet elements in our automata:

$$[operation\ name, precondition, postcondition, tags].$$

Note that there can be unspecified slots in this notation: for example, one can specify the activation of a specific operation *op* provided with a precondition *pre*, but leaves the postcondition and tags slots free, denoted by [*op*,*pre*,_,_], in which symbol "_" designate a free slot, meaning that it can be any operation, the predicate `true` and any tag. A such label represent a set of atomic events that are triggered by the transition. For example, in example 3 the label C represent one atomic event whereas the label A represent the set of atomic events for any tag in any operation. Also, the tag list references the tags that may be activated. If the *isCalled* specifies a list of excluded tags, we can extract the set of tags that may be used (the complementary set of tags) as we know all the tags of a specific operation from the model.

The *ChangeEvent* can be seen as a free operation call with a pre- and postcondition without specifying the operation name nor the tags to be used. Notice that any operation can be called. The precondition is the negation of the `becomesTrue` predicate, illustrating the fact that, before the call, the execution of the operation has made the predicate become true. Therefore, all `becomesTrue(`P`)` events can be denoted by the label [_,¬P,P,_].

Each scope and pattern are associated to a skeleton of substitution automata whose transition labels are instantiated with the events appearing in the property that they define.

Example 3 (Labels). Consider the first property given in Example 1. According to the label notation introduced in Sec. 4.2, the labels A, B and C in

Fig. 5 and Fig. 6 are respectively [_, `self.current_user.isOclUndefined()`, `not(self.current_user.isOclUndefined())`, _], [_, `not(self.current_user. isOclUndefined())`, `self.current_user.isOclUndefined()`, _] and [`buyTicket`, _, _, {`@AIM:BUY_Success`}].

Notice that two labels can be the same set of events even if they are not written with the same 4-uplet. It is also possible that the set of atomic events of a label includes the set of atomic events of another label. In practice, these cases must represent an ambiguous test property. Therefore, we assume that all the events labelling the outgoing transitions of the automata states are mutually exclusive, producing, in that, deterministic automata.

When the two automata, the scope and the pattern, have been defined, they have to be combined to obtain a single automaton, that does not contain any substitution state. We now describe how substitution works.

4.3 Substitution

The substitution operation defined in Def. 2 replaces the substitution state s, representing a generic pattern, by an automaton as, representing an instance of a pattern, in an automaton a, representing a test property. For an automaton a, we denote by X_a the component X of a. For the definition, we assume that there is only one substitution state in a and no substitution state in as, i.e. $S_{as} = \emptyset$ and $R_{as} = \emptyset$. We also assume that the label of transitions in a and in as are different and that the set of states of a and as are disjoint.

The set of states of the resulting automaton c is the set of states of a without its substitution states S_a union each state of the substituted automaton as. When s is a final (resp. initial) state, the set of final (resp. initial) states of c is the set of a without its final (resp. initial) substitution states union each final (resp. initial) state of the substituted automaton as. Otherwise, the set of final (resp. initial) states is this of a. c contain no substitution state and consequently no restriction.

We denote by q a non-substitution state ($q \in Q - S$), s a substitution state ($s \in S$) and E a set of labels, the transitions of c are defined in four cases:

1. any transition $q \xrightarrow{E} q'$ in a is a transition of c,
2. for a transition $q \xrightarrow{E} s$ in a, there is a transition $q \xrightarrow{E} q'$ in c for the initial state of as,
3. for a transition $s \xrightarrow{E} q'$ in a, there is a transition $q \xrightarrow{E} q'$ in c for any final state q of as,
4. any transition $q \xrightarrow{E'} q'$ in as becomes a transition $q \xrightarrow{E} q'$ in c where E is the set of labels E' reduced by the labels $R(s)$.

Definition 2 (Substitution Operation). *Let a be an automaton such that $S_a = \{s\}$. Let as be an automaton such that $S_{as} = \emptyset$. The substitution of the state s by the automaton as in a is the automaton c defined as:*

- $Q_c = (Q_a - S_a) \cup Q_{as}$,
- *when* $s \in F_a$, $F_c = (F_a - \{s\}) \cup F_{as}$, *otherwise* $F_c = F_a$,
- *when* $s = q_{0_a}$, $q_{0_c} = q_{0_{as}}$, *otherwise* $q_{0_c} = q_{0_a}$,
- $S_c = \emptyset$ *and* $R_c = \emptyset$,
- $q \xrightarrow{E} q' \in T_c$ *if and only if:*
  1. $q, q' \in Q_a - S_a$ *and* $q \xrightarrow{E} q' \in T_a$,
  2. $q \in Q_a - S_a$ *and* $q \xrightarrow{E} s \in T_a$ *and* $q' = q_{0_{as}}$,
  3. $q' \in Q_a - S_a$ *and* $s \xrightarrow{E} q' \in T_a$ *and* $q \in F_{as}$,
  4. $\exists E' \in \mathcal{P}(\Sigma)$ *such that* $q \xrightarrow{E'} q' \in T_{as}$ *and* $E = E' - R(s)$.

Example 4 (Substitution Automata Composition). Consider again the scope and the property substitution automata represented in Fig. 5 and Fig. 6. Figure 7 shows the flattened automaton obtained by applying the substitution operation. This is the automaton of the property given in Example 1.

The resulting automaton represents the executions of the system allowed by the property. We now measure the coverage of this automaton to establish a metrics that will be used to evaluate the exhaustiveness of a test suite.

5 Property Coverage Measure

This section presents the technique used to measure the coverage of the property. It is based on the semantics of the property language that was previously introduced.

5.1 Automata Completion

Before performing the coverage measure, we need to complete our property automaton so as to match every possible event on our property automaton. In the example given in Fig. 7, the automaton is complete in the sense that any event will be matched from any state of the automaton. Nevertheless, in practice, the automaton is not necessarily complete. Indeed, the substitution can result in an incomplete automaton: it only represents all valid paths for the property. The complete form of the automaton thus represents all possible paths, including all faulty (with respect to the property) execution. The completion process simply

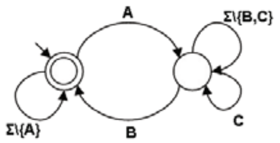


Fig. 7. Graphical representation of the composition for property `eventually` C `at least` 0 `times between` A `and` B

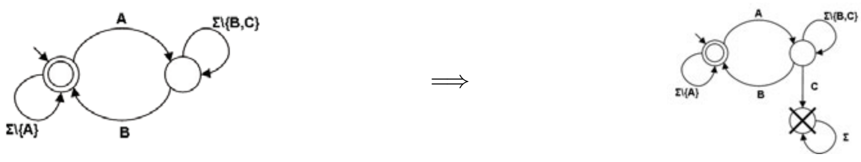


Fig. 8. Completion with rejection states (automaton for never C between A and B)

creates a new state that can be seen as a rejection state while final states represent acceptance states. If a state does not allow the triggering of a transition for an alphabet element, we create a transition from this state to the newly created state. Figure 8 illustrates this completion where the rejection state is the circle marked of one cross.

Remark. Ideally, in a Model-Based Testing process, the model describes faithfully the system, and it has no risk of violating a given (temporal) property that is supposed to hold on the system. However, in reality, the model may contain faults and thus, invalidate the property. The completion of the automaton is used to capture events that lead to an invalidate state, which detects the violation of the property. If the model satisfies the property, these additional transitions are useless, since they will never be activated on the model. Nevertheless, we add them to be able to detect possible violations of the property w.r.t. the model execution, indicating a fault in the model or in the property. Thus, we are able to partially verify (i.e. test) the model using existing test sequences, by monitoring the absence of property violation.

5.2 Performing the Measure

The evaluation of the property coverage is based on the coverage of its underlying automaton. Using the CertifyIt animation engine[4], it is possible to replay a set of existing test cases on a model. At each step (i.e. after each operation invocation), the corresponding state can be used to evaluate a given OCL predicate.

The algorithm for measuring the coverage of the property is quite straightforward, and sketched in Fig. 9. This algorithm takes as input a model M, a test suite TS and a completed substitution automaton A, supposed to represent a property. For each test, the automaton exploration starts from its (single) initial state. At each step of the test, the corresponding event on the automaton is matched. If it triggers an outgoing transition from the current state, then the exploration of the automaton progresses, and corresponding transition and states are marked by the test, and the corresponding step. When an rejection state is reached, the exploration stops and an error is returned. Once all the steps have been performed, the algorithm moves to the next test. In the end, we have, for each test, the states and transitions of the automaton reached by the test.

[4] Provided in the context of the TASCCC project by the Smartesting company

```
input Model $M$, Test Suite $TS$, Automaton $a$
begin
  for each Test $t \in TS$ do
    $cover \leftarrow false$
    $currentState \leftarrow q_{0_a}$
    mark $currentState$ as covered by $\langle t, init\rangle$
    for each Step $st$ of $t$ do
      $tr \leftarrow$ find transition triggered by step $st$
      $next \leftarrow tr.destination$
      if $next$ is a rejection state then
        throw error("Model does not respect the property");
      elseif $next \neq q_{0_a}$ then $cover \leftarrow true$
      end if
      mark $next$ state and $tr$ as covered by $\langle t, st\rangle$
      $currentState \leftarrow next$
    done
    if not $cover$ or $currentState \notin F$ then remove marking of $t$ end if
  done
end
```

Fig. 9. Coverage Measure Algorithm

When the test is replayed and the coverage of the underlying automaton is performed, three alternatives may happen: (*i*) the test leads to a rejection state, then the model does not respect the property, (*ii*) the test explores at least one transition of the automaton and reaches at least one final state (i.e. it should not stay in a initial and final state), then we say that the test "covers" the property, (*iii*) the test explores the automaton but does not reach any final state (except a final state that is also an initial state), then the test does not cover the property.

Finally, we measure classical automata coverage criteria (all-nodes, all-edges, etc.) and report the coverage of a test suite w.r.t. these criteria. Notice that only the test covering the property are considered.

Example 5 (Measure of the CertifyIt test suite coverage). As explained before, the CertifyIt test generation strategy aims at producing functional test suites. Consider the three test cases dedicated to the `buyTicket` operation, for which we want to evaluate their relevance w.r.t. the two test properties represented by the two automata depicted in Fig. 7 and Fig. 8. None of these tests do cover the property as defined in (*ii*) hereabove, as they never reach a final state (the only final state covered is also initial) in which the user disconnects from the system.

A test suite that satisfies the all-nodes and all-edges coverage criterion for the first property could be the following:

```
{ init; sut.login(REGISTERED_USER,REGISTERED_PWD); sut.showBoughtTickets();
sut.logout(), init; sut.login(REGISTERED_USER,REGISTERED_PWD);
sut.buyTicket(TITLE1); sut.logout(); }
```

6 Conclusion, Related and Future Works

We have presented in this paper a property language for UML/OCL models, based on scopes and patterns, and aiming at expressing test properties. We have proposed to evaluate the relevance of a given test suite by measuring the coverage of an automaton representing the admissible traces of the model's execution that cover the property. This approach is tool-supported (a tool prototype has been made available for the members of the TASCCC project) and experimented on the GlobalPlatform[5] case study, a last generation smart card operating system, provided by Gemalto[6].

This approach has several interesting features. First, the evaluation of a test suite is relevant w.r.t. a Common Criteria evaluation [7] of a security product which requires specific security requirements to be covered by the test cases during the validation phase. Second, the uncovered parts of the properties that have been discovered indicate precisely on which part of the system the validation engineer has to focus to complete the existing test suite. In addition, the existing tests can be used to identify relevant pieces of model executions that make it possible to reach a given state in the property automaton, helping the validation engineer to design complementary test cases, that do not only rely on its own interpretation of the property.

Related works. This approach is inspired from property monitoring approaches. Many related works fall into the category of passive testing, in which properties are monitored on a system under test [2,1]. This kind of approach is particularly used in security testing, where the violation of security properties can be detected at run-time and strengthen the test verdict [3]. The closest work is reported in [10] which uses a similar approach, also based on Dwyer's property patterns and the classification of occurrence/precedence patterns, in order to monitor test properties. Our approach differs in the sense that we aim at evaluating test cases w.r.t. properties. Also, [14] proposes the generation approach of relevant test sequences from UML statecharts guided by temporal properties. A test relevance criterion is also defined. In [13], temporal properties written in Java Temporal Pattern Language (also inspired by Dwyer's patterns) are designed and translated into JML annotations that are monitored during the execution of Java classes.

Future works. We are currently investigating the way of building these missing test cases automatically using a Scenario-Based Testing approach [6]. In addition, we are also looking for the automated generation of robustness test cases, to be extracted from these user-defined test properties, by using a mutation based testing approach applied to the property, also coupled with a Scenario-Based Testing approach. Another extension of this work will be to define dedicated temporal property coverage criteria e.g. inspired from [12].

References

1. Arnedo, J.A., Cavalli, A., Núñez, M.: Fast testing of critical properties through passive testing. In: Hogrefe, D., Wiles, A. (eds.) TestCom 2003. LNCS, vol. 2644, pp. 295–310. Springer, Heidelberg (2003)

[5] `www.globalplatform.org/specifications.asp`

[6] `www.gemalto.com`

2. Ayache, J.M., Azema, P., Diaz, M.: Observer: a concept for on-line detection of control errors in concurrent systems. In: 9th Sym. on Fault-Tolerant Computing (1979)
3. Bayse, E., Cavalli, A., Núñez, M., Zaidi, F.: A passive testing approach based on invariants: application to the wap. Computer Networks 48, 247–266 (2005)
4. Beizer, B.: Black-Box Testing: Techniques for Functional Testing of Software and Systems. John Wiley & Sons, New York (1995)
5. Bouquet, F., Grandpierre, C., Legeard, B., Peureux, F., Vacelet, N., Utting, M.: A subset of precise UML for model-based testing. In: A-MOST 2007, 3rd Int. Workshop on Advances in Model Based Testing, pp. 95–104. ACM, London (2007)
6. Cabrera Castillos, K., Dadeau, F., Julliand, J.: Scenario-based testing from UML/OCL behavioral models – application to POSIX compliance. STTT, International Journal on Software Tools for Technology Transfer (2011); Special Issue on Verified Software: Tools, Theory and Experiments (VSTTE 2009) (to appear)
7. Common Criteria for Information Technology Security Evaluation, version 3.1. Technical Report CCMB-2009-07-001 (July 2009)
8. Cengarle, M.V., Knapp, A.: Towards OCL/RT. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 390–408. Springer, Heidelberg (2002)
9. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: ICSE 1999: Proceedings of the 21st International Conference on Software Engineering, pp. 411–420. IEEE Computer Society Press, Los Alamitos (1999)
10. Falcone, Y., Fernandez, J.-C., Jéron, T., Marchand, H., Mounier, L.: More testable properties. In: Petrenko, A., Simão, A., Maldonado, J.C. (eds.) ICTSS 2010. LNCS, vol. 6435, pp. 30–46. Springer, Heidelberg (2010)
11. Flake, S.: Mueller: Formal Semantics of Static and Temporal State-Oriented OCL Constraints. Software and Systems Modeling (SoSyM) 2, 186 (2003)
12. Fraser, G., Wotawa, F.: Complementary criteria for testing temporal logic properties. In: Dubois, C. (ed.) TAP 2009. LNCS, vol. 5668, pp. 58–73. Springer, Heidelberg (2009)
13. Giorgetti, A., Groslambert, J., Julliand, J., Kouchnarenko, O.: Verification of class liveness properties with Java modeling language. IET Software 2(6), 500–514 (2008)
14. Li, S., Qi, Z.-C.: Property-oriented testing: An approach to focusing testing efforts on behaviours of interest. In: Beydeda, S., Gruhn, V., Mayer, J., Reussner, R., Schweiggert, F. (eds.) Proceedings of SOQUA 2004, Testing of Component-Based Systems and Software Quality, LNI, vol. 58, pp. 191–206. GI (2004)
15. Object Management Group. Object Constraint Language (February 2010), http://www.omg.org/spec/OCL/2.2
16. Ziemann, P., Gogolla, M.: An OCL Extension for Formulating Temporal Constraints. Technical report, Universität Bremen (2003)

Conformance Relations for Distributed Testing Based on CSP

Ana Cavalcanti[1], Marie-Claude Gaudel[2], and Robert M. Hierons[3]

[1] University of York, UK
[2] LRI, Université de Paris-Sud and CNRS, France
[3] Brunel University, UK

Abstract. CSP is a well established process algebra that provides comprehensive theoretical and practical support for refinement-based design and verification of systems. Recently, a testing theory for CSP has also been presented. In this paper, we explore the problem of testing from a CSP specification when observations are made by a set of distributed testers. We build on previous work on input-output transition systems, but the use of CSP leads to significant differences, since some of its conformance (refinement) relations consider failures as well as traces. In addition, we allow events to be observed by more than one tester. We show how the CSP notions of refinement can be adapted to distributed testing. We consider two contexts: when the testers are entirely independent and when they can cooperate. Finally, we give some preliminary results on test-case generation and the use of coordination messages.

1 Introduction

As a notation for refinement, CSP has well understood models and associated model-checking techniques and tools [14]. Testing using CSP specifications, however, has not been widely studied yet. In [3], Cavalcanti and Gaudel present a CSP framework for testing against refinement, with a unique tester that has visibility of all interactions with the system under test. In this paper, we investigate the effect of having distributed testers with limited or no global observation.

Distributed and shared systems are increasingly common, but are difficult to observe and control globally; this raises difficulties for testing them. Here, we address these issues for testing based on CSP, in the line of works by Jard et al. [12,13,11], Ural and Williams [16], and Hierons and Nunez [7,8].

To formalise the fact that multiple independent users have a weaker power of observation than a centralised user, it is necessary to state adequate weaker notions of refinement, as proposed for CSP by Jacob [10], or similarly weaker conformance relations, as developed as alternatives for the well-known *ioco* relation in [7,9]. This paper studies such refinements relations for CSP.

First, we consider cooperating refinement, where there is a possibility of collectively checking the observations at some points (namely, after complete runs). Second, we study independent refinement, where there is no way for synthesizing observations. The notions of cooperating and independent refinement have been introduced in [10] for a general unspecified notion of observation. Here, we instantiate these relations to obtain CSP versions of the conformance relations studied for Input/Output Transition Systems (IOTSs) in [7,8].

We relate the notion of independent refinement to that of lazy abstraction in [14]. In this way, we reveal the natural instantiation of independent refinement when the observations are failures; this is not covered in [7,8]. Via a number of examples, we explore the properties of the relations introduced here. Finally, we briefly consider test generation. The approach previously devised for CSP [3] can be adapted, but the resulting test cases need not be sound. We then show how the use of coordination messages suggested by Hierons in [5] can be adapted to CSP to produce sound test cases that establish traces refinement.

The paper is organised as follows. In the next section, we give an overview of CSP and the existing work on distributed testing for IOTSs. Section 3 introduces and discusses our proposed definitions of cooperating, and independent traces and failures refinement. In Section 4, we consider coordination messages. We draw our conclusions, and discuss future work in our final Section 5.

2 Preliminaries

We cover aspects of CSP, and relevant results on distributed testing for IOTSs.

2.1 CSP

In CSP, systems (and their components) are modelled as processes that interact synchronously with each other and their environment via events representing communications over channels. The set of (external) events in which a process P can engage is denoted αP. Sets of events are called alphabets.

The process $STOP$ is deadlocked, $SKIP$ terminates immediately, and $a \rightarrow P$ can engage in the event a, and then behave like the process P. An external choice $P_1 \mathrel{\Box} P_2$ offers to its environment the behaviour of either P_1 or P_2; the choice is determined by the first event on which the environment synchronises. An internal choice $P_1 \sqcap P_2$ is nondeterministic; it can behave as either P_1 or P_2.

Processes can also be combined in parallel. We use the alphabetised parallelism: $P_1 \mathrel{[\![A]\!]} P_2$, which executes P_1 and P_2 concurrently, requiring that they synchronise on the events in the set A. We also use the replicated parallel operator $\parallel i : I \bullet [A(i)]P(i)$, where the processes $P(i)$ with alphabet $A(i)$, for i in the indexing set I, are run in parallel, synchronising on their common channels.

Events can be external, that is, observable and controllable by the environment, or internal. Using the hiding operator, like in $P \setminus A$, we define a process that behaves like P, but whose events in the set A are internal.

CSP has three standard semantic models: the traces, the (stable) failures, and the failures-divergences models. In the traces model, a process P is characterised by its set $traces(P)$ of traces t of type $\operatorname{seq} \Sigma^{\checkmark}$. These are finite sequences of events in which it can engage. The special event $\checkmark$ records termination.

The empty trace is $\langle\rangle$. The set of all events, excluding $\checkmark$, is Σ; the set with $\checkmark$ is $\Sigma^{\checkmark}$. The set $traces(P)$, for process P, is prefix closed: if a process can engage in a given sequence of events, then it can engage in all its prefixes.

For a trace t of a process P and a subset R of $\Sigma^{\checkmark}$, the pair (t, R) is a failure for P if, and only if, after performing t, P may refuse all events of R. The set $failures(P)$ containing all failures of P is subset closed: if P may deadlock when the choice among events in a set R is proposed by its environment after a trace t, it may deadlock as well if only a subset of R is proposed.

The set $divergences(P)$ contains the traces of P that lead to an infinite sequence of internal events. The canonical semantics of CSP is based on its failures-divergences model $\mathcal{N}$, where the set of traces is determined by the set of *failures*. There are also CSP models that record the set $infinites(P)$ of infinite traces of P. They capture unbounded nondeterminism more accurately [14].

As usual, we assume that specifications and systems are divergence free. A divergent specification is necessarily a mistake. Also, when testing, divergences raise problems of observability. Therefore, we identify divergence with deadlock.

In the traces model, a process P is defined to be trace refined by a process Q, that is, $P \sqsubseteq_T Q$, if, and only if, $traces(Q) \subseteq traces(P)$. For divergence-free processes, the notion of refinement in the canonical model $\mathcal{N}$ of CSP is failures refinement $P \sqsubseteq_F Q$, which requires $failures(Q) \subseteq failures(P)$. For the model of infinite traces, we consider $P \sqsubseteq_\infty Q$, which, when P and Q are divergence-free, also requires reverse containment of (both finite and infinite) traces.

All these models and notions of refinement are based on the possibility of global observations of the system. Later, in Section 3, we consider distribution.

2.2 Distributed Testing for IOTS

Most work on formal distributed testing concerns testing from a Deterministic Finite State Machines (DFSM). While DFSMs are suitable for specifying some classes of systems, they require that the set of states is finite and that input and output alternate. In addition, many distributed systems are nondeterministic. There has been, thus, interest in distributed testing from an IOTS [1].

In this paper we build on recent work that defines conformance relations for distributed testing from an IOTS [7,6]. It considers two scenarios. In the first, the testers are independent in that no external agent can receive information from more than one of them. Here, it is sufficient that the local behaviour observed by a tester is consistent with a trace of the specification.

The implementation relation **p-dioco** is based on this idea; it requires that for each finite trace σ of the implementation and tester p there is a trace σ' of the specification such that the projections of σ and σ' at p are identical. An important characteristic of **p-dioco** is that given a trace σ of the implementation, the trace σ' that the specification uses to simulate it can vary with the tester.

In the second scenario, there is the possibility that information from two or more testers can be received by an external agent. As a result, the local behaviours observed by the testers could be brought together and so a stronger implementation relation **dioco** is introduced.

Work on DFSM has identified the controllability problem, which occurs when a test leads to a situation where a tester does not know when to apply an input [15]. As an example, we consider a test that starts with an input $?i_1$ that is to be applied by a tester 1 and lead to an output $!o_1$ for 1, after which the input $?i_2$ should be applied by a tester 2. The problem is that 2 does not observe $?i_1$ or $!o_1$, and so does not know when to apply $?i_2$.

The reduction in observational power also affects the ability of distinguishing between a trace of the specification and a trace of the implementation during testing. This has been called an observability problem [4].

Generating tests without controllability problems restricts testing. An alternative is to overcome these problems through the exchange of coordination messages between testers [2]. It has been shown that, when testing from an IOTS, coordination messages can be used to allow global traces to be observed, but this requires several coordination messages for each event [12]. Recent work has shown how fewer messages can be added to a test sequence [5] to overcome controllability problems. It is this approach that we adapt.

3 Distributed Testing for CSP

In this section, we define for CSP relations corresponding to **dioco** and **p-dioco**; more precisely, we define notions of cooperating and independent refinement. We consider observations of both traces and failures, but not divergences.

Our work builds on that presented in [10], which considers notions of refinement for CSP processes, when the environment consists of several users. It proposes general notions of cooperating and independent refinement that we instantiate here for the observations of interest in the scenarios studied in [7]. In [10] they are used to define traces-based refinement for transactions.

We characterise users U by non-empty sets of events. Unlike [10], we do not assume that these sets are disjoint, and actually require that $\checkmark$ can be observed by all users. Additionally, to allow the use of synchronous coordination messages in testing experiments, users need to have non-disjoint alphabets. We use $\mathcal{A}$ to denote the finite set of all users and assume that $\bigcup \mathcal{A} = \Sigma^{\checkmark}$. In examples we do not explicitly list the event $\checkmark$ when defining a user, since it is always included.

3.1 Cooperating Refinement

Like **dioco**, cooperating refinement caters for a scenario in which the observations of the various users are reconciled at some point. This means that the users can compare their observations, and what is checked is that, collectively, their observations can account for one single behaviour of the process.

If the users get together too soon, or compare their observation at different stages of the interaction, then inappropriate distinctions can be made.

Example 1. We consider the specification $P = a \rightarrow b \rightarrow STOP$ and users $\{a\}$ and $\{b\}$. If we have an accurate implementation, $\{a\}$ observes the traces $\langle\rangle$ and $\langle a \rangle$. The traces for $\{b\}$ are $\langle\rangle$ and $\langle b \rangle$. If $\{a\}$ observes $\langle a \rangle$, and $\{b\}$ observes $\langle\rangle$, then in comparing their observations we have the global trace $\langle a \rangle$. We cannot, however, compare all pairs of traces observed by $\{a\}$ and $\{b\}$. There is, for instance, no global trace corresponding to $\langle\,\rangle$ from $\{\,a\,\}$, and $\langle b \rangle$ from $\{b\}$. $\square$

Complete runs. There are various ways of identifying the observations that are of interest for comparison. Here, we pursue the solution proposed by the **dioco** relation, which is based on the notion of a complete run.

For CSP, we define the set $\mathcal{C}(P)$ of complete runs of a process P in the infinite-traces model. Namely, the complete runs are those characterised by traces that record a termination, lead to a deadlock, or are infinite. The need for infinite traces is justified, for instance, by the process $P = a \rightarrow P$, which does not terminate or deadlock. If we consider the model $\mathcal{N}$, P has no complete runs.

Definition 1 (Complete run)

$$\mathcal{C}(P) \mathrel{\widehat{=}} \{\, t : \mathit{traces}(P) \mid \mathrm{last}\, t = \checkmark \vee (t, \Sigma^{\checkmark}) \in \mathit{failures}(P)\,\} \cup \mathit{infinites}(P)$$

For any finite sequence s, we use $\mathrm{last}\, s$ to denote its last element.

Local equivalence. Cooperating refinement is based on a notion of local equivalence for traces. For traces s and t, we write $s \sim_T t$ if s and t are locally trace equivalent (with respect to the set of users $\mathcal{A}$). This means that the set of individual observations of the users in $\mathcal{A}$ cannot distinguish s from t.

Definition 2 (Local trace equivalence)

$$s \sim_T t \mathrel{\widehat{=}} (\forall\, U : \mathcal{A} \bullet \pi_U(s) = \pi_U(t))$$

where, for every trace t and user U, $\pi_U(t) = t \upharpoonright U$.

The sequence $s \upharpoonright F$ is that obtained from s by removing all elements not in F.

It does not make sense to define a similar failure-based equivalence, since we only consider complete runs. All sets of events are refused after termination or a deadlock, and there are no failures for infinite traces.

Definition and properties. Using the notion of complete run, we define cooperating traces refinement as a direct instantiation of the definition in [10].

Definition 3 (Cooperating traces refinement)

$$P \sqsubseteq_{CT} Q \mathrel{\widehat{=}} \forall\, s : \mathcal{C}(Q) \bullet \exists\, t : \mathcal{C}(P) \bullet s \sim_T t$$

A process P is cooperating refined by Q if, for every complete run of Q there is a complete run of P that is local trace equivalent to it.

Example 2. The only complete run of $P = a \rightarrow b \rightarrow \mathit{SKIP}$ is $\langle a, b, \checkmark \rangle$. The complete runs of $Q = a \rightarrow b \rightarrow \mathit{SKIP} \,\Box\, b \rightarrow a \rightarrow \mathit{SKIP}$ are $\langle a, b, \checkmark \rangle$ and $\langle b, a, \checkmark \rangle$. If we consider users $\{a\}$ and $\{b\}$, then $\langle a, b, \checkmark \rangle$ is locally equivalent to $\langle b, a, \checkmark \rangle$. Therefore, not only $Q \sqsubseteq_{CT} P$, but also $P \sqsubseteq_{CT} Q$. In other words, P and Q are equal from the point of view of cooperating refinement. This reflects the fact that the users do not have a record of the time in which their observations are made, and so cannot compare their relative order. $\square$

It is not difficult to see that, in general, $P \sqsubseteq_{\infty} Q$ implies $P \sqsubseteq_{CT} Q$, since in this case all observations (traces, including the infinite ones, and failures) of Q are also observations of P. This, of course, includes the complete runs of Q.

Traces refinement, however, does not entail cooperating refinement.

Example 3. The processes $P = a \rightarrow STOP \sqcap STOP$ and $Q = a \rightarrow STOP$ are equal from the point of view of traces refinement. On the other hand, $\mathcal{C}(P) = \{\langle\,\rangle, \langle a\rangle\}$ and $\mathcal{C}(Q) = \{\langle a\rangle\}$. Since $\langle\,\rangle$ has no equivalent in $\mathcal{C}(Q)$ for a user $\{a\}$, we have that $P \sqsubseteq_{CT} Q$, but not $Q \sqsubseteq_{CT} P$. □

Cooperating refinement, and all other relations presented here, are not compositional. They are, therefore, in general not amenable to compositional analysis.

Example 4. The processes $P = a \rightarrow b \rightarrow STOP$ and $Q = b \rightarrow a \rightarrow STOP$ are equal (for users $\{a\}$ and $\{b\}$) from the point of view of cooperating refinement. Their complete runs $\mathcal{C}(P) = \{\langle a, b\rangle\}$ and $\mathcal{C}(Q) = \{\langle b, a\rangle\}$ are locally equivalent. We consider, however, the context defined by the process function F below.

$$F(X) = (X \;[\![\, \{a, b\} \,]\!]\; b \rightarrow STOP)$$

We have that $F(P) = STOP$ and $F(Q) = b \rightarrow STOP$. From the point of view of a user $\{b\}$, these processes can be distinguished. □

Lack of compositionality restricts the opportunities of practical (and scalable) use of our relations for development and analysis. For testing of complete systems, however, this is not an issue, and we expect that certain architectural patterns ensure compositionality. This will be considered in our future work.

Since local equivalence is transitive, so is cooperating refinement.

3.2 Independent Refinement

The scenario considered in independent refinement is similar to that in **p-dioco**, namely, a situation in which the users do not have a way of comparing their observations. Here, we consider both observations of traces and failures.

Independent traces refinement. The **p-dioco** relation is the inspiration for what we call here independent traces refinement, and define as follows.

Definition 4 (Independent traces refinement)

$$P \sqsubseteq_{IT} Q \mathrel{\widehat{=}} (\forall\, U : \mathcal{A};\; s : traces(Q) \bullet (\exists\, t : traces(P) \bullet \pi_U(s) = \pi_U(t)))$$

For every user U and trace s of Q, we require there to be a trace t of P such that U cannot distinguish between s and t. This is different from cooperating traces refinement, where we require the existence of a single corresponding trace t in P that cannot be distinguished from s from the point of view of all users.

Example 5. The processes $P = a.1 \rightarrow b.1 \rightarrow STOP \,\Box\, a.2 \rightarrow b.2 \rightarrow STOP$ and $Q = a.1 \rightarrow b.2 \rightarrow STOP \,\Box\, a.2 \rightarrow b.1 \rightarrow STOP$ are different under cooperating refinement with users $\{a.1, a.2\}$ and $\{b.1, b.2\}$. For instance, the complete run $\langle a.1, b.2 \rangle$ of Q is not equivalent to any of the complete runs $\langle a.1, b.1 \rangle$ and $\langle a.2, b.2 \rangle$ of P. In the case of $\langle a.1, b.1 \rangle$, the user $\{b.1, b.2\}$ can make a distinction, and for $\langle a.2, b.2 \rangle$, the user $\{a.1, a.2\}$ can detect a distinction.

Under independent (traces) refinement, however, P and Q cannot be distinguished, because there is no opportunity for the users to compare their observations. For example, for the trace $\langle a.1, b.2 \rangle$ of Q, we have in P the trace $\langle a.1 \rangle$ for the user $\{a.1, a.2\}$, and the trace $\langle a.2, b.2 \rangle$ for the user $\{b.1, b.2\}$. $\Box$

The processes P and Q, and process function F in Example 4 also provide an example to show that independent traces refinement is not compositional. Namely $P =_{IT} Q$, but not $F(P) =_{IT} F(Q)$; as before $F(P)$ and $F(Q)$ can be distinguished by $\{b\}$, with no trace in $F(P)$ corresponding to $\langle b \rangle$.

Lazy abstraction. The notion of independence is related to that of lazy abstraction [14, page 297]. In that work, it is discussed under the assumption that the processes are divergence-free and nonterminating, that the nondeterminism is bounded, and that users are disjoint. In what follows, we establish the relationship between lazy abstraction and independent traces refinement. Afterwards, we use that as inspiration to define independent failures refinement.

Following [14], we define the process $P@U$, which characterises the behaviour of P for a user U. Below, we define the traces and stable failures of $P@U$.

In considering the independent behaviour of P from the point of view of U, we observe that the behaviour of other users might affect the perception that U has of P. First of all, there is the possibility of the introduction of deadlock. If, for example, U is waiting for an event b that is only available after P engages in an event a that is not under the control of U, then U may experience a deadlock. This is because the users that control a may not agree on that event.

A second aspect is related to divergences. Like in [14], we assume that divergence is not introduced, even if P offers an infinite trace of events of a user different from U, and therefore an infinite trace of events effectively hidden from U. This means that we assume that no user is fast enough to block P, or that P is fair. As a consequence, what we reproduce below is the canonical failures-divergences model of $P@U$ if P, and therefore, $P@U$ are divergence-free [14].

Definition 5 ($P@U$)

$$traces(P@U) \mathrel{\widehat{=}} \{\, t : traces(P) \bullet \pi_U(t) \,\}$$
$$failures(P@U) \mathrel{\widehat{=}} \{ t : \mathrm{seq}\, \Sigma^{\checkmark};\; A : \mathbb{P}\, \Sigma^{\checkmark} \mid (t, A \cap U) \in failures(P) \bullet (\pi_U(t), A) \,\}$$

The set $traces(P@U)$ contains the traces $\pi_U(t)$ obtained by removing from a trace t of P all events not in U. The alphabet of refusals, on the other hand, is $\Sigma^{\checkmark}$. (This allows us to compare the views of the different users, and the view of a user with that of the system.) Therefore, the failures $(\pi_U(t), A)$ of $P@U$ are obtained by considering the failures $(t, A \cap U)$ of P. For the trace t, we consider

$\pi_U(t)$, as already explained. For the refusal $A \cap U$, we observe that if $A \cap U$ is refused by P, then A, which can contain events not in U, is refused by $P@U$. Since an event not in U cannot be observed by U, it is refused by $P@U$.

Example 6. We consider the process $P = a \rightarrow b \rightarrow STOP$, and a user $\{b\}$. The set $traces(P@\{b\})$ is $\{\langle\rangle, \langle b\rangle\}$. The traces $\langle a\rangle$ and $\langle a, b\rangle$ of P are not (entirely) visible to $\{b\}$. The failures of $P@\{b\}$, on the other hand, include $(\langle\,\rangle, \{a, b\})$ (and so, all subsets of $\{a, b\}$). This indicates that, from the point of view of $\{b\}$, the process can deadlock, since interaction with a may not go ahead. □

The following lemma states that independent trace refinement holds when all independent users observe a traces refinement.

Lemma 1. $P \sqsubseteq_{IT} Q \Leftrightarrow \forall\, U : \mathcal{A} \bullet (P@U) \sqsubseteq_T (Q@U)$

Proof.

$$\forall\, U : \mathcal{A} \bullet (P@U) \sqsubseteq_T (Q@U)$$

$$\Leftrightarrow \forall\, U : \mathcal{A} \bullet traces(Q@U) \subseteq traces(P@U) \qquad [\text{definition of } \sqsubseteq_T]$$

$$\Leftrightarrow \forall\, U : \mathcal{A} \bullet (\forall\, s : traces(Q@U) \bullet (\exists\, t : traces(P@U) \bullet s = t)) \quad [\text{property of sets}]$$

$$\Leftrightarrow \begin{pmatrix} \forall\, U : \mathcal{A} \bullet (\forall\, s : \operatorname{seq} \Sigma^{\checkmark} \mid (\exists\, so : traces(Q) \bullet s = \pi_U(so)) \bullet \\ \quad (\exists\, t : \operatorname{seq} \Sigma^{\checkmark};\ to : traces(P) \bullet t = \pi_U(to) \wedge s = t)) \end{pmatrix}$$

$$[\text{definition of } traces(P@U) \text{ and } traces(Q@U)]$$

$$\Leftrightarrow \begin{pmatrix} \forall\, U : \mathcal{A} \bullet (\forall\, s : \operatorname{seq} \Sigma^{\checkmark} \mid (\exists\, so : traces(Q) \bullet s = \pi_U(so)) \bullet \\ \quad (\exists\, to : traces(P) \bullet s = \pi_U(to))) \end{pmatrix}$$

$$[\text{one-point rule}]$$

$$\Leftrightarrow \forall\, U : \mathcal{A};\ so : traces(Q) \bullet (\exists\, to : traces(P) \bullet \pi_U(so) = \pi_U(to))$$

$$[\text{one-point rule}]$$

$$\Leftrightarrow P \sqsubseteq_{IT} Q \qquad [\text{definition of } P \sqsubseteq_{IT} Q]$$

□

It is a straightforward consequence of the above lemma that independent traces refinement is transitive, since traces refinement is transitive.

Example 7. We consider again $P = a.1 \rightarrow b.1 \rightarrow STOP \,\Box\, a.2 \rightarrow b.2 \rightarrow STOP$ and $Q = a.1 \rightarrow b.2 \rightarrow STOP \,\Box\, a.2 \rightarrow b.1 \rightarrow STOP$. So, $traces(P@\{a.1, a.2\})$ is $\{\langle\rangle, \langle a.1\rangle, \langle a.2\rangle\}$ and $traces(P@\{b.1, b.2\})$ is $\{\langle\rangle, \langle b.1\rangle, \langle b.2\rangle\}$. These are also the traces of $Q@\{a.1, a.2\}$ and $Q@\{b.1, b.2\}$, as expected from our earlier conclusion that P and Q are indistinguishable under independent traces refinement. □

Example 8. The processes $P = a \rightarrow b \rightarrow STOP$ and $Q = b \rightarrow a \rightarrow STOP$ are not related by traces refinement. As we have already seen, they cannot be distinguished under cooperating refinement with users $\{a\}$ and $\{b\}$. It turns out that these processes are also equal under independent traces refinement. This is because $traces(P@\{a\}) = \{\,\langle\rangle, \langle\, a\rangle\,\}$ and $traces(P@\{b\}) = \{\,\langle\rangle, \langle\, b\rangle\,\}$, and the same holds if we consider Q instead of P. This again reflects the fact that, in isolation, $\{a\}$ and $\{b\}$ cannot decide in which order the events occur. □

Independent failures refinement. The definition of $P \sqsubseteq_{IT} Q$ is inspired by [10], and it is interesting that it is similar to the definition of **p-dioco** [7]. Lemma 1 indicates the way for considering also independent failures refinement.

Definition 6 (Independent failures refinement)

$$P \sqsubseteq_{IF} Q \mathrel{\widehat=} \forall\, U : \mathcal{A} \bullet (P @ U) \sqsubseteq_F (Q @ U)$$

Example 9. Independent failures refinement does not hold (in either direction) for the processes P and Q in Example 8. Intuitively, this is because, from the point of view of $\{a\}$, P is immediately available for interaction, and then deadlocks. On the other hand, Q may deadlock immediately, if b does not happen. Similarly, from the point of view of $\{b\}$, P may deadlock immediately, but not Q. Accordingly, the failures of P and Q for these users are as sketched below.

$$\begin{aligned}
&\mathit{failures}(P @ \{a\}) = \{\, (\langle\,\rangle, \{b, \checkmark\}), \ldots, (\langle a \rangle, \{a, b, \checkmark\}), \ldots \}\\
&\mathit{failures}(P @ \{b\}) = \{\, (\langle\,\rangle, \{a, b, \checkmark\}), \ldots, (\langle b \rangle, \{a, b, \checkmark\}), \ldots \}\\
&\mathit{failures}(Q @ \{a\}) = \{\, (\langle\,\rangle, \{a, b, \checkmark\}), \ldots, (\langle a \rangle, \{a, b, \checkmark\}), \ldots \}\\
&\mathit{failures}(Q @ \{b\}) = \{\, (\langle\,\rangle, \{a, \checkmark\}), \ldots, (\langle b \rangle, \{a, b, \checkmark\}), \ldots \}
\end{aligned}$$

We omit the failures that can be deduced by the fact that these sets are subset closed. Deadlock is characterised by a failure whose refusal has all events. □

Example 10. For an example where independent failures refinement holds, consider users $\{\, \{a\}, \{b\}, \{c.1, c.2\} \,\}$, $P = a \to c.1 \to \mathit{STOP} \mathrel{\Box} b \to c.2 \to \mathit{STOP}$, and $Q = a \to c.2 \to \mathit{STOP} \mathrel{\Box} b \to c.1 \to \mathit{STOP}$. We have the following.

$$\begin{aligned}
&\mathit{traces}(P @ \{a\}) = \mathit{traces}(Q @ \{a\}) = \{\, \langle\,\rangle, \langle a \rangle \,\}\\
&\mathit{traces}(P @ \{b\}) = \mathit{traces}(Q @ \{b\}) = \{\, \langle\,\rangle, \langle b \rangle \,\}\\
&\mathit{traces}(P @ \{c.1, c.2\}) = \mathit{traces}(Q @ \{c.1, c.2\}) = \{\, \langle\,\rangle, \langle c.1 \rangle, \langle c.2 \rangle \,\}
\end{aligned}$$

Regarding refusals, for both P and Q, the view of $\{a\}$ is that the process may nondeterministically choose between deadlocking (if b reacts "more quickly" and takes the choice) or doing an a. The situation for $\{b\}$ is similar. Finally, for $\{c.1, c.2\}$, there is a nondeterministic choice between a deadlock, if neither a nor b happens, or carrying out a $c.1$ or a $c.2$ and then deadlocking. Accordingly, the failures obtained from P and Q are the same; they are sketched below.

$$\begin{aligned}
&\mathit{failures}(P @ \{a\}) = \mathit{failures}(Q @ \{a\}) =\\
&\quad \{\, (\langle\,\rangle, \{a, b, c.1, c.2, \checkmark\}), \ldots (\langle a \rangle, \{a, b, c.1, c.2, \checkmark\}), \ldots \}\\
&\mathit{failures}(P @ \{b\}) = \mathit{failures}(Q @ \{b\}) =\\
&\quad \{\, (\langle\,\rangle, \{a, b, c.1, c.2, \checkmark\}), \ldots (\langle b \rangle, \{a, b, c.1, c.2, \checkmark\}), \ldots \}\\
&\mathit{failures}(P @ \{c.1, c.2\}) = \mathit{failures}(Q @ \{c.1, c.2\}) =\\
&\quad \{\, (\langle\,\rangle, \{a, b, c.1, c.2, \checkmark\}), \ldots\\
&\qquad (\langle c.1 \rangle, \{a, b, c.1, c.2, \checkmark\}), \ldots, (\langle c.2 \rangle, \{a, b, c.1, c.2, \checkmark\}), \ldots \}
\end{aligned}$$

This reflects the fact that no user can observe whether the communication on c follows an a or a b. □

Unlike the (standard) failures-refinement relation, independent failures refinement cannot be used to reason about deadlock.

Example 11. The process $P = a \rightarrow STOP \mathrel{\square} b \rightarrow STOP$ is independent failures refined by $STOP$ for users $\{a\}$ and $\{b\}$, since for both of them an immediate deadlock is possible. We have the following failures.

$$\begin{aligned} &failures(P@\{a\}) = \{\, (\langle\,\rangle, \{a, b, \checkmark\}), \ldots, (\langle a\rangle, \{a, b, \checkmark\}), \ldots\} \\ &failures(P@\{b\}) = \{\, (\langle\,\rangle, \{a, b, \checkmark\}), \ldots, (\langle b\rangle, \{a, b, \checkmark\}), \ldots\} \end{aligned}$$

The set of failures of $STOP$, for both users, is $\{\, (\langle\,\rangle, \{a, b, \checkmark\}), \ldots\}$, which is a subset of the sets above. So, a deadlocked implementation is correct with respect to P, under independent failures refinement. □

Using the above result, the example below establishes that, like the other relations defined previously, independent failures refinement is not compositional.

Example 12. We define the process function $F(X) = (X \mathbin{[\![\{a\}]\!]} a \rightarrow b \rightarrow STOP)$. If P is as defined in Example 11, then $F(P) = a \rightarrow b \rightarrow STOP \mathrel{\square} b \rightarrow STOP$ and $F(STOP) = STOP$. Now, failures of $F(P)@\{b\}$ includes just $(\langle\,\rangle, \emptyset)$ for the empty trace. So, it is not the case that $F(P) \sqsubseteq_{IF} STOP$. □

Additionally, in some cases, internal and external choice may be perceived by individual users in the same way. An example is provided below.

Example 13. Process $P = a \rightarrow STOP \mathrel{\square} b \rightarrow STOP$ is equal, under independent failures refinement, to $Q = a \rightarrow STOP \sqcap b \rightarrow STOP$ if the users are $\{a\}$ and $\{b\}$. This is because, for P or Q, it is possible for $\{a\}$ or $\{b\}$ to observe a deadlock. For $\{a\}$, for instance, in the case of P, deadlock can happen if $\{b\}$ is quicker in making its choice, and in the case of Q, if the internal choice is made in favour of $b \rightarrow STOP$. A similar situation arises for the user $\{b\}$. □

This does not mean, however, that internal and external choice are indistinguishable using independent failures refinement.

Example 14. We now consider $P = a \rightarrow b \rightarrow STOP \mathrel{\square} b \rightarrow a \rightarrow STOP$ and $Q = a \rightarrow b \rightarrow STOP \sqcap b \rightarrow a \rightarrow STOP$, then we do not have an equality. In the case of P, the user $\{a\}$, for example, never experiences a deadlock, but in the case of Q, if $b \rightarrow a \rightarrow STOP$ is chosen, then a deadlock may occur for $\{a\}$, if $\{b\}$ is not ready for interaction. Accordingly, we have the following failures.

$$\begin{aligned} &failures(P@\{a\}) = \{\, (\langle\,\rangle, \{b, \checkmark\}), \ldots, (\langle a\rangle\,, \{a, b, \checkmark\}), \ldots\} \\ &failures(Q@\{a\}) = \{\, (\langle\,\rangle, \{a, b, \checkmark\}), \ldots, (\langle a\rangle, \{a, b, \checkmark\}), \ldots\} \end{aligned}$$

With the empty trace, there is no refusal of $P@\{a\}$ including a. □

As already discussed, in CSP, a process is in charge of the internal choices, and the environment, as a user, has no control over how they are made. With multiple users, we have the possibility of introducing more nondeterminism (from the point of view of a particular user), as there are more players who may be in sole control of choices that the process itself leaves to the environment.

The proof of the following is trivial. It considers the standard case in which the environment is a single user U that can observe every event: $\mathcal{A} = \{\Sigma^{\checkmark}\}$.

Lemma 2. $P @ \Sigma^{\checkmark} = P$

From this result, Lemma 1 and Definition 3, we conclude that independent refinement amounts to the traditional refinement relations if there is a single observer with a global view. Thus, existing exhaustive test sets for CSP apply in this case.

We give below another characterisation of independent failures refinement.

Lemma 3

$$P \sqsubseteq_{IF} Q \Leftrightarrow \left(\begin{array}{l} \forall\, U : \mathcal{A};\ s : \operatorname{seq} \Sigma^{\checkmark};\ A : \mathbb{P}\,\Sigma^{\checkmark} \mid (s, A \cap U) \in \mathit{failures}_{\perp}(Q) \bullet \\ \quad \exists\, t : \operatorname{seq} \Sigma^{\checkmark} \bullet (t, A \cap U) \in \mathit{failures}_{\perp}(P) \wedge \pi_U(s) = \pi_U(t) \end{array}\right)$$

Proof

$$P \sqsubseteq_{IF} Q$$

$$\Leftrightarrow \forall\, U : \mathcal{A} \bullet (P @ U) \sqsubseteq_F (Q @ U) \qquad [\text{definition of } \sqsubseteq_{IF}]$$

$$\Leftrightarrow \forall\, U : \mathcal{A} \bullet \mathit{failures}(Q @ U) \subseteq \mathit{failures}(P @ U) \qquad [\text{definition of } \sqsubseteq_F]$$

$$\Leftrightarrow \left(\begin{array}{l} \forall\, U : \mathcal{A};\ su : \operatorname{seq} \Sigma^{\checkmark};\ A : \mathbb{P}\,\Sigma^{\checkmark} \mid (su, A) \in \mathit{failures}(Q @ U) \bullet \\ \quad (su, A) \in \mathit{failures}(P @ U) \end{array}\right)$$

[property of sets]

$$\Leftrightarrow \left(\begin{array}{l} \forall\, U : \mathcal{A};\ su : \operatorname{seq} \Sigma^{\checkmark};\ A : \mathbb{P}\,\Sigma^{\checkmark} \mid \\ \quad (\exists\, s : \operatorname{seq} \Sigma^{\checkmark} \bullet (s, A \cap U) \in \mathit{failures}(Q) \wedge su = \pi_U(s)) \bullet \\ \qquad (\exists\, t : \operatorname{seq} \Sigma^{\checkmark} \bullet (t, A \cap U) \in \mathit{failures}(P) \wedge su = \pi_U(t)) \end{array}\right)$$

[definition of $\mathit{failures}(Q @ U)$ and $\mathit{failures}(P @ U)$]

$$\Leftrightarrow \left(\begin{array}{l} \forall\, U : \mathcal{A};\ su : \operatorname{seq} \Sigma^{\checkmark};\ A : \mathbb{P}\,\Sigma^{\checkmark};\ s : \operatorname{seq} \Sigma^{\checkmark} \mid \\ \quad (s, A \cap U) \in \mathit{failures}(Q) \wedge su = \pi_U(s) \bullet \\ \qquad (\exists\, t : \operatorname{seq} \Sigma^{\checkmark} \bullet (t, A \cap U) \in \mathit{failures}(P) \wedge su = \pi_U(t)) \end{array}\right)$$

[predicate calculus]

$$\Leftrightarrow \left(\begin{array}{l} \forall\, U : \mathcal{A};\ s : \operatorname{seq} \Sigma^{\checkmark};\ A : \mathbb{P}\,\Sigma^{\checkmark} \mid (s, A \cap U) \in \mathit{failures}(Q) \bullet \\ \quad (\exists\, t : \operatorname{seq} \Sigma^{\checkmark} \bullet (t, A \cap U) \in \mathit{failures}(P) \wedge \pi_U(s) = \pi_U(t)) \end{array}\right)$$

[one-point rule]

□

This states that $P \sqsubseteq_{IF} Q$ requires that, for every user U, every failure of Q whose refusal includes only events visible to U, has a corresponding failure in P. The failures can have different traces s and t, as long as they are the same from the point of view of U. The refusals must be the same.

Revisiting divergences. To remove the assumption that the behaviour of other users cannot cause user U to experience divergence, we need a different abstraction $P@^d U$ of P for U. If we assume that P cannot terminate, we can use $P@^d U = (P \,[\![\, \overline{U} \,]\!]\, Chaos(\overline{U})) \setminus \overline{U}$, where $\overline{U} = \Sigma \setminus U$ is the set of events under the control of other users. They are hidden in $P@^d U$, where the parallelism captures the fact that the behaviour of other users is arbitrary. Process $Chaos(A) = STOP \sqcap (\Box\, e : A \bullet e \rightarrow Chaos(A))$ can deadlock or perform any event in A at any time. Divergence arises in $P@^d U$ if P offers an infinite sequence of events in $\overline{U}$. This abstraction is suggested in [14], but there the introduction of divergence is considered inadequate. In testing, it is best not to make assumptions about the possible behaviours of a user.

In the traces model $P@U = P@^d U$, so using $P@^d U$ makes no difference for independent traces refinement. Additionally, to take into account the possibility that P may terminate, we need only to ensure that the parallelism in $P@^d U$ terminates when P does. To provide a more general definition that considers terminating P, we can, therefore, consider, for example, a (syntactic) function that changes P to indicate its imminent termination using a fresh event ok. With this modified version $\mathcal{OK}(P)$ of P, to define $P@^d U$ we can use $(\mathcal{OK}(P) \,[\![\, \overline{U} \,]\!]\, (Chaos(\overline{U}) \Box\, ok \rightarrow SKIP)) \setminus (\overline{U} \cup \{ok\})$. The failures of $P@^d U$ include those of $P@U$ plus those that can arise from divergence.

4 Distributed Testing and Traces Refinement

In this section we discuss distributed testing from CSP.

4.1 Global Testing for Traces Refinement

Since traces refinement $P \sqsubseteq_T Q$ prescribes $traces(Q) \subseteq traces(P)$, but not the reverse, there is no need to test that Q can execute the traces of P. It is sufficient to test Q against those traces of events in the alphabet of P that are not traces of P, and to check that they are refused. Moreover, it is sufficient to consider the minimal prefixes of forbidden traces that are forbidden themselves. Formally, testing for traces refinement is performed by proposing to the system under test the traces $s \frown \langle\, a \,\rangle$, where s is in $traces(P)$, and a is a forbidden continuation.

For one test execution, the verdict is as follows. If s is followed by a deadlock, then the test execution is said to be a *success*. If $s \frown \langle a \rangle$ is observed, the result is a *failure*. If a strict prefix of s followed by a deadlock is observed, then the execution is *inconclusive*. In this case, the trace s of P has not been executed by the system under test. As explained above, according to traces refinement, we do not have a failure, but the test does not produce conclusive information.

In [3], the three special events in the set $V = \{pass, fail, inc\}$ are introduced to perform on-the-fly verdict. For a finite trace $s = a_1, a_2, \ldots, a_n$ and a forbidden continuation event a, the CSP test process $T_T(s, a)$ is defined as follows.

$$T_T(s, a) = inc \rightarrow a_1 \rightarrow inc \rightarrow a_2 \rightarrow inc \ldots a_n \rightarrow pass \rightarrow a \rightarrow fail \rightarrow STOP$$

As explained above, the last event before a deadlock gives the verdict.

Formally, we can define $T_T(s, a)$ inductively as shown below.

$$\begin{aligned}
&T_T(\langle\,\rangle, a) = \mathit{pass} \rightarrow a \rightarrow \mathit{fail} \rightarrow \mathit{STOP}\\
&T_T(\langle \checkmark \rangle, a) = \mathit{pass} \rightarrow a \rightarrow \mathit{fail} \rightarrow \mathit{STOP}\\
&T_T(\langle b \rangle \frown s, a) = \mathit{inc} \rightarrow b \rightarrow T_T(s, a)
\end{aligned}$$

Execution $\mathit{Execution}^P_Q(T)$ of a test for Q, against a specification P, is described by the CSP process $(Q \parallel[\, \alpha P \,]\parallel T) \setminus \alpha P$. The exhaustive test set $\mathit{Exhaust}_T(P)$ for trace refinement of P contains all $T_T(s, a)$ formed from a trace $s \in \mathit{traces}(P)$, and forbidden continuation a. Proof of exhaustivity is in [3].

4.2 Local Distributed Testing

For simplicity we identify users by numbers, and index their events by these numbers. Since users need not have disjoint alphabets, an event may be indexed by several numbers. Moreover, we augment the set of events U of every user, with the set $V_U = \{\mathit{pass}_U, \mathit{fail}_U, \mathit{inc}_U\}$ of events for local verdicts.

Given $T_T(s, a)$ and a user U, we derive a local test $T_T(s, a)|_U$ by removing from $T_T(s, a)$ all events unobservable by U and associated verdicts.

Example 15. We consider users 1 and 2 defined by $\{a_1, b_1\}$ and $\{a_2, b_2\}$, and a specification $P = a_1 \rightarrow a_2 \rightarrow b_1 \rightarrow b_2 \rightarrow \mathit{STOP}$. We have a global test $T_T(\langle a_1, a_2 \rangle, a_1) = \mathit{inc} \rightarrow a_1 \rightarrow \mathit{inc} \rightarrow a_2 \rightarrow \mathit{pass} \rightarrow a_1 \rightarrow \mathit{fail} \rightarrow \mathit{STOP}$. The local tests are $T_T(\langle a_1, a_2 \rangle, a_1)|_1 = \mathit{inc}_1 \rightarrow a_1 \rightarrow \mathit{inc}_1 \rightarrow a_1 \rightarrow \mathit{fail}_1 \rightarrow \mathit{STOP}$ and $T_T(\langle a_1, a_2 \rangle, a_1)|_2 = \mathit{inc}_2 \rightarrow a_2 \rightarrow \mathit{pass}_2 \rightarrow \mathit{STOP}$, in this case. □

For every traces-refinement test T, that is, a CSP process in the range of the function T_T, and a user U, we define $T|_U$ inductively as follows.

$$\begin{aligned}
&(\mathit{inc} \rightarrow T)|_U = \mathit{inc}_U \rightarrow T|_U\\
&(a \rightarrow v \rightarrow T)|_U = a \rightarrow v_U \rightarrow T|_U, \text{ if } a \in U \setminus V_U, v \in V\\
&(a \rightarrow v \rightarrow T)|_U = T|_U, \text{ if } a \notin U, v \in V\\
&\mathit{STOP}|_U = \mathit{STOP}
\end{aligned}$$

The global tests for the empty trace start already with a *pass* event. The corresponding local tests are defined as follows.

$$\begin{aligned}
&(\mathit{pass} \rightarrow a \rightarrow \mathit{fail} \rightarrow \mathit{STOP})|_U = \mathit{pass}_U \rightarrow a \rightarrow \mathit{fail}_U \rightarrow \mathit{STOP}, \text{ if } a \in U\\
&(\mathit{pass} \rightarrow a \rightarrow \mathit{fail} \rightarrow \mathit{STOP})|_U = \mathit{inc}_U \rightarrow \mathit{STOP}, \text{ if } a \notin U
\end{aligned}$$

The distributed execution $\mathit{Execution}^P_Q(T, \mathcal{A})$ of local tests corresponding to a global test T, with set of users $\mathcal{A}$, for implementation Q, against a specification P, is described by the CSP process $(Q \parallel[\, \alpha P \,]\parallel (\parallel U : \mathcal{A} \bullet [U]\ T|_U)) \setminus \alpha P$. The test $(\parallel U : \mathcal{A} \bullet [U]\ T|_U)$ runs the local tests $T|_U$ for users U in $\mathcal{A}$ in parallel, with synchronisation only on common (original) events. Since the verdict events V_U of each user are different, each test produces its own verdict. The overall verdict arising from the experiment is *failure* if any user U observes a fail_U. If not, it is a *success* if any user observes a pass_U, and *inconclusive* otherwise.

We need to observe, however, that the local tests are not necessarily sound.

Example 16. We consider again users $\{a_1, b_1\}$ and $\{a_2, b_2\}$. For the specification $P = a_1 \rightarrow a_2 \rightarrow STOP \,\Box\, a_2 \rightarrow STOP$, we have a trace $\langle a_2 \rangle$, with forbidden continuation a_1. We have a global test $inc \rightarrow a_2 \rightarrow pass \rightarrow a_1 \rightarrow fail \rightarrow STOP$. The local tests are $inc_1 \rightarrow a_1 \rightarrow fail_1 \rightarrow STOP$ and $inc_2 \rightarrow a_2 \rightarrow pass_2 \rightarrow STOP$. If the system performs the trace $\langle a_1, a_2 \rangle$, the verdict of these tests is *failure*, even though $\langle a_1, a_2 \rangle$ is a trace of the specification P. □

Here we have a controllability problem: the second local test should not start until after event a_2. Under certain conditions, soundness is guaranteed: for instance, if for every component $a \rightarrow v \rightarrow b \rightarrow T$ of the test, where a and b are not verdict events, but v is, at least one user can observe both a and b. In the next section we explore the use of coordination messages to address this issue.

4.3 Coordination Messages and Traces Refinement

The approach presented here is inspired by that in Hierons [5]. First, we introduce coordination messages as events $coord.i.j$ observable by users i and j. The role of such an event is to allow the tester i to warn the tester j that the event a_i has just been performed and j is entitled to propose the event a_j.

For a global test $T_T(s, a)$, defined from a trace s and a forbidden event a observable by a user k, coordination messages are inserted in the local tests as follows. For every pair a_i and a_j of consecutive events of s observed by different users i and j, the event $coord.i.j$ is inserted in $T_T(s, a)|_i$ after a_i and in $T_T(s, a)|_j$ before a_j. If the user i that observes the last event a_i of s is not k, then $coord.i.k$ is inserted in $T_T(s, a)|_i$ after a_i and in $T_T(s, a)|_k$ before a.

Example 17. We consider the global test $T_T(\langle a_1, a_2 \rangle, a_1)$ from Example 15. We get $inc_1 \rightarrow a_1 \rightarrow coord.1.2 \rightarrow inc_1 \rightarrow coord.2.1 \rightarrow a_1 \rightarrow fail_1 \rightarrow STOP$ as the coordinated version of the local test $T_T(\langle a_1, a_2 \rangle, a_1)|_1$. That of $T_T(\langle a_1, a_2 \rangle, a_1)|_2$ is $inc_2 \rightarrow coord.1.2 \rightarrow a_2 \rightarrow coord.2.1 \rightarrow pass_2 \rightarrow STOP$. □

The function $C^i(T)$ that defines the annotated local test for user i from a global test T can be defined inductively as follows.

$$
\begin{array}{l}
C^i(inc \rightarrow T) = inc_i \rightarrow C^i(T) \\
C^i(a_i \rightarrow v \rightarrow b_i \rightarrow T) = a_i \rightarrow v_i \rightarrow C^i(b_i \rightarrow T) \\
C^i(a_i \rightarrow v \rightarrow a_k \rightarrow T) = a_i \rightarrow coord.i.k \rightarrow v_i \rightarrow C^i(a_k \rightarrow T), \text{ if } k \neq i \\
C^i(a_j \rightarrow v \rightarrow a_i \rightarrow T) = coord.j.i \rightarrow C^i(a_i \rightarrow T), \text{ if } j \neq i \\
C^i(a_j \rightarrow v \rightarrow a_k \rightarrow T) = C^i(a_k \rightarrow T), \text{ if } j \neq i, k \neq i \\
C^i(a_i \rightarrow fail \rightarrow STOP) = a_i \rightarrow fail_i \rightarrow STOP \\
C^i(a_j \rightarrow fail \rightarrow STOP) = STOP, \text{ if } j \neq i \\
C^i(pass \rightarrow a_i \rightarrow fail \rightarrow STOP) = pass_i \rightarrow a_i \rightarrow fail_i \rightarrow STOP \\
C^i(pass \rightarrow a_j \rightarrow fail \rightarrow STOP) = inc_i \rightarrow STOP, \text{ if } j \neq i
\end{array}
$$

The distributed test is defined by $(\,\|\, U : \mathcal{A} \bullet [AC(U)] C^U(T)) \setminus \{| coord |\}$. As before, we have a parallel composition of the local tests. $AC(U)$ is the alphabet of $C^U(T)$, including U, and the events $coord.U.i$ and $coord.i.U$, for every user

i in $\mathcal{A}$. The set $\{\!|coord|\!\}$ of all coordination events is hidden, as they are used for interaction among the tests, but not with the system under test.

The role of the coordination messages as defined above is to preserve the global order of events when using local tests. Since the synchronisation constraints they introduce force the local tests to follow the order of the global tests, which have been defined to exhaustively test for traces refinement, they introduce a distributed coordinated way to test for such refinement. It is at the price of their proliferation, which seems unavoidable in general, if traces refinement is required and if there is no way of performing global testing. It is likely, however, that there are some types of systems where the coordination of distributed tests is less costly, for instance when global traces have some locality properties, like long local subtraces, and few switches between users.

5 Conclusions

This paper has explored distributed testing from a CSP specification. While there has been much work on distributed testing from an IOTS or a finite state machine, the use of CSP introduces new challenges. For example, since some refinement relations for CSP assume that we can observe failures in addition to traces, there is a need to incorporate failures into the framework. The distinction between internal choice and external choice also introduces interesting issues.

We have considered two situations. In the first, the testers are distributed, but their observations can be brought together. This leads to the notion of cooperating refinement. In this case, it is necessary to decide when the observations can be brought together, since the testers need to know that they are reporting observations regarding the same trace. We have, therefore, defined the notion of a complete run, which is either infinite or a trace that terminates. Since the failure sets are the same after all complete runs, there is no value in observing them, and so we only observe projections of traces.

In the alternative situation, the testers act entirely independently. Here it is sufficient for the observations made by each tester to be consistent with the specification, even if the set of observations is not. A single tester can observe traces and failures, and as a result we have defined independent traces refinement, under which only traces are observed, and independent failures refinement, under which traces and failures are observed. We have also considered test generation and showed how coordination messages might be used to make tests sound.

There are several avenues for future work. First, under cooperating refinement the testers do not observe failures and it would be interesting to find ways of incorporating information regarding failures. In addition, CSP does not distinguish between inputs and outputs. It is, however, possible to include such a distinction through the notion of non-delayable events to model outputs. This is certainly a crucial step to enable the study of more elaborate concerns regarding observability and control. Recent work has shown how the notion of a scenario, which is a sequence of events after which the testers might synchronise, can be used in distributed testing from an IOTS and it should be possible to introduce scenarios into CSP. Additionally, the work in [3] considers *conf* as well as traces refinement; distributed testing for *conf* might raise interesting issues for coordination. Finally, tools for test generation and case studies are needed to explore the applications of the theory presented here.

Acknowledgments. Cavalcanti and Gaudel are grateful for the support of the Royal Academy of Engineering. We acknowledge invaluable contributions by Manuel Núñez to a previous version of this paper and to many discussions.

References

1. Brinksma, E., Heerink, L., Tretmans, J.: Factorized test generation for multi-input/output transition systems. In: 11th IFIP Workshop on Testing of Communicating Systems, pp. 67–82. Kluwer Academic Publishers, Dordrecht (1998)
2. Cacciari, L., Rafiq, O.: Controllability and observability in distributed testing. IST 41(11-12), 767–780 (1999)
3. Cavalcanti, A.L.C., Gaudel, M.-C.: Testing for Refinement in CSP. In: Butler, M., Hinchey, M.G., Larrondo-Petrie, M.M. (eds.) ICFEM 2007. LNCS, vol. 4789, pp. 151–170. Springer, Heidelberg (2007)
4. Dssouli, R., Bochmann, G.v.: Error detection with multiple observers. In: PSTV, pp. 483–494. Elsevier Science, Amsterdam (1985)
5. Hierons, R.M.: Overcoming controllability problems in distributed testing from an input output transition system (submitted), `people.brunel.ac.uk/~csstrmh/coord.pdf`
6. Hierons, R.M., Merayo, M.G., Núñez, M.: Controllable test cases for the distributed test architecture. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 201–215. Springer, Heidelberg (2008)
7. Hierons, R.M., Merayo, M.G., Nunez, M.: Implementation relations for the distributed test architecture. In: Suzuki, K., Higashino, T., Ulrich, A., Hasegawa, T. (eds.) TestCom/FATES 2008. LNCS, vol. 5047, pp. 200–215. Springer, Heidelberg (2008)
8. Hierons, R.M., Merayo, M.G., Núñez, M.: Scenarios-based testing of systems with distributed ports. Software - Practice and Experience (accepted for publication, 2011), doi:10.1002/spe.1062
9. Hierons, R.M., Nunez, M.: Scenarios-based testing of systems with distributed ports. In: 10th QSIC (2010)
10. Jacob, J.: Refinement of shared systems. In: The Theory and Practice of Refinement, pp. 27–36. Butterworths (1989)
11. Jard, C.: Synthesis of distributed testers from true-concurrency models of reactive systems. IST 45(12), 805–814 (2003)
12. Jard, C., Jéron, T., Kahlouche, H., Viho, C.: Towards automatic distribution of testers for distributed conformance testing. In: FORTE, pp. 353–368. Kluwer Academic Publishers, Dordrecht (1998)
13. Pickin, S., Jard, C., Traon, Y.L., Jéron, T., Jézéquel, J.-M., Guennec, A.L.: System test synthesis from UML models of distributed software. In: Peled, D.A., Vardi, M.Y. (eds.) FORTE 2002. LNCS, vol. 2529, pp. 97–113. Springer, Heidelberg (2002)
14. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall, Englewood Cliffs (1998)
15. Sarikaya, B., Bochmann, G.v.: Synchronization and specification issues in protocol testing. IEEE Transactions on Communications 32, 389–395 (1984)
16. Ural, H., Williams, C.: Constructing checking sequences for distributed testing. FAC 18(1), 84–101 (2006)

Praspel: A Specification Language for Contract-Based Testing in PHP

Ivan Enderlin, Frédéric Dadeau, Alain Giorgetti, and Abdallah Ben Othman

LIFC / INRIA CASSIS Project – 16 route de Gray - 25030 Besançon cedex, France
{ivan.enderlin,abdallah.ben_othman}@edu.univ-fcomte.fr,
{frederic.dadeau,alain.giorgetti}@univ-fcomte.fr

Abstract. We introduce in this paper a new specification language named Praspel, for PHP Realistic Annotation and SPEcification Language. This language is based on the Design-by-Contract paradigm. Praspel clauses annotate methods of a PHP class in order to both specify their contracts, using pre- and postconditions, and assign realistic domains to the method parameters. A realistic domain describes a set of concrete, and hopefully relevant, values that can be assigned to the data of a program (class attributes and method parameters). Praspel is implemented into a unit test generator for PHP that offers a random test data generator, which computes test data, coupled with a runtime assertion checker, which decides whether a test passes or fails by checking the satisfaction of the contracts at run-time.

Keywords: PHP, Design-by-Contract, annotation language, unit testing, formal specifications.

1 Introduction

Over the years, testing has become the main way to validate software. A challenge is the automation of test generation that aims to unburden the developers from writing their tests manually. Recent development techniques, such as Agile methods, consider tests as first-class citizens, that are written prior to the code. Model-based testing [5] is an efficient paradigm for automating test generation. It considers a model of the system that is used for generating conformance test cases (w.r.t. the model) and computing the oracle (*i.e.* the expected result) that is used to decide whether a test passes or fails.

In order to ease the model description, annotation languages have been designed, firstly introduced by B. Meyer [19], creating the *Design-by-Contract* paradigm. These languages make it possible to express formal properties (invariants, preconditions and postconditions) that directly annotate program entities (class attributes, methods parameters, etc.) in the source code. Many annotation languages exist, such as the Java Modeling Language (JML) [15] for Java, Spec# [3] for C#, or the ANSI-C Specification Language (ACSL) [4] for C. Design-by-Contract considers that a system has to be used in a contractual way: to invoke a method the caller has to fulfil its precondition; in return, the method establishes its postcondition.

Contract-Driven Testing. Annotations can be checked at run time to make sure that the system behaves as specified, and does not break any contract. Contracts are thus well-suited to testing, and especially to unit testing [16]. The idea of Contract-Driven Testing [1] is to rely on contracts for both producing tests, by computing test data satisfying the contract described by the precondition, and for test verdict assessment, by checking that the contract described by the postcondition is ensured after execution. On one hand, method preconditions can be used to generate test data, as they characterize the states and parameters for which the method call is licit. For example, the Jartege tool [20] generates random test data, in accordance with the domain of the inputs of a given Java method, and rejects the values that falsify the precondition. The JML-Testing-Tools toolset [6] uses the JML precondition of a method to identify boundary states from which the Java method under test will be invoked. The JMLUnit [9] approach considers systematic test data for method input parameters, and filters irrelevant ones by removing those falsifying the method precondition. On the other hand, postconditions are employed similarly in all the approaches [6,8,9]. By runtime assertion checking, the postcondition is verified after each method execution, to provide the test verdict.

Contributions. In this paper, we present a new language named Praspel, for *PHP Realistic Annotation and SPEcification Language*. Praspel is a specification language for PHP [21] which illustrates the concept of realistic domains. Praspel introduces Design-by-Contract in PHP, by specifying realistic domains on class attributes and methods. Consequently, Praspel is adapted to test generation: contracts are used for unit test data generation and provide the test oracle by checking the assertions at runtime.

Our second contribution is a test framework supporting this language. This online test generation and execution tool works in three steps: (*i*) the tool generates values for variables according to the contract (possibly using different data generators), (*ii*) it runs the PHP program with the generated values, and (*iii*) the tool checks the contract postcondition to assign the test verdict.

Paper outline. The paper is organized as follows. Section 2 briefly introduces the concept of realistic domains. Section 3 presents an implementation of realistic domains in the Praspel language, a new annotation language for PHP. Section 4 describes the mechanism of automated generation of unit tests from PHP files annotated with Praspel specifications. The implementation of Praspel is described in Section 5. Section 6 compares our approach with related works. Finally, Section 7 concludes and presents our future work.

2 Realistic Domains

When a method precondition is any logical predicate, say from first-order logic, it can be arbitrarily difficult to generate input data satisfying the precondition. One could argue that the problem does not appear in practice because usual preconditions are only simple logical predicates. But defining what is a "simple" predicate w.r.t. the problem of generating values satisfying it (its *models*) is a difficult question, still partly open. We plan to address this question and to put its positive answers at the disposal of the test community. In order to reach this goal we introduce the concept of realistic domains.

Realistic domains are intended to be used for test generation purposes. They specify the set of values that can be assigned to a data in a given program. Realistic domains are well-suited to PHP, since this language is dynamically typed. Therefore, realistic domains introduce a specification of data types that are mandatory for test data generation. We introduce associated features to realistic domains, and we then present the declination of realistic domains for PHP.

2.1 Features of Realistic Domains

Realistic domains are structures that come with necessary properties for the validation and generation of data values. Realistic domains can represent all kinds of data; they are intended to specify relevant data domains for a specific context. Realistic domains are more subtle than usual data types (integer, string, array, etc.) and are actually refinement of those latters. For example, if a realistic domain specifies an email address, we can validate and generate strings representing syntactically correct email addresses, as shown in Fig. 1.

Realistic domains display two features, which are now described and illustrated.

Predicability. The first feature of a realistic domain is to carry a characteristic predicate. This predicate makes it possible to check if a value belongs to the possible set of values described by the realistic domain.

Samplability. The second feature of a realistic domain is to propose a value generator, called the *sampler*, that makes it possible to generate values in the realistic domain. The data value generator can be of many kinds: a random generator, a walk in the domain, an incrementation of values, etc.

We now present our implementation of realistic domains in PHP and show some interesting additional principles they obey.

2.2 Realistic Domains in PHP

In PHP, we have implemented realistic domains as classes providing at least two methods, corresponding to the two features of realistic domains. The first method is named `predicate($q)` and takes a value `$q` as input: it returns a boolean indicating the membership of the value to the realistic domain. The second method is named `sample()` and generates values that belong to the realistic domain. An example of realistic domain implementation in a PHP class is given in Fig. 1. This class represents the `EmailAddress` realistic domain already mentioned. Our implementation of realistic domains in PHP exploit the PHP object programming paradigm and benefit from the following two additional principles.

```
class EmailAddress extends String {

  public function predicate($q) {
    // regular expression for email addresses
    //  see. RFC 2822, 3.4.1. address specs.
    $regexp = '...';
    if(false === parent::predicate($q))
      return false;

    return preg_match($regexp,$q);
  }

  public function sample() {
    // string of authorized chars
    $chars = 'ABCDEFGHIJKL...';
    // array of possible domain extensions
    $doms  = array('net','org','edu','com');
    $q     = '';
    $nbparts = rand(2, 4);

    for($i = 0; $i < $nbparts; ++$i) {
      if($i > 0)
        // add separator or arobase
        $q .= ($i == $nbparts - 1)
                   ? '@' : '.';

      $len = rand(1,10);
      for($j=0; $j < $len; ++$j) {
        $index = rand(0, strlen($chars) - 1);
        $q .= $chars[$index];
      }
    }
    $q .= '.' .
        $doms[rand(0, count($doms) - 1)];
    return $q;
  }
}
```

Fig. 1. PHP code of the EmailAddress realistic domain

Hierarchical inheritance. PHP realistic domains can inherit from each other. A realistic domain child inherits the two features of its parent and is able to redefine them. Consequently, all the realistic domains constitute an universe.

Parameterizable realistic domains. Realistic domains may have parameters. They can receive arguments of many kinds. In particular, it is possible to use realistic domains as arguments of realistic domains. This notion is very important for the generation of recursive structures, such as arrays, objects, graphs, automata, etc.

Example 1 (Realistic domains with arguments). The realistic domain `boundinteger(`X`, `Y`)` contains all the integers between X and Y. The realistic domain `string(`L`, `X`, `Y`)` is intended to contain all the strings of length L constituted of characters from X to Y Unicode code-points. In the realistic domain `string(boundinteger(4, 12), 0x20, 0x7E)`, the string length is defined by another realistic domain.

3 PHP Realistic Annotation and Specification Language

Realistic domains are implemented for PHP in Praspel, a dedicated annotation language based on the Design-by-Contract paradigm [19]. In this section, we present the syntax and semantics of the language.

Praspel specifications are written in API documentation comments as shown in Fig. 3, 4, 5 and 6.

Praspel makes it possible to mix informal documentations and formal constraints, called *clauses* and described hereafter. Praspel clauses are ignored by PHP interpreters and integrated development environments. Moreover, since each Praspel clause begins with the standard `@` symbol for API keywords, it is usually well-handled by pretty printers and API documentation generators.

$$
\begin{array}{rcl}
\mathit{annotation} & ::= & (\mathit{clause})^* \\
\mathit{clause} & ::= & \mathit{requires\text{-}clause}\,\texttt{;} \\
 & | & \mathit{ensures\text{-}clause}\,\texttt{;} \\
 & | & \mathit{throwable\text{-}clause}\,\texttt{;} \\
 & | & \mathit{predicate\text{-}clause}\,\texttt{;} \\
 & | & \mathit{invariant\text{-}clause}\,\texttt{;} \\
 & | & \mathit{behavior\text{-}clause} \\
\mathit{requires\text{-}clause} & ::= & \texttt{@requires}\ \mathit{expressions} \\
\mathit{ensures\text{-}clause} & ::= & \texttt{@ensures}\ \mathit{expressions} \\
\mathit{throwable\text{-}clause} & ::= & \texttt{@throwable}\ (\underline{\mathit{identifier}})^+_{,} \\
\mathit{invariant\text{-}clause} & ::= & \texttt{@invariant}\ \mathit{expressions} \\
\mathit{behavior\text{-}clause} & ::= & \texttt{@behavior}\ \mathit{identifier}\ \{ \\
 & & (\mathit{requires\text{-}clause}\,\texttt{;} \\
 & | & \mathit{ensures\text{-}clause}\,\texttt{;} \\
 & | & \mathit{throwable\text{-}clause}\,\texttt{;})^+\ \} \\
\mathit{expressions} & ::= & (\mathit{expression})^+_{\texttt{and}} \\
\mathit{expression} & ::= & \mathit{real\text{-}dom\text{-}spec} \\
 & | & \texttt{\\pred(}\underline{\mathit{predicate}}\texttt{)} \\
\mathit{real\text{-}dom\text{-}spec} & ::= & \mathit{variable}\ (\texttt{:}\ \mathit{real\text{-}doms} \\
 & & \quad |\ \texttt{domainof}\ \mathit{variable})
\end{array}
\qquad
\begin{array}{rcl}
\mathit{variable} & ::= & \mathit{constructors}\ |\ \underline{\mathit{identifier}} \\
\mathit{constructors} & ::= & \texttt{\\old(}\underline{\mathit{identifier}}\texttt{)}\ |\ \texttt{\\result} \\
\mathit{real\text{-}doms} & ::= & \mathit{real\text{-}dom}^+_{\texttt{or}} \\
\mathit{real\text{-}dom} & ::= & \underline{\mathit{identifier}}\ \texttt{(}\mathit{arguments}\texttt{)} \\
 & | & \mathit{built\text{-}in} \\
\mathit{built\text{-}in} & ::= & \texttt{void()} \\
 & | & \texttt{integer()} \\
 & | & \texttt{float()} \\
 & | & \texttt{boolean()} \\
 & | & \texttt{string(}\mathit{arguments}\texttt{)} \\
 & | & \texttt{array(}\mathit{arguments}\texttt{)} \\
 & | & \texttt{class(}\mathit{arguments}\texttt{)} \\
\mathit{arguments} & ::= & (\mathit{argument})^*_{,} \\
\mathit{argument} & ::= & \underline{\mathit{number}}\ |\ \underline{\mathit{string}} \\
 & | & \mathit{real\text{-}dom}\ |\ \mathit{array} \\
\mathit{array} & ::= & \texttt{[}\mathit{pairs}\texttt{]} \\
\mathit{pairs} & ::= & (\mathit{pair})^*_{,} \\
\mathit{pair} & ::= & (\texttt{from}\ \mathit{real\text{-}doms}\ )?\texttt{to}\ \mathit{real\text{-}doms}
\end{array}
$$

Fig. 2. Grammar of the concrete syntax

The grammar of Praspel annotations is given in Fig. 2. Notation $(\sigma)?$ means that σ is optional. $(\sigma)^r_s$ represents finite sequences of elements matching σ, in which r is either `+` for one or more matches, or `*` for zero or more matches, and s is the separator for the elements in the sequence.

Underlined words are PHP entities. They are exactly the same as in PHP. A $\underline{\textit{predicate}}$ is a valid logical PHP expression that returns a boolean. An $\underline{\textit{identifier}}$ is the name of a PHP class or the name of a method or method parameter. It cannot be the name of a global variable or an attribute, which are respectively prohibited (as bad programming) and defined as invariants in Praspel. The syntax of identifiers strictly follows the syntax of PHP variables.

The other syntactic entities in this grammar are explained in the PHP manual [21]. Praspel expressions are conjunctions of realistic domain assignments and of relations between realistic domains, explained in Section 3.1. The special case of array descriptions is explained in Section 3.2. Finally, clauses are described in Section 3.3.

3.1 Assigning Realistic Domains to Data

We now explain how to declare the realistic domains of method parameters and we give the semantics of these declarations w.r.t. the method inputs and output.

The syntactic construction:

$$\texttt{i:}\ t_1(\dots)\ \texttt{or}\ \dots\ \texttt{or}\ t_n(\dots)$$

associates at least one realistic domain (among $t_1(\dots)$, ..., $t_n(\dots)$) to an identifier (here, `i`). We use the expression "domains disjunction" when speaking about syntax, and the expression "domains union" when speaking about semantics. The left-hand side represents the name of some method argument, whereas the right-hand side is a list of realistic domains, separated by the "`or`" keyword.

The semantics of such a declaration is that the realistic domain of the identifier `i` may be $t_1(\ldots)$ or …or $t_n(\ldots)$, *i.e.* it is the union (as in the C language) of the realistic domains $t_1(\ldots)$ to $t_n(\ldots)$. These realistic domains are (preferably) mutually exclusive.

Example 2 (An identifier with many realistic domains). The declaration:

```
y: integer() or float() or boolean()
```

means that `y` can either be an integer, a floating-point number or a boolean.

The `domainof` operator describes a dependency between the realistic domains of two identifiers. The syntactic construction: *identifier* `domainof` *identifier* creates this relation. The semantics of i `domainof` j is that the realistic domain chosen at runtime for j is the same as for i.

When an object is expected as a parameter, it can be specified using the `class(`C`)` construct, in which C is a string designating the class name.

Example 3 (Use of a class as a realistic domain). The following declaration:

```
o: class('LimitIterator') or class('RegexIterator')
```

specifies that `o` is either an instance of `LimitIterator` or `RegexIterator`.

3.2 Array Description

A realistic domain can also be an array description. An array description has the following form:

$$\begin{array}{l}\texttt{[from } T_1^1\texttt{(...) or ... or } T_i^1\texttt{(...) to } T_{i+1}^1\texttt{(...) or ... or } T_n^1\texttt{(...),}\\ \ \ \texttt{...}\\ \ \texttt{from } T_1^k\texttt{(...) or ... or } T_j^k\texttt{(...) to } T_{j+1}^k\texttt{(...) or ... or } T_m^k\texttt{(...)]}\end{array}$$

It is a sequence between square brackets "`[`" and "`]`" of pairs separated by symbol "`,`". Each pair is composed of a domain introduced by the keyword "`from`", and a co-domain introduced by the keyword "`to`". Each domain and co-domain is a disjunction of realistic domains separated by the keyword "`or`". The domain is optional.

The semantics of an array description depends of the realistic domain where it is used. We detail this semantics in the most significant case, when the array description is a parameter of the realistic domain `array`. Notice that an array description and the realistic domain `array` are different. The realistic domain `array` has two arguments: the array description and the array size.

Example 4 (Array specification). Consider the following declarations:

```
a1: array([from integer() to boolean()], boundinteger(7, 42))
a2: array([to boolean(), to float()], 7)
a3: array([from integer() to boolean() or float()], 7)
a4: array([from string(11) to boolean(), to float() or integer()], 7)
```

`a1` describes an homogeneous array of integers to booleans. The size of the yielded array is an integer between 7 and 42. In order to produce a fixed-size array, one must use an explicit constant, as in the subsequent examples. `a2` describes two homogeneous arrays: either an array of booleans, or an array of floats, but not both. In all cases, the yielded array will have a size of 7. If no realistic domain is given as domain (*i.e.* if the keyword "`from`" is not present), then an auto-incremented integer (an integer that is incremented at each sampling) will be yielded. `a2` is strictly equivalent to `array([to boolean()], 7) or array([to float()], 7)`. `a3` describes an heterogeneous array of booleans and floats altogether. Finally, `a4` describes either an homogeneous array of strings to booleans or an heterogeneous array of floats and integers.

3.3 Designing Contracts in Praspel

This section describes the syntax and semantics of the part of Praspel that defines contracts for PHP classes and methods. Basically, a contract clause is either the assignment of a realistic domain to a given data, or it is a PHP predicate that provides additional constraints over variables values (denoted by the `\pred` construct).

Invariants. In the object programming paradigm, class attributes may be constrained by properties called "invariants". These properties must hold before and after any method call and have the following syntax.

`@invariant` I_1 `and` ... `and` I_p`;`

Classically, invariants have to be satisfied after the creation of the object, and preserved through each method execution (*i.e.* assuming the invariant holds before the method, then it has to hold once the method has terminated). Invariants are also used to provide a realistic domain to the attributes of an object.

```
class C {
  /**
   * @invariant a: boolean();
   */
  protected $a;
}
```

Fig. 3. Example of invariant clause

Example 5 (Simple invariant). The invariant in Fig. 3 specifies that the attribute `a` is a boolean before and after any method call.

Method contracts. Praspel makes it possible to express contracts on the methods in a class. The contract specifies a precondition that has to be fulfilled for the method to be executed. In return, the method has to establish the specified postcondition. The contract also specifies a set of possible exceptions that can be raised by the method. The syntax of a basic method contract is given in Fig. 4. The semantics of this contract is as follows.

Each R_x $(1 \leq x \leq n)$ represents either the assignment of a realistic domain to a data, or a PHP predicate that provides a precondition. The caller of the method must guarantee that the method is called in a state where the properties conjunction $R_1 \wedge \ldots \wedge R_n$ holds (meaning that the PHP predicate is true, and the value of the data should match their assigned realistic domains). By default, if no `@requires` clause is declared, the parameter is implicitly declared with the `undefined` realistic domain (which has an always-true predicate and sample an integer by default).

Each E_z $(1 \leq z \leq m)$ is either the assignment of a realistic domain to a data (possibly the result of the method) after the execution of the method, or an assertion specified using a PHP predicate that specifies the postcondition of the method. The method, if it terminates normally, should return a value such that the conjunction $E_1 \wedge \ldots \wedge E_m$ holds.

```
/**
 * @requires   R_1 and ... and R_n;
 * @ensures    E_1 and ... and E_m;
 * @throwable T_1, ..., T_p;
 */
function foo ( ... ) { ... }
```

Fig. 4. Syntax of Method Contracts

Each T_y $(1 \leq y \leq p)$ is an exception name. The method may throw one of the specified exceptions $T_1 \vee \ldots \vee T_p$ or a child of these ones. By default, the specification does not authorize any exception to be thrown by the method.

Postconditions (`@ensures` clauses) usually have to refer to the method result and to the variables in the pre-state (before calling the function). Thus, PHP expressions are extended with two new constructs: `\old(`e`)` denotes the value of expression e in the pre-state of the function, and `\result` denotes the value returned by the function. Notice that our language does not include first-order logic operators such as universal or existential quantifiers. Nevertheless, one can easily simulate such operators using a dedicated boolean function, that would be called in the PHP predicate.

```
/**
 * @requires  x: boundinteger(0,42);
 * @ensures   \result: eveninteger()
 *        and \pred(x >= \old(x));
 * @throwable FooBarException;
 */
function foo ( $x ) {
  if($x === 42)
    throw new FooBarException();
  return $x * 2;
}
```

Fig. 5. Example of Simple Contract

Example 6 (Simple contract). Consider the example provided in Fig. 5. This function `foo` doubles the value of its parameter `$x` and returns it. In a special case, the function throws an exception.

Behavorial clauses. In addition, Praspel makes it possible to describe explicit *behaviors* inside contracts.

A behavior is defined by a name and local `@requires`, `@ensures`, and `@throwable` clauses (see Fig. 6(a)). The semantics of behavioral contracts is as follows.

```
/**
 * @requires $R_1$ and ... and $R_n$;
 * @behavior $\alpha$ {
 *   @requires  $A_1$ and ... and $A_k$;
 *   @ensures   $E_1$ and ... and $E_j$;
 *   @throwable $T_1$, ..., $T_t$;
 * }
 * @ensures   $E_{j+1}$ and ... and $E_m$;
 * @throwable $T_{t+1}$, ..., $T_l$;
 */
function foo ( $x_1$... ) { body }
```

(a)

```
/**
 * @requires x: integer();
 * @behavior foo {
 *   @requires  y: positiveinteger()
 *         and  z: boolean();
 * }
 * @behavior bar {
 *   @requires  y: negativeinteger()
 *         and  z: float();
 *   @throwable BarException;
 * }
 * @ensures \result: boolean();
 */
function foo ( $x, $y, $z ) { ... }
```

(b)

Fig. 6. Behavioral Contracts

The caller of the method must guarantee that the call is performed in a state where the property $R_1 \wedge \ldots \wedge R_n$ holds. Nevertheless, property $A_1 \wedge \ldots \wedge A_k$ should also hold. The called method establishes a state where the property $(A_1 \wedge \ldots \wedge A_k \Rightarrow E_1 \wedge \ldots \wedge E_j) \wedge E_{j+1} \wedge \ldots \wedge E_m$ holds, meaning that the postcondition of the specified behavior only has to hold if the precondition of the behavior is satisfied. Exceptions T_i $(1 \leq i \leq t)$ can only be thrown if the preconditions $R_1 \wedge \ldots \wedge R_n$ and $A_1 \wedge \ldots \wedge A_k$ hold.

The `@behavior` clause only contains `@requires`, `@ensures` and `@throwable` clauses. If a clause is declared or used outside a behavior, it will automatically be set into a default/global behavior. If a clause is missing in a behavior, the clause in the default behavior will be chosen.

Example 7 (Behavior with default clauses). The specification in Fig. 6(b) is an example of a complete behavioral clause. This contract means: the first argument `$x` is always an integer and the result is always a boolean, but if the second argument `$y` is a positive integer, then the third argument `$z` is a boolean (behavior `foo`), else if `$y` is a negative integer, and the `$z` is a float and then the method may throw an exception (behavior `bar`).

4 Automated Unit Test Generator

The unit test generator works with the two features provided by the realistic domains. First, the sampler is implemented as a random data generator, that satisfies the precondition of the method. Second, the predicate makes it possible to check the postcondition (possibly specifying realistic domains too) at runtime after the execution of the method.

4.1 Test Verdict Assignment Using Runtime Assertion Checking

The test verdict assignment is based on the runtime assertion checking of the contracts specified in the source code. When the verification of an assertion

```
public function foo ($x1 ...) {

  $this->foo_pre(...);

  // evaluation of \old(e)

  try {

    $result = $this->foo_body($x1 ... );

  }
 catch ( Exception $exc ) {
    $this->foo_exception($exc);
    throw $exc;
 }

 $this->foo_post($result, ...);

 return $result;
}
```

```
public function foo_pre(...) {

  return    verifyInvariants(...)
         && verifyPreCondition(...);
}

public function foo_post(...) {

  return    verifyPostCondition(...)
         && verifyInvariants(...);
}

public function foo_exception($e) {

  return    verifyException($e)
         && verifyInvariants(...);
}

public function foo_body($x1 ... ) ...
```

Fig. 7. PHP Instrumentation for Runtime Assertion Checking

fails, a specific error is logged. The runtime assertion checking errors (a.k.a. Praspel failures) can be of five kinds. (*i*) precondition failure, when a precondition is not satisfied at the invocation of a method, (*ii*) postcondition failure, when a postcondition is not satisfied at the end of the execution of the method, (*iii*) throwable failure, when the method execution throws an unexpected exception, (*iv*) invariant failure, when the class invariant is broken, or (*v*) internal precondition failure, which corresponds to the propagation of the precondition failure at the upper level.

The runtime assertion checking is performed by instrumenting the initial PHP code with additional code which checks the contract clauses. The result of the code instrumentation of a given method `foo` is shown in Fig. 7. It corresponds to the treatment of a behavioral contract, as shown in Fig. 6. The original method `foo` is duplicated, renamed (as `foo_body`) and substituted by a new `foo` method which goes through the following steps:

- First, the method checks that the precondition is satisfied at its beginning, using the auxiliary method `foo_pre`.
- Second, the `\old` expressions appearing in the postconditions are evaluated and stored for being used later.
- Third, the replication of the original method body is called. Notice that this invocation is surrounded by a try-catch block that is in charge of catching the exception that may be thrown in the original method.
- Fourth, when the method terminates with an exception, this exception is checked against the expected ones, using the auxiliary method `foo_exception`. Then the exception is propagated so as to preserve the original behavior of the method.

– Fifth, when the method terminates normally, the postconditions and the invariants are checked, using the auxiliary method `foo_post`.
– Sixth, and finally, the method returns the value resulting of the execution of `foo_body`.

Test cases are generated and executed online: the random test generator produces test data, and the instrumented version of the initial PHP file checks the conformance of the code w.r.t. specifications for the given inputs. The test succeeds if no Praspel failure (listed previously) is detected. Otherwise, it fails, and the log indicates where the failure has been detected.

4.2 Random Test Data Generation

To generate test data, we rely on a randomizer, a sampling method that is in charge of generating a random value for a given realistic domain. The randomizer works with the realistic domain assignments provided in the `@requires` clauses of the contracts.

Assume that the precondition of a method specifies the realistic domain of parameter `i` as follows: `@requires i:` $t_1(\ldots)$ `or ... or` $t_n(\ldots)$`;`. When this method is randomly tested, a random value for `i` is generated in the domain of one of its n declared realistic domains. If $n \geq 2$, then a realistic domain $t_c(\ldots)$ is first selected among $t_1(\ldots)$, ..., $t_n(\ldots)$ by uniform random generation of c between 1 and n. Then, the randomizer generates a value of domain $t_c(\ldots)$ for i using the sampling method provided by this realistic domain.

When the data to be generated is an object of class C, the invariant of class C is analyzed in order to recover the realistic domains associated to the class attributes, and it recursively generates a data value for each class attribute. An instance of class C is created and the generated data values are assigned to the class attributes.

By default, the test case generation works by rejection of irrelevant values, as described in the simplified algorithm in Fig. 8. This algorithm has three parameters. The first one, *nbTests*, represents the number of tests the user wants to generate. The second parameter, *maxTries*, is introduced in order to ensure that the generator stops if all the yielded values are rejected. Indeed, the generator may fail to yield a valid value because the preconditions lay down too strong constraints. The third and last parameter, *methods*, represents the set of methods to test.

For many realistic domains this default test generation method can be overloaded by more efficient methods. We have already implemented the following enhancements:

– Generation by rejection is more efficient when the rejection probability is low. Therefore, the hierarchy of realistic domains presented in Section 2.2 is constructed so that each class restricts the domain of its superclass as little as possible, and rejection always operates w.r.t. this direct superclass for more efficiency.

```
function generateTests($nbTests$, $maxTries$, $methods$)
begin
  $tests \leftarrow 0$
  do
    $tries \leftarrow 0$
    $f \leftarrow$ select method under test amongst $methods$
    do
      $tries \leftarrow tries + 1$
      for each parameter $p$ of $f$ do
        $t \leftarrow$ select realistic domain of $p$
        $i \leftarrow$ generate random value $\in [1..card(t)]$
        $v \leftarrow i^{th}$ value of realistic domain $t$
        assign value $v$ to parameter $p$
      done
    while $f$ precondition fails $\wedge$ $tries \leq maxTries$
    if $tries \leq maxTries$ then
      run test case / keep test case
      $tests \leftarrow tests + 1$
    end if
  while $tests \leq nbTests$
end
```

Fig. 8. Test Cases Generation Algorithm

– Realistic domains representing intervals are sampled without rejection by a direct call to a uniform random generator, in fact two generators: a discrete and a continuous one.

The users are free to add their own enhancements and we plan to work in this direction in a near future.

5 Tool Support and Experimentation

The Praspel tool implements the principles described in this paper. Praspel is freely available at `http://hoa-project.net/`. It is developed in Hoa [12] (since release 0.5.5b), a framework and a collection of libraries developed in PHP. Hoa combines a modular library system, a unified streams system, generic solutions, etc. Hoa also aims at bridging the gap between research and industry, and tends to propose new solutions from academic research, as simple as possible, to all users. Praspel is a native library of Hoa (by opposition to user libraries). This deep integration ensures the durability and maintenance of Praspel. It is provided with an API documentation, a reference manual, a dedicated forum, etc. Moreover, we get feedbacks about the present research activity through Hoa user community.

A demonstration video of the Praspel tool is also available at the following address: `http://hoa-project.net/Video/Praspel.html`. As shown in this demonstration, the Praspel language is easy to learn and to use for developers. The tool support makes it possible to produce test data and run test cases in a dedicated framework with little effort. Praspel has been presented and demonstrated at the ForumPHP'10 conference, which gathers the PHP community, and was received very favourably.

We applied Praspel to web application validation. In such applications, PHP functions are used to produce pieces of HTML code from simple inputs. Being also teachers, we have decided to test the (buggy!) code of our students of the "Web Languages" course. Using a dedicated library of Hoa to build LL(k) parsers, we easily designed the realistic domains associated with the expected output HTML code, that had to be well-structured, and respecting some structural constraints (e.g. exactly one `<option>` of a `<select>` has to be set by default). We analysed their functions using this mechanism.

To save space, we have reported this experimentation on a web page at the following address: `http://lifc.univ-fcomte.fr/home/~fdadeau/praspel.html`, along with the links to download the Hoa framework (implementing Praspel), and some samples of the possibilities of Praspel.

6 Related Works

Various works consider Design-by-Contract for unit test generation [6,9,10,14,17]. Our approach is inspired by the numerous works on JML [15]. Especially, our test verdict assignment process relies on Runtime Assertion Checking, which is also considered in JMLUnit [9], although the semantics on exceptions handling differs. Recently, JSConTest [14] uses Contract-Driven Testing for JavaScript. We share a common idea of adding types to weakly typed scripting languages (JavaScript vs PHP). Nevertheless our approach differs, by considering flexible contracts, with type inheritance, whereas JSConTest considers basic typing informations on the function profile and additional functions that must be user-defined. As a consequence, due to a more expressive specification language, Praspel performs more general runtime assertion checks. ARTOO [10] (Adaptative Random Testing for Object-Oriented software) uses a similar approach based on random generation involving contracts written in Eiffel [18], using the AutoTest tool. ARTOO defines the notion of distance between existing objects to select relevant input test data among the existing objects created during the execution of a program. Our approach differs in the sense that object parameters are created on-the-fly using the class invariant to determine relevant attributes values, in addition to the function precondition. Praspel presents some similarities with Eiffel's types, especially regarding inheritance between realistic domains. Nevertheless, realistic domains display the two properties of predicability and samplability that do not exist in Eiffel. Moreover, Praspel adds clauses that Eiffel contracts do not support, as `@throwable` and `@behavior`, which are inspired from JML.

Our test generation process, based on random testing, is similar to Jartege [20] for JML. Jartege is able to generate a given number of tests sequences of a given length, for Java programs with JML specifications. Jartege uses the type of input parameters to compute random values for method parameters, that are then checked against the precondition of the method, using an online test generation approach.

Also for JML, Korat [7] uses a user-defined boolean Java function that defines a valid data structure to be used as input for unit testing. A constraint solving approach is then used to generate data values satisfying the constraints given by this function, without producing isomorphic data structures (such as trees). Our approach also uses a similar way to define acceptable data (the predicate feature of the realistic domains). Contrary to Korat, which automates the test data generation, our approach also requires the user to provide a dedicated function that generates data. Nevertheless, our realistic domains are reusable, and Praspel provides a set of basic realistic domains that can be used for designing other realistic domains. Java PathFinder [23] uses a model-checking approach to build complex data structures using method invocations. Although this technique can be viewed as an automation of our realistic domain samplers, its application implies an exhaustive exploration of a system state space. Recently, the UDITA language [13] makes it possible to combine the last two approaches, by providing a test generation language and a method to generate complex test data efficiently. UDITA is an extension of Java, including non-deterministic choices and assumptions, and the possibility for the users to control the patterns employed in the generated structures. UDITA combines generator- and filter-based approaches (respectively similar to the sampler and characteristic predicate of a realistic domain).

Finally, in the domain of web application testing, the Apollo [2] tool makes it possible to generate test data for PHP applications by code analysis. The tests mainly aim at detecting malformed HTML code, checked by an common HTML validator. Our approach goes further as illustrated by the experimentation, as it makes it possible not only to validate a piece of HTML code (produced by a Praspel-annotated function/method), but also to express and check structural constraints on the resulting HTML code. On the other hand, the test data generation technique proposed by Apollo is of interest and we are now investigating similar techniques in our test data generators.

7 Conclusion and Future Works

In this paper, we have introduced the concept of realistic domain in order to specify test data for program variables. Realistic domains are not types but may replace them in weakly-typed languages, such as Web languages like PHP, our implementation choice. Realistic domains are a (realistic) trade-off between two extreme ways to specify preconditions: a too coarse one with types and a too fine one with first-order formulas. They provide two useful features for automated test generation: *predicability* and *samplability*. Predicability is the ability to check the realistic domain of a data at run time, whereas samplability provides a means to automatically generate data values matching a given realistic domain. Realistic domains are specified as annotations in the code of the language. We have implemented these principles on PHP, in a dedicated specification language named Praspel, based on the Design-by-Contract paradigm. This approach is implemented into a test generator that is able to (*i*) generate automatically unit test data for PHP methods, using a random data generator that produces input parameter values that satisfy the precondition, (*ii*) run the tests and (*iii*) check the postcondition to assign the test verdict. Praspel implements the two features of runtime assertion checking and test data generation independently. As a consequence, the user is not restricted to use a random test data generator and can write his own test data generator.

We are currently investigating the improvement of our data generator, to replace randomly generated values by a constraint-based approach [11] for which the realistic domains of input parameters would be used to define data domains. Constraint solving techniques will then be employed to produce test data that satisfy the precondition of the method, as in Pex [22]. We are also planning to consider a grey-box testing approach that would combine a structural analysis of the code of PHP methods, coupled with the definition of the realistic domains of their input parameters. Finally, we are interested in extending (and generalizing) the concept of realistic domains to other programming languages, especially those already providing a type system, to illustrate the benefits of this concept.

References

1. Aichernig, B.K.: Contract-based testing. In: Aichernig, B.K. (ed.) Formal Methods at the Crossroads. From Panacea to Foundational Support. LNCS, vol. 2757, pp. 34–48. Springer, Heidelberg (2003)
2. Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A., Ernst, M.D.: Finding bugs in dynamic web applications. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis, ISSTA 2008, pp. 261–272. ACM, New York (2008)
3. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
4. Baudin, P., Filliâtre, J.-C., Hubert, T., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI C Specification Language (preliminary design V1.2) (2008)
5. Beizer, B.: Black-box testing: techniques for functional testing of software and systems. John Wiley & Sons, Inc., New York (1995)
6. Bouquet, F., Dadeau, F., Legeard, B.: Automated Boundary Test Generation from JML Specifications. In: Misra, J., Nipkow, T., Karakostas, G. (eds.) FM 2006. LNCS, vol. 4085, pp. 428–443. Springer, Heidelberg (2006)
7. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated Testing based on Java Predicates. In: ISSTA 2002: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 123–133. ACM, New York (2002)
8. Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., Veanes, M.: Testing concurrent object-oriented systems with spec explorer. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 542–547. Springer, Heidelberg (2005)
9. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: The JML and JUnit way. In: Deng, T. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 231–255. Springer, Heidelberg (2002)
10. Ciupa, I., Leitner, A., Oriol, M., Meyer, B.: ARTOO: Adaptive Random Testing for Object-Oriented Software. In: ICSE 2008: Proceedings of the 30th International Conference on Software Engineering, pp. 71–80. ACM, New York (2008)

11. DeMillo, R.A., Offutt, A.J.: Constraint-Based Automatic Test Data Generation. IEEE Trans. Softw. Eng. 17(9), 900–910 (1991)
12. Enderlin, I.: Hoa Framework project (2010), http://hoa-project.net
13. Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., Marinov, D.: Test generation through programming in UDITA. In: ICSE 2010: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, pp. 225–234. ACM, New York (2010)
14. Heidegger, P., Thiemann, P.: Contract-Driven Testing of JavaScript Code. In: Vitek, J. (ed.) TOOLS 2010. LNCS, vol. 6141, pp. 154–172. Springer, Heidelberg (2010)
15. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A notation for detailed design. In: Kilov, H., Rumpe, B., Simmonds, I. (eds.) Behavioral Specifications of Businesses and Systems, pp. 175–188. Kluwer Academic Publishers, Boston (1999)
16. Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the design of JML accommodates both runtime assertion checking and formal verification. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2002. LNCS, vol. 2852, pp. 262–284. Springer, Heidelberg (2003)
17. Madsen, P.: Unit Testing using Design by Contract and Equivalence Partitions. In: Marchesi, M., Succi, G. (eds.) XP 2003. LNCS, vol. 2675, pp. 425–426. Springer, Heidelberg (2003)
18. Meyer, B.: Eiffel: programming for reusability and extendibility. SIGPLAN Not. 22(2), 85–94 (1987)
19. Meyer, B.: Applying "design by contract". Computer 25(10), 40–51 (1992)
20. Oriat, C.: Jartege: A Tool for Random Generation of Unit Tests for Java Classes. In: Reussner, R., Mayer, J., Stafford, J.A., Overhage, S., Becker, S., Schroeder, P.J. (eds.) QoSA 2005 and SOQUA 2005. LNCS, vol. 3712, pp. 242–256. Springer, Heidelberg (2005)
21. PHP Group. The PHP website (2010), http://www.php.net
22. Tillmann, N., de Halleux, J.: Pex: White box test generation for.net. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)
23. Visser, W., Păsăreanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. SIGSOFT Softw. Eng. Notes 29(4), 97–107 (2004)

Using Testing Techniques for Vulnerability Detection in C Programs[⋆]

Amel Mammar[1], Ana Cavalli[1], Willy Jimenez[1], Wissam Mallouli[2], and Edgardo Montes de Oca[2]

[1] Télécom SudParis, SAMOVAR,
9 rue Charles Fourier, 91011 Evry Cedex, France
{amel.mammar,ana.cavalli,willy.jimenez}@it-sudparis.eu
[2] Montimage, 39 rue Bobillot Paris 75013, France
{wissam.mallouli,edgardo.montesdeoca}@montimage.com

Abstract. This paper presents a technique for vulnerability detection in C programs. It is based on a vulnerability formal model called "Vulnerability Detection Conditions" (VDCs). This model is used together with passive testing techniques for the automatic detection of vulnerabilities. The proposed technique has been implemented in a dynamic code analysis tool, TestInv-Code, which detects the presence of vulnerabilities on a given code, by checking dynamically the VDCs on the execution traces of the given program. The tool has been applied to several C applications containing some well known vulnerabilities to illustrate its effectiveness. It has also been compared with existing tools in the market, showing promising performances.

Keywords: Dynamic Code Analysis, Vulnerabilities Detection, Passive Testing.

1 Introduction

1.1 Context and Motivations

The detection of vulnerabilities[1] in software has become a major concern in the software industry. Although efforts are being made to reduce security vulnerabilities in software, according to published statistics, the number of vulnerabilities and the number of computer security incidents resulting from exploiting these vulnerabilities are growing [7].

One of the reasons for this is that information on known vulnerabilities is not easily available to software developers, or integrated into the tools they use. Thus many activities are designed to support secure software development like security education on vulnerability causes, security goal and vulnerability class identification, goal and vulnerability driven inspections etc. Vulnerability cause presence testing is one of

[⋆] The research leading to these results has received funding from the European ITEA-2 project DIAMONDS.

[1] In this paper, a vulnerability is defined as a specific instance of not intended functionality in a certain product/environment leading to degradation of security properties or violation of the security policy. It can be exploited by malicious code or misuse.

the main activities that support the validation of secure software. It is used to detect vulnerabilities in software products in order to remove/mitigate them. Several testing techniques can be used to perform this detection based on different models and approaches (static/dynamic code analysis, fuzz testing, active/passive testing, etc.). In this paper, we present a systematic approach to increase software security by bridging the gap between security experts and software practitioners. Indeed, we provide providing software developers with the means to effectively prevent and remove occurrences of known vulnerabilities when building software. To achieve this goal, we will rely on a formal method for dynamic code analysis technique based on vulnerability detection conditions (VDCs) models.

Currently, there are a large number of techniques and related tools that help developers improve software security quality. Among these techniques, we can cite formal verification and validation (V&V)[11] and also the static and dynamic code analyzers [20,16]. However, existing approaches are often limited and do not present rigorous descriptions of vulnerabilities they deal with [9,12,15]. It is quite difficult for a user to know which vulnerabilities are detected by each tool since they are poorly documented. A more detailed description of the related work is provided in section 2.

1.2 Contribution

Our approach combines a new formalism called *Vulnerability Detection Conditions* (VDCs) and formal passive testing in order to implement a new method to detect vulnerabilities in C programs. These two concepts are detailed respectively in sections 3 and 4.

A VDC allows to formally describe a vulnerability without ambiguity. This task is performed by a security expert that needs to study vulnerabilities then determine its **causes**. Each cause needs to be extracted and translated into a logical predicate on which it becomes possible to reason. In a second step, VDCs descriptions are instantiated by a dynamic analysis tool to allow the automatic detection of this vulnerability in any C program. The tool is based on passive testing technique, which has proven to be very effective for detecting faults in communication protocols [4]. In summary, the main contributions introduced by this paper are:

– A new formalism, called *Vulnerability Detection Conditions* (VDCs), is designed to describe vulnerability causes in a rigorous way without ambiguity. This formalism also constitutes a good way to have a good understanding of each software vulnerability and its causes. It bridges the gap between security experts, developers and testers.
– An editor tool to build new VDCs based on a set of know vulnerability causes described in the the SHIELDS SVRS[2].

[2] The SHIELDS SVRS is a centralized repository that allows the storage and sharing security models in order to reduce known security vulnerabilities during software development.

- A model-based dynamic analysis tool *TestInv-Code*[3] that automatically detects vulnerabilities in C programs based on VDCs;
- An end-to-end methodology that allows to detect vulnerabilities and provides for each detection information about the vulnerability, the different ways to avoid it and the C code line where the vulnerability occurs (if the code is available).
- Application of the approach and the obtained results on an open source application XINE that contains a known vulnerability.

The approach proposed in this paper is original since it covers all the steps of vulnerability detection, from the modelling phase relying on VDCs, to their automatic detection on the executable traces using the *TestInv-Code* tool.

The rest of the paper is organized as follows. The section 2 presents different approaches used in literature for dynamic detection of vulnerabilities. Section 3 introduces the VDC model, its basics and use. Section 4 introduces the dynamic code analysis technique based on these models and its tool TestInv-Code. Section 5 introduces the experimentation and results and Section 6 summarizes our work and describes future work.

2 Related Work

Different techniques have been proposed to perform dynamic detection of vulnerabilities [3]. *Fuzz testing* is an approach that has been proposed to improve the security and reliability of system implementations [14]. Fuzz testing consists in stimulating the system under test, using random inputs or mutated ones, in order to detect unwanted behavior as crashing or confidentiality violation. *Penetration testing* is another technique that consists in executing a predefined test scenario with the objective to detect design vulnerabilities or implementation vulnerabilities [22]. *Fault injection* is a similar technique that injects different types of faults in order to test the behavior of the system [10]. Following a fault injection the system behavior is observed. The failure to tolerate faults is an indicator of a potential security flaw in the system. These techniques have been applied in industry and shown to be useful. However, most of the current detection techniques based on these approaches are ad hoc and require a previous knowledge of the target systems or existing exploits.

Model checking techniques have also been revisited for vulnerability detection. Hadjidj et al.[13] present a security verification framework that uses a conventional push down system model checker for reachability properties to verify software security properties. Wang et al. [23] have developed a constraint analysis combined with model checking in order to detect buffer overflow vulnerabilities. The memory size of buffer-related variables is traced and the code is instrumented with constraints assertions before the potential vulnerable points. The vulnerability is then detected with the reachability of the assertion using model checking. All model checking works are based on the design of a model of the system, which can be complex and subject to the combinatorial explosion of the number of states.

[3] TestInv-Code testing tool is one of Montimage tools (`http://www.montimage.com`). It is a dynamic code analysis tool that aims at detecting vulnerabilities by analyzing the traces of the code while it is executing.

In the *dynamic taint approach* proposed by Chess and West [8], tainted data are monitored during the execution of the program to determine its proper validation before entering sensitive functions. It enables the discovery of possible input validation problems which are reported as vulnerabilities. The sanitization technique to detect vulnerabilities due to the use of user supplied data is based on the implementation of new functions or custom routines. The main idea is to validate or sanitize any input from the users before using it inside a function. Balzarotti et al. [2] present an approach using static and dynamic analysis to detect the correctness of sanitization process in web applications that could be bypassed by an attacker.

3 Vulnerability Modelling

In order to describe the presence of a vulnerability in a program, we rely in this paper on Vulnerability Detection Conditions (VDCs) formalism. VDCs basically indicate that the execution of an action under certain conditions could be dangerous or risky for the program. They permit to express in a logical predicate the different causes that lead to the considered vulnerability. The main idea behind the definition of the VDC formalism is to point out the use of a dangerous action under some particular conditions, for instance "it is dangerous to use unallocated memory". Thus, if we evaluate a piece of code where we find such VDC we know that it is vulnerable.

3.1 Definitions

Definition 1. *(Vulnerability Detection Condition). Let Act be a set of action names, Var be a set of variables, and P be a set of predicates on $(Var \cup Act)$. We say that Vdc is a vulnerability detection condition if Vdc is of the form (long brackets denote an optional element):*

$$Vdc ::= a/P(Var,Act)|a[/P(Var,Act)];P'(Var,Act)$$

where a denotes an action, $P(Var,Act)$ and $P'(Var,Act)$ represent any predicates on variables Var and actions Act. A vulnerability detection condition $a/P(Var,Act)$ means that action a occurs when specific conditions denoted by predicate $P(Var,Act)$ hold.

Similarly, a vulnerability detection condition $a[/P(Var,Act)];P'(Var,Act))$ means that action a used under the optional conditions $P(Var,Act)$ is followed by a statement whose execution satisfies $P'(Var,Act)$. Naturally, if action a is not followed by an action, the predicate $P'(Var,Act)$ is assumed to be true.

More complex vulnerability detection conditions can be built inductively using the different logical operators according to the following definition.

Definition 2. *(General Vulnerability Detection Conditions). If Vdc_1 and Vdc_2 are vulnerability detection conditions, then $(Vdc_1 \vee Vdc_2)$ and $(Vdc_1 \wedge Vdc_2)$ are also vulnerability detection conditions.*

3.2 Some Examples

Let us define a vulnerability detection condition Vdc_1 that can be used to detect possible accesses to a free or an unallocated memory. If we denote by $Assign(x,y)$ the assignment of value y to the memory variable x and *IsNot_Allocated* a condition to check if memory x is unallocated then the VDC is given by the expression:

$$Vdc_1 = Assign(x,y)/IsNot\_Allocated(x)$$

In programming languages like C/C++, there are some functions that might lead to a vulnerability if they are applied on out-of-bounds arguments. The use of a tainted variable as an argument to a memory allocation function (e.g. `malloc`) is a well-known example of such a vulnerability, which is expressed by the vulnerability detection condition Vdc_2 below. A variable is tainted if its value is obtained from a non-secure source, or in other words, produced by reading from a file, getting input from a user or the network, etc.

$$Vdc_2 = memoryAllocation(S)/tainted(S).$$

3.3 Describing Vulnerabilities with Formal Vulnerability Detection Conditions

An informal description of a vulnerability states the conditions under which the execution of a dangerous action leads to a possible security breach. So, it should include the following elements:

1. *A master action*: an action denotes a particular point in a program where a task or an instruction that modifies the value of a given object is executed. Some examples of actions are variable assignments, copying memory or opening a file. A master action *Act_Master* is a particular action that produces the related vulnerability.
2. A set of conditions: a condition denotes a particular state of a program defined by the value and the status of each variable. For a buffer, for instance, we can find out if it has been allocated or not. Once the master action is identified for a scenario, all the other facts are conditions $\{C_1, \ldots, C_n\}$ under which the master action is executed. Among these conditions, a particular condition C_k may exist, called *missing condition*, which must be satisfied by an action following *Act_Master*.

Let $\{P_1, \ldots, P_k, \ldots, P_n\}$ be the predicates describing conditions $\{C_1, \ldots, C_k, \ldots, C_n\}$. The formal vulnerability detection condition expressing this dangerous scenario is defined by:

$$Act/(P_1 \wedge \ldots \wedge P_{k-1} \wedge P_{k+1} \ldots \wedge P_n); P_k$$

Finally, the vulnerability detection condition representing the entire vulnerability is defined as the disjunction of the all sub-vulnerability detection conditions for each scenario (Vdc_i denotes the VDC associated with each scenario i):

$$Vdc_1 \vee \ldots \vee Vdc_n$$

For example, consider the vulnerability CVE-2009-1274, a buffer overflow in *XINE* media player. According to the description, the vulnerability is the result of computing a buffer size by multiplying two user-supplied integers without previously checking the operands or without checking the result of the allocation. An attacker may cause the execution of arbitrary code by providing a specially crafted media file to the user running the *XINE* application. A scenario associated to this vulnerability can be expressed as:

1. An allocation function is used to allocate a buffer
2. The allocated buffer is not adaptive
3. The size used for that allocation is calculated using tainted data (data read from the media file)
4. The result returned by the allocation function is not checked

To define the VDC associated with this scenario, we have to express each of these conditions with a predicate:

Use of malloc/calloc/realloc the program uses C-style memory management functions, such as `malloc`, `calloc` or `realloc` to allocate memory. For each memory function allocation f, applied on value V to allocate a buffer B, the following predicate holds:

$$memoryAllocation(f, B, V)$$

Use of nonadaptive buffers the program uses buffers whose sizes are fixed when they are allocated (allocation may take place at run-time, e.g. `malloc`, or at compile-time). Non-adaptive buffers can only hold a specific amount of data; attempting to write beyond their capacity results in a buffer overflow. Adaptive buffers, in contrast, can adapt themselves to the amount of data written to them. For each declared nonadaptive buffer B, the following predicate holds:

$$nonAdaptiveBuffer(B)$$

User supplied data influences buffer size the size of a dynamically allocated buffer is computed, at least in part, from user-supplied data. This allows external manipulation of the buffer size. If a buffer is made too large, this may result in a denial of service condition; if it is too small, then it may later result in a buffer overflow. For each variable V whose value is produced from an insecure source, the following predicate holds:

$$tainted(V)$$

Note that a tainted variable will be untainted if it is bound checked by the program.

Failed to check return value from calloc the program does not contain mechanisms to deal with low memory conditions in a safe manner (i.e. deal with NULL return values from `calloc`). Running out of memory in programs that are not written to handle such a situation may result in unpredictable behavior that can possibly be exploited. This cause is detected when the return value B of an allocation function is not followed by a check statement. For each value B returned from an allocation memory function, the following formula is defined:

$$notChecked(B, null)$$

The vulnerability detection condition expressing this scenario is then defined by:

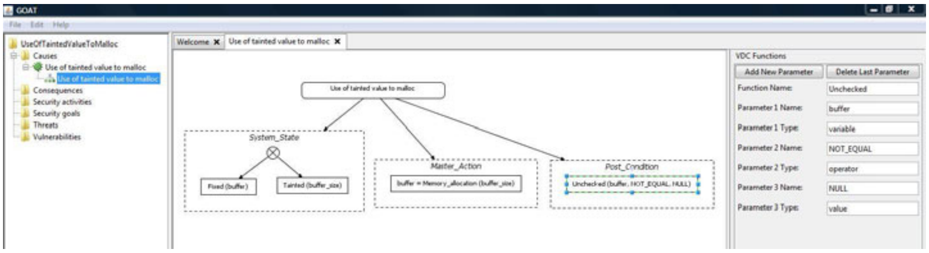
$$memoryAllocation(f, B, V) / \begin{pmatrix} nonAdaptiveBuffer(B) \\ \wedge \\ tainted(V) \end{pmatrix} ; notChecked(B, null)$$

This last vulnerability detection condition expresses a potential vulnerability when a given allocation function f is used with a non-adaptive buffer B whose size V is produced from an insecure source and its return value is not checked with respect to NULL.

3.4 VDC Editor

The VDC editor is a GOAT[4] plug-in, which offers security experts the possibility to create vulnerability detection conditions (VDCs). These VDCs will be used to detect the presence of vulnerabilities by checking software execution traces using Montimage TestInv-Code testing tool. The VDC editor user interface includes some features that allow simplifying the construction and composition of VDCs. The VDC editor has the following functionalities:

– The creation of new VDCs corresponding to vulnerability causes from scratch and their storage in an XML format.
– The visualization of already conceived VDCs.
– The editing (modification) of existing VDCs in order to create new ones.



Function Name: Unchecked
Parameter 1 Name: buffer
Parameter 1 Type: variable
Parameter 2 Name: NOT_EQUAL
Parameter 2 Type: operator
Parameter 3 Name: NULL
Parameter 3 Type: value

Fig. 1. Vulnerability detection condition for "Use of tainted value to malloc" in GOAT

The VDCs are stored in an XML file that constitutes one of the inputs for the Montimage TestInv-Code tool. A vulnerability is discovered if a VDC signature is detected on the execution trace. A VDC is composed within the editor of at most 3 parts:

1. *Master condition:* The triggering condition called also master action (denoted a). When analysing the execution trace, if this condition is detected, we should verify if the state and post conditions of the VDC hold as well. If this is the case, then a vulnerability has been detected. The master condition is mandatory in a VDC.

[4] `http://www.ida.liu.se/divisions/adit/security/goat/`

2. *State condition:* A set of conditions related to the system state (denoted P(Var,Act)). The state condition describes the states of the specified variables at the occurrence of the master action. The state condition is mandatory in a VDC.
3. *Post condition:* A set of conditions related to the system future state (denoted P′(Var,Act)). If a master action is detected in the state condition context, then we should verify if the post condition holds in the execution that follows. If this is the case, a vulnerability has been detected. This post condition is not mandatory in a VDC.

4 Dynamic Code Analysis for Vulnerability Detection

4.1 Basics: Passive Testing

Our approch for dynamic code analysis is inspired from the classical passive testing technique [1,19,17] designed for telecommunication traffic analysis. Passive testing allows to detect faults and security flaws by examining captured trafic packets (live trafic or log files) according to a set of events-based properties that denote either:

– a set of functional or security rules that the trafic has to fulfill[4,5,18], or
– a set behavioral attacks like those used in classical intrusion and detection systems.

In the case of executable code analysis, events are assimilated to the disassembled instructions that are being executed in the processor. They are produced by executing the program under the control of the *TestInv-Code* tool, similar to what a debugger does.

For dynamic program analysis to be effective, the target program must be executed with sufficient test inputs to cover different program behaviours. Use of classical testing techniques for code coverage helps to ensure that an adequate part of the program's set of possible behaviours has been observed. Also, care must be taken to minimize the effect that instrumentation has on the execution (including temporal properties) of the target program.

While static analysis collects information based on source code, dynamic analysis is based on the system execution (binary code), often using instrumentation. The advantages that can be expected from using dynamic analysis are:

– Has the ability to detect dependencies that are not detectable in static analysis. Ex.: dynamic dependencies using reflection, dependency injection etc.
– Allows the collection of temporal information.
– Allows the possibility of dealing with runtime values.
– Allows the identification of vulnerabilities in a runtime environment.
– Allows the use of automated tools to provide flexibility on what to scan for.
– Allows the analysis of applications for which you do not have access to the actual code.
– Allows identifying vulnerabilities that might be false negatives in the static code analysis.
– Permits validating static code analysis findings.
– It can be conducted on any application.

4.2 Using VDCs in *TestInv-Code*

In order to use the *TestInv-Code* tool, the main step consists in defining the vulnerabilities causes that are of interest. Starting from informal descriptions of the vulnerabilities and VDCs models, a set of conditions that lead to a vulnerability are derived. These conditions are formally specified as regular expressions that constitute the first input for *TestInv-Code* tool.

Thus, end-to-end code analysis using *TestInv-Code* proceeds along the following steps:

1. *Informal definition of vulnerable scenarios.* A security expert describes the different scenarios under which a vulnerability may appear. A scenario denotes a set of causes that produces the vulnerability.
2. *Definition of VDC.* A VDC, expressing formally the occurrence of the related vulnerability, is created for each possible situation that leads to the vulnerability using the VDC editor.
3. *Vulnerability checking.* Finally, *TestInv-Code* checks for evidence of the vulnerabilities during the execution of the program. Using the VDCs, it will analyze the execution traces to produce messages identifying the vulnerabilities found, if any, indicating where they are located in the code.

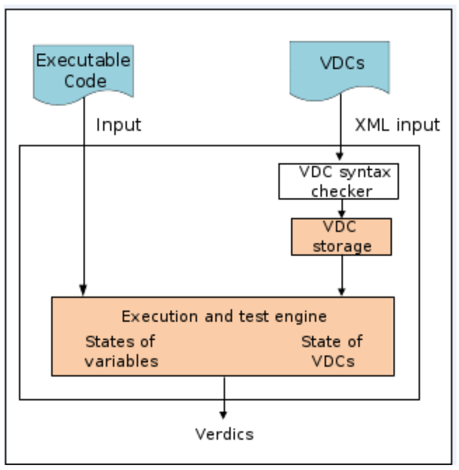


Fig. 2. Passive testing for vulnerability detection

Figure 2 depicts the passive testing architecture for vulnerability detection. As shown, the *TestInv-Code* tool takes as input:

1. *The vulnerability causes.* The file containing the vulnerabilities causes formally specified using VDCs.

2. *The executable.* The Executable Linked Format (ELF) file for the application that is to be tested. This file contains the binary code of the application and it should include debug information if we want the tool to be able to determine the line of code where the vulnerability occurs and provide this information in the final verdicts.

In order to detect the presence of a VDC in an execution trace, it needs to be processed in such a way that it is detected when and if it occurs during the execution of the program. In the case of *TestInv-Code*, predicates and actions in the VDCs correspond to functions that allow analysing the executed instructions and determining it they are satisfied. The tool keeps information on the state of all the variables used by the program, heap or stack memory addresses and registers. The states, are for instance, tainted or not, bound checked or not, allocated or not etc. It also maintains information on the potential VDCs. The tool is able to detect when a system call is made, the controls that are made on variables or return values from function calls, when buffer allocations are made, etc. Thus it can verify all the conditions that are used in the VDCs and generate messages if the VDCs are satisfied. The symbolic tables are necessary to be able to determine the line of code that provokes the vulnerability that is detected.

It must be noted that the functions used to detect the VDC conditions could vary depending on the execution environment, the compiler and the compilation options used. In this work we assume that the execution environment is Linux version 2.6, the compiler is gcc version 4.3.3 and that the compilation was performed for debugging (including symbolic tables) and without any optimisations. Other variants could work but this has not yet been tested on other platforms. Certain optimizations performed by the compiler could make it necessary to adapt the algorithms of the functions to certain particularities introduced by the compiler.

5 Experiment and Results

5.1 XINE Application

We demonstrate the application of our vulnerability detection method to an open source application and free multimedia player that plays back audio and video, XINE[5] written in C. This application was selected as an example since it is a real world application, open source (so the source files are available free of copyright), and contains a number of known vulnerabilities which can be used to demonstrate the effectiveness of our approach.

The application contains a set of modules and librairies. The one we are concentrated on is *xine-lib*[6] (xine core). This is a module developed in C language and which has several vulnerabilities inside its files. We will select an obsolete version of xine-lib so we can use the vulnerabilities found in them.

[5] `http://www.xine-project.org`

[6] Xine-lib source code can be downloaded from: `http://sourceforge.net/projects/xine`

5.2 Xine Selected Vulnerability

The xine v1.1.15 application has a number of vulnerabilities. The one that we will deal with is CVE-2009-1274.

- Summary: Integer overflow in the qt_error parse_trak_atom function in de-muxers/demux_qt.c in xine-lib 1.1.16.2 and earlier allows remote attackers to execute arbitrary code via a Quicktime movie file with a large count value in an STTS atom, which triggers a heap-based buffer overflow.
- Published: 04/08/2009
- CVSS Severity: 5.0 (MEDIUM)

The exploitation occurs when someone is trying to play with xine a Quicktime encoded video that an attacker has modified to make one of its building blocks (the “time to sample" or STTS atom) have an incorrect value. The malformed STTS atom processing by xine leads to an integer overflow that triggers a heap-based buffer overflow probably resulting in arbitrary code execution. The patch to this Vulnerability is in v1.1.16.1 that is also included in the v1.1.16.3.

CVE-2009-1274 is a vulnerability instance and can be considered as part of the family or class of vulnerabilities named “Integer Overflow" has the ID CWE 190 in the Common Weakness Enumeration database. The CWE 190 description is summarised as follows “The software performs a calcu-lation that can produce an integer overflow or wraparound, when the logic assumes that the resulting value will always be larger than the original value. This can introduce other weaknesses when the calculation is used for resource management or execution control" [21].

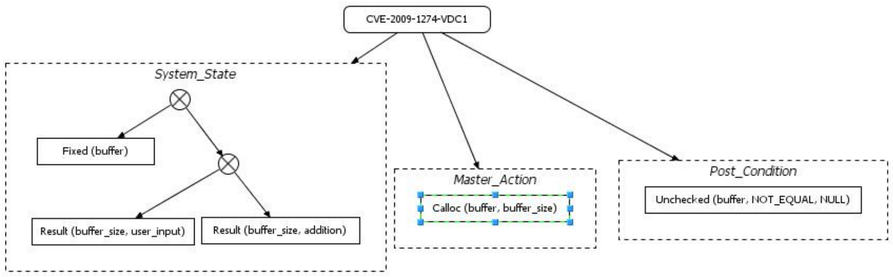

CVE-2009-1274-VDC1
System_State
Fixed (buffer)
Result (buffer_size, user_input)
Result (buffer_size, addition)
Master_Action
Calloc (buffer, buffer_size)
Post_Condition
Unchecked (buffer, NOT_EQUAL, NULL)

Fig. 3. VDC model of CVE-2009-1274 vulnerability

5.3 Vulnerability Modelling

Starting from the informal description of CVE-2009-1274 vulnerability, we have designed the 3 VDCs and the corresponding regular expressions to be used for input to the *TestInv-Code* tool.

1. Calloc(buffer, buffer_size) / Fixed(buffer) $\wedge$ Result(buffer_size, user_input)$\wedge$ Result(buffer_size, addition); Unchecked(buffer, NULL)
2. Calloc(buffer, buffer_size) / Fixed(buffer) $\wedge$ Result(buffer_size, user_input) $\wedge$ Result(buffer_size, addition) $\wedge$ Unchecked(buffer_size, buffer_bounds)
3. CopyVar(loop_counter, user_input) / Fixed(buffer) $\wedge$ Unchecked(loop_counter, counter_bounds); CopyData(buffer, user_input, loop_counter)

Using the VDC editor, we can build the VDC models for each cause scenario. Figure 3 illustrates the VDC model for the first scenario.

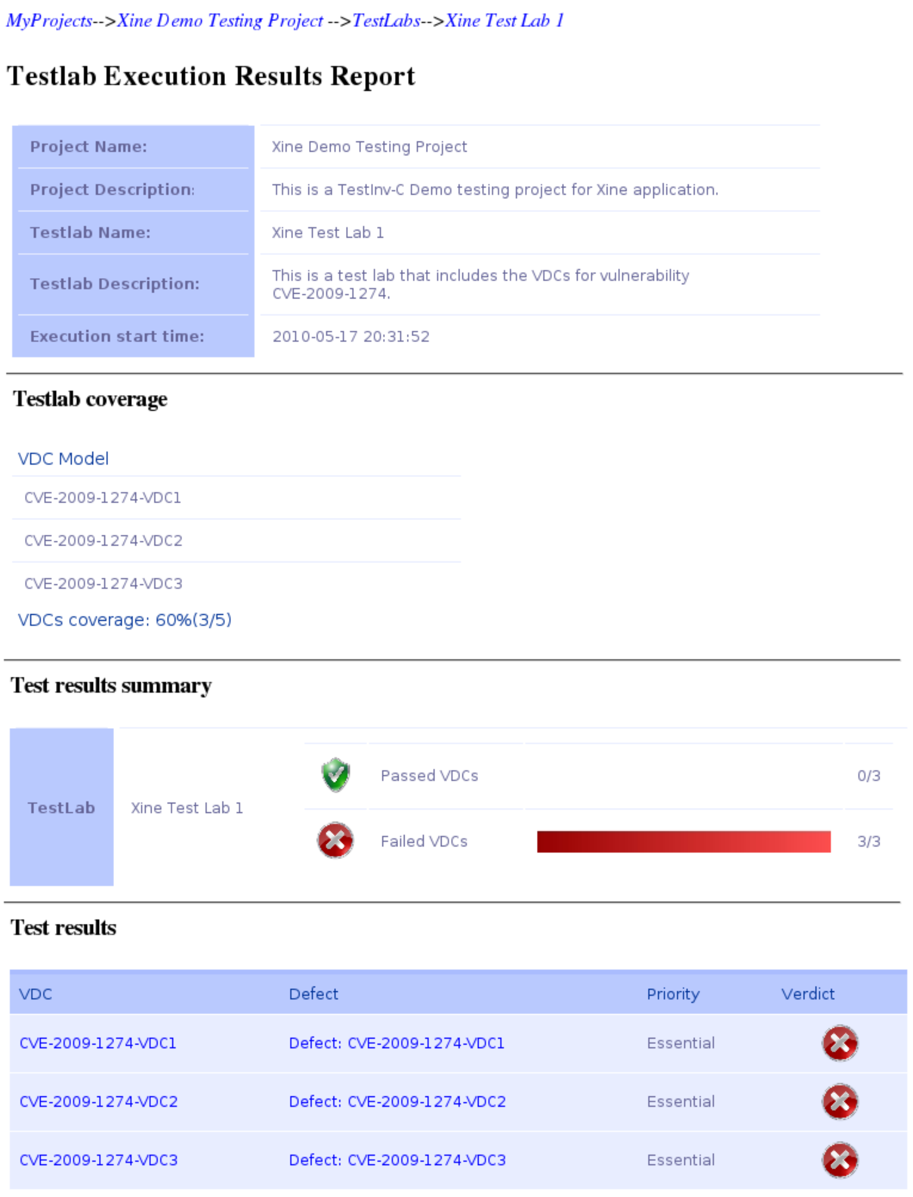
5.4 Application of TestInv-Code

The created VDCs are one of the inputs needed by the TestInv-C testing tool. In order to analyse the xine-lib it is necessary to use it. To be able to reach the plug-in that contains the error (the quicktime file demuxer), the muxine application was run on a quicktime file. The TestInv-Code tool allows performing the analysis on all the application's functions (including those of the library and the plug-ins). The user can also identify a given function or set of functions that he wants to analyse. Using this feature is necessary to avoid performance issues, particularly in applications that perform intensive data manipulations (like video players). The complete list of available functions can be obtained automatically. Another feature that helps improve the performance of the tool is the possibility of limiting the number of times a piece of code in a loop is analysed. The following *XINE* code is executed:

```
Code fragment from demux_qt.c
...
1907 trak->time_to_sample_table = calloc(
1908  trak->time_to_sample_count+1,
         sizeof(time_to_sample_table_t));
1909 if (!trak->time_to_sample_table) {
1910   last_error = QT_NO_MEMORY;
1911   goto free_trak;
1912 }
1913
1914 /* load the time to sample table */
1915 for(j=0;j<trak->time_to_sample_count;j++)
...
```

where `trak->time_to_sample_table` is tainted since it is set from information taken from the external QuickTime file.

The tool will detect the particular vulnerability used here (CVE-2009-1274) when it is launched on the muxine application using a quicktime video file. This needs to be done using the option to analyse all the functions (of the application, the library and the plug-ins) or just the function parse_trak_atom in the quicktime plug-in. The result of the vulnerability cause presence testing activity provided by TestInv-Code is shown in figure 4.



MyProjects-->*Xine Demo Testing Project* -->*TestLabs*-->*Xine Test Lab 1*

Testlab Execution Results Report

Project Name:	Xine Demo Testing Project
Project Description:	This is a TestInv-C Demo testing project for Xine application.
Testlab Name:	Xine Test Lab 1
Testlab Description:	This is a test lab that includes the VDCs for vulnerability CVE-2009-1274.
Execution start time:	2010-05-17 20:31:52

Testlab coverage

VDC Model
CVE-2009-1274-VDC1
CVE-2009-1274-VDC2
CVE-2009-1274-VDC3

VDCs coverage: 60%(3/5)

Test results summary

TestLab	Xine Test Lab 1	✓	Passed VDCs		0/3
		✗	Failed VDCs	🟥	3/3

Test results

VDC	Defect	Priority	Verdict
CVE-2009-1274-VDC1	Defect: CVE-2009-1274-VDC1	Essential	✗
CVE-2009-1274-VDC2	Defect: CVE-2009-1274-VDC2	Essential	✗
CVE-2009-1274-VDC3	Defect: CVE-2009-1274-VDC3	Essential	✗

Fig. 4. Screenshot of TestInv-Code result for xine vulnerability

5.5 Analysis

If we apply the same VDCs to other code under the same programming environment, we will be able to detect the same types of vulnerabilities. For instance, we applied the same VDCs on ppmunbox, a program developed by Linköpings university to remove borders from portable pixmap image files (ppm) and we detected the same vulnerabilities.

This vulnerability is located in the ppmunbox.c file specifically in the following:

```
Code fragment from ppmunbox.c
...
/* Read the dimensions */
if(fscanf(fp_in,"%d%d%d",&cols,&rows &maxval)<3){
 printf("unable to read dimensions from PPM file");
 exit(1);
 }

/* Calculate some sizes */
pixBytes = (maxval > 255) ? 6 : 3;
rowBytes = pixBytes * cols;
rasterBytes=rows;rasterBytes=rowBytes*rows;

/* Allocate the image */
img = malloc(sizeof(*img));
img->rows = rows;
img->cols = cols;
img->depth = (maxval > 255)?2:1;
p = (void*)malloc(rasterBytes);
img->raster = p;

/* Read pixels into the buffer */
while (rows--) {
...
```

To illustrate the applicability and scalability of TestInv-Code, it has been applied to six different open source programs to determine if known vulnerabilities can be detected using a single model. The following paragraphs describe the vulnerabilities and give a short explanation of the results obtained. The results are summarized in table 1.

Table 1. Summary of results running TestInv-Code with VDC codes

Vulnerability	Software	Detected ?
CVE-2009-1274	Xine	Yes
Buffer overflow	ppmunbox	Yes
CVE-2004-0548	aspell	Yes (two)
CVE-2004-0557	SoX	Yes
CVE-2004-0559	libpng	Yes
CVE-2008-0411	Ghostscript	Yes

Besides, the application of the tool to the case study gave good performances. We did some experiments in order to check the scalability of the tool by the application of a high number of VDCs (more than 100) to a software data intensive (as in the case of video decoders). The tool performance remains good. We compared the performance of our tool according to known dynamic code analysis tools in the market like Dmalloc[7], DynInst [8], and Valgrind[9] and the results were comparable. Indeed, the detection based on our tool does not insert a big overhead (the execution time is almost equal to the programm execution time).

To optimize our analysis, the tool is being modified so that the user can select specific functions to check in the program. But in this case all the input parameters for this function are marked as tainted even if they are not. Another solution that is being studied is to only check the first iteration of loops in the program, thus avoiding to check the same code that is executed more than once.

At present, we have checked applications written in C, which do not have a complex architecture. We are now starting to experiment more complex applications with architectures that integrate different modules, plugins, pointers to function, variable number of parameters or mixing different programming languages.

6 Conclusions and Future Work

Security has become a critical part of nearly every software project, and the use of automated testing tools is recommended by best practices and guidelines. Our interest lies in defining a formalism, called *Vulnerability Detection Conditions*, to describe vulnerabilities so we can detect them using automated testing.

In this paper, we have also shown how a model-based dynamic code analysis tool, *TestInv-Code*, is used to analyze execution traces and determine if they show evidence of a vulnerability or not. VDCs can be very precise, we believe making it possible to detect vulnerabilities with a low rate of false positives. This is planned to be studied and demonstrated in future work.

Since the vulnerability models are separate from the tool, it is possible for any security expert to keep them up-to-date and to add new models or variants. It also becomes possible for the tool user to add e.g. product-specific vulnerabilities and using the tool to detect them. This is very different from the normal state of affairs, where users have no choice but to rely on the tool vendor to provide timely updates. Nevertheless, it should be noted that if new predicates or actions are required, the function that will allow to detect them needs to be added to the tool.

The work presented in this paper is part of the SHIELDS EU project [21], in which we have developed a shared security repository through which security experts can

[7] Dmalloc is a library for checking memory allocation and leaks. Software must be recompiled, and all files must include the special C header file dmalloc.h.

[8] DynInst is a runtime code-patching library that is useful in developing dynamic program analysis probes and applying them to compiled executables. Dyninst does not require source code or recompilation in general, however non-stripped executables and executables with debugging symbols present are easier to instrument.

[9] Valgrind runs programs on a virtual processor and can detect memory errors (e.g. misuse of malloc and free) and race conditions in multithread programs.

share their knowledge with developers by using security models. Models in the SHIELDS repository are available to a variety of development tools; *TestInv-Code* is one such tool.

Looking to the future, we plan on applying the methods presented here to various kinds of vulnerabilities in order to identify which predicates are required, and whether the formalism needs to be extended.

References

1. Alcalde, B., Cavalli, A.R., Chen, D., Khuu, D., Lee, D.: Network Protocol System Passive Testing for Fault Management: A Backward Checking Approach. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 150–166. Springer, Heidelberg (2004)
2. Balzarotti, D., Cova, M., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In: IEEE Symposium on Security & Privacy, pp. 387–401 (2008)
3. Bardin, S., Herrmann, P., Leroux, J., Ly, O., Tabary, R., Vincent, A.: The BINCOA Framework for Binary Code Analysis. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 165–170. Springer, Heidelberg (2011)
4. Bayse, E., Cavalli, A., Núnez, M., Zaidi, F.: A Passive Testing Approach Based on Invariants: Application to the Wap. Computer Networks and ISDN Systems 48(2), 247–266 (2005)
5. Cavalli, A.R., Gervy, C., Prokopenko, S.: New Approaches for Passive Testing using an Extended Finite State Machine Specification. Information & Software Technology 45(12), 837–852 (2003)
6. Cavalli, A.R., Vieira, D.: An Enhanced Passive Testing Approach for Network Protocols. In: ICN, ICONS, MCL, pp. 169–169 (2006)
7. CERT Coordination Center. CERT/CC statistics (accessed October 2007)
8. Chess, B., West, J.: Dynamic Taint Propagation: Finding Vulnerabilities without Attacking. Information Security Technical Report 13(1), 33–39 (2008)
9. Coverity. Prevent (accessed September 2008)
10. Du, W., Mathur, A.: Vulnerability Testing of Software System using Fault Injection. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000), Workshop on Dependability Versis Malicious Faults (2000)
11. Fenz, S., Ekelhart, A.: Verification, Validation, and Evaluation in Information Security Risk Management. IEEE Security and Privacy (IEEESP) 9(2), 58–65 (2011)
12. Fortify Software. Fortify SCA (accessed September 2008)
13. Hadjidj, R., Yang, X., Tlili, S., Debbabi, M.: Model Checking for Software Vulnerabilities Detection with Multi-Language Support. In: Sixth Annual Conference on Privacy, Security and Trust, pp. 133–142 (2008)
14. Howard, M.: Inside the Windows Security Push. In: IEEE Symposium on Security & Privacy, pp. 57–61 (2003)
15. Klocwork. K7 (accessed September 2008)
16. Kuang, C., Miao, Q., Chen, H.: Analysis of Software Vulnerability. In: ISP 2006: Proceedings of the 5th WSEAS International Conference on Information Security and Privacy, pp. 218–223. World Scientific and Engineering Academy and Society (WSEAS), Stevens Point (2006)
17. Lee, D., Netravali, A.N., Sabnani, K.K., Sugla, B., John, A.: Passive Testing and Applications to Network Management. In: Proceedings of the 1997 International Conference on Network Protocols (ICNP 1997). IEEE Computer Society, Washington, DC (1997)

18. Mallouli, W., Bessayah, F., Cavalli, A., Benameur, A.: Security Rules Specification and Analysis Based on Passive Testing. In: The IEEE Global Communications Conference, GLOBECOM 2008 (2008)
19. Miller, R.E., Arisha, K.A.: Fault Identification in Networks by Passive Testing. In: Advanced Simulation Technologies Conference (ASTC), pp. 277–284. IEEE Computer Society, Los Alamitos (2001)
20. Redwine, S., Davis, N.: Processes to Produce Secure Software (2004); Task Force on Security Across the Software Development Lifecycle, Appendix A
21. SHIELDS: Detecting Known Security Vulnerabilities from within Design and Development Tools. D1.4 Final SHIELDS approach guide
22. Thompson, H.: Application of Penetration Testing. In: IEEE Symposium on Security & Privacy, pp. 66–69 (2005)
23. Wang, L., Zhang, Q., Zhao, P.: Automated Detection of Code Vulnerabilities Based on Program Analysis and Model Checking. In: Eighth IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 165–173 (2008)

Learning-Based Testing for Reactive Systems Using Term Rewriting Technology

Karl Meinke and Fei Niu

School of Computer Science and Communication,
Royal Institute of Technology, 100-44 Stockholm, Sweden
karlm@nada.kth.se, niu@csc.kth.se

Abstract. We show how the paradigm of learning-based testing (LBT) can be applied to automate specification-based black-box testing of reactive systems using term rewriting technology. A general model for a reactive system can be given by an extended Mealy automata (EMA) over an abstract data type (ADT). A finite state EMA over an ADT can be efficiently learned in polynomial time using the CGE regular inference algorithm, which builds a compact representation as a complete term rewriting system. We show how this rewriting system can be used to model check the learned automaton against a temporal logic specification by means of narrowing. Combining CGE learning with a narrowing model checker we obtain a new and general architecture for learning-based testing of reactive systems. We compare the performance of this LBT architecture against random testing using a case study.

1 Introduction

Learning-based testing (LBT) is an emerging technology for *specification-based black-box testing* that encompasses the three essential steps of : (1) test case generation (TCG), (2) test execution, and (3) test verdict (the oracle step). It has been successfully applied to testing *procedural systems* in [13] and [15], and *reactive systems* in [16]. The basic idea of LBT is to automatically generate a large number of high-quality test cases by combining a model checking algorithm with an *incremental model inference algorithm*, and integrating these two with the system under test (SUT) in an iterative loop. The use of incremental learning is critical in making this technology both fast and scalable to large systems under test (SUTs). Our previous research ([15] and [16]) has repeatedly shown that LBT has the capability to significantly outperform random testing in the speed with which it finds errors in an SUT.

For testing complex embedded software systems, there is a significant need to generate test cases over *infinite data types* such as integer and floating point types, and *abstract data types* (ADTs) such as strings, arrays, lists and various symbolic data types. Specification-based TCG is essentially a constraint solving problem. So this generalisation from finite to infinite and symbolic data types is highly non-trivial since the satisfiability problem for many logics over abstract and infinite data types is undecidable. Thus a search for test cases is not guaranteed to terminate.

Model checking over abstract and infinite data types is therefore a state of the art problem. Recently some success has been achieved with the use of satisfiability modulo theories (SMT) solvers such as Z3 [5], which are based on heuristic techniques. However, an alternative approach is to use constraint solving based on a *narrowing algorithm*. Narrowing is a flexible technology, based on term rewriting, which is applicable to any data type for which we can find a complete (confluent and terminating) term rewriting system (see e.g. [1]). It has well understood theoretical properties such as completeness of solutions and conditions for termination. Narrowing has been successfully applied to model checking of infinite state systems in [6]. However, the use of narrowing for test case generation has not yet been considered. In fact, our aim in this paper is much wider. We will show that narrowing combines easily with symbolic learning algorithms for automata such as CGE [14] to yield a new *LBT architecture for specification-based testing of reactive systems computing over abstract data types*. Initial case studies suggest that despite the significant increase in the problem complexity, this new LBT architecture is also competitive with random testing.

The structure of this paper is as follows. In the remainder of Section 1 we review related work. In Section 2, we recall some essential mathematical preliminaries needed to discuss narrowing. In Section 3, we formalise a general model of a reactive system as an *extended Mealy automaton (EMA) over an abstract data type* (ADT). We introduce a *linear time temporal logic* (LTL) for such EMA, and we show how an LTL formula can be translated into constraint sets consisting of equations and negated equations. In Section 4, we present a model checking algorithm based on narrowing applied to a constraint set. In Section 5, we combine this model checking method with a symbolic automata learning algorithm (the CGE learning algorithm of [14]) to define a new LBT architecture for specification-based testing of reactive systems. In Section 6, we present a case study of this LBT architecture applied to testing the TCP protocol. Finally, in Section 7 we draw some conclusions and discuss open questions to be addressed by future work.

1.1 Related Work

In [16], LBT was applied to testing reactive systems modeled as Boolean Kripke structures. Our work here extends this previous work to allow symbolic and infinite data types. Even for finite data types, this approach simplifies the expression of control and data properties of an SUT. For this extension we use a more powerful symbolic learning algorithm, new model checking technology based on term rewriting theory, and a more powerful oracle construction for test verdicts.

Several previous studies, (for example [19], [9] and [20]) have considered a combination of learning and model checking to achieve testing and/or formal verification of reactive systems. Within the model checking community the verification approach known as *counterexample guided abstraction refinement*

(CEGAR) also combines learning and model checking, (see e.g. [3] and [2]). The LBT approach described here can be distinguished from these other approaches by: (i) an emphasis on testing rather than verification, (ii) the focus on incremental learning for efficient scalable testing, (iii) the use of narrowing as a model checking technique, and (iv) the introduction of abstract data types.

There is of course an extensive literature on the use of model checkers (without learning) to generate test cases for reactive systems. A recent survey is [8]. Generally this work emphasizes glass-box testing (so no learning is necessary), and the use of structural coverage measures to constrain the search space for test cases. Furthermore, behavioral requirements may or may not be present. By contrast, the LBT approach concerns black-box testing. Furthermore, in LBT behavioral requirements are always present, both to solve the oracle problem and to constrain the search space and guide the search for effective test cases.

In [21], black-box reactive system testing using learning but without model checking is considered. This is also shown to be more effective than random testing. Thus we can conclude that learning and model checking are two mutually independent techniques that can be applied to systems testing separately or together. In the long term we hope to show that the combination of both techniques is ultimately more powerful than using either one alone.

2 Mathematical Preliminaries and Notation

It is helpful to have some familiarity with the theories of abstract data types and term rewriting. Both use the notation and terminology of many-sorted algebra (see e.g. [17]). Let S be a finite set of sorts or types. An *S-sorted signature* Σ consists of an $S^* \times S$-indexed family of sets $\Sigma = \langle \Sigma_{w,s} \mid w \in S^*,\ s \in S \rangle$. For the empty string $\varepsilon \in S^*$, $c \in \Sigma_{\varepsilon,s}$ is a *constant symbol* of sort s. For $w = s_1, \ldots, s_n \in S^+$, $f \in \Sigma_{w,s}$ is a *function symbol* of *arity* n, *domain type* w and *codomain type* s. An *S-sorted Σ-algebra* A consists of sets, constants and functions that interpret Σ by a particular semantics. Thus A has an S-indexed family of sets $A = \langle A_s \mid s \in S \rangle$, where A_s is termed the *carrier set* of sort s. For each $s \in S$ and constant symbol $c \in \Sigma_{\varepsilon,s}$, $c_A \in A_s$ is a constant, and for each $w = s_1, \ldots, s_n \in S^+$ and each $f \in \Sigma_{w,s}$, $f_A : A_{s_1} \times \ldots \times A_{s_n} \rightarrow A_s$ is a function. Let $X = \langle X_s \mid s \in S \rangle$ be an S-indexed family of disjoint sets X_s of variables of sort s. We assume $X_s \cap \Sigma_{\varepsilon,s} = \emptyset$. The set $T(\Sigma,\ X)_s$ of all *terms of sort* $s \in S$ is defined inductively by: (i) $c \in T(\Sigma,\ X)_s$ for $c \in \Sigma_{\varepsilon,s}$, (ii) $x \in T(\Sigma,\ X)_s$ for $x \in X_s$, and (iii) $f(t_1, \ldots, t_n) \in T(\Sigma,\ X)_s$ for $f \in \Sigma_{w,s}$ $w = s_1, \ldots, s_n$ and $t_i \in T(\Sigma,\ X)_{s_i}$ for $1 \leq i \leq n$. We use $\equiv$ to denote syntactic equality between terms. An *equation* e (respectively *negated equation*) over Σ and X is a formula of the form $(t = t')$ (respectively $\neg(t = t')$) for $t,\ t' \in T(\Sigma,\ X)_s$. We let $Vars(t)$ (respectively $Vars(e)$, $Vars(\neg(e))$) denote the set of all variables from X occurring in t (respectively e, $\neg e$).

A *variable assignment* $\alpha : X \rightarrow A$ is an S-indexed family of mappings $\alpha_s : X_s \rightarrow A_s$. A *substitution* σ is a variable assignment $\sigma : X \rightarrow T(\Sigma,\ X)$ such that $\sigma_s(x) \neq x$ for just finitely many $s \in S$ and variables $x \in X_s$, and this set of variables is the *domain* of σ_s. The result of applying a substitution σ to a term $t \in T(\Sigma, X)_{s'}$ is defined inductively in the usual way and denoted by $\sigma(t)$. If σ and σ' are substitutions then their *composition* $\sigma \circ \sigma'$ is defined by $\sigma \circ \sigma'(x) = \sigma(\sigma'(x))$. A *variable renaming* is a family of bijective substitutions $\sigma_s : X_s \to X_s$. A substitution σ is *more general* than a substitution τ, denoted $\sigma \leq \tau$ if there exists a substitution δ such that $\delta \circ \sigma = \tau$.

A *disunification problem* $S = \{ (t_i\ Q_i\ t'_i) \mid i = 1, \ldots, n,\ n \geq 0,\ Q_i \in \{ =, \neq \} \}$ is a finite (possibly empty) set of equations and negated equations over Σ and X. A substitution $\sigma : X \to T(\Sigma, X)$ is a *syntactic unifier* of a set S if for all $1 \leq i \leq n$, if Q_i is $=$ then $\sigma(t_i) \equiv \sigma(t'_i)$, and if Q_i is $\neq$ then $\sigma(t_i) \not\equiv \sigma(t'_i)$. We let $U(S)$ denote the set of all syntactic unifiers of S. A unifier $\sigma \in U(S)$ is a *most general unifier* (mgu) if $\sigma \leq \tau$ for all $\tau \in U(S)$.

If $t \in T(\Sigma, X)_s$ is a term then $O(t)$ denotes the set of all *positions* in t, i.e. all nodes in the parse tree of t and is inductively defined by $O(c) = O(x) = \{ \varepsilon \}$ and $O(f(t_1, \ldots, t_n)) = \{ \varepsilon, k.\bar{i} \mid 1 \leq k \leq n, \bar{i} \in O(t_k) \}$. We write $t|_p$ for the *subterm of* t *found at position* $p \in O(t)$, and if $t|_p, u \in T(\Sigma, X)_s$ then $t[u]_p$ denotes the term obtained by replacing the subterm found at p in t by u. We say that $p \in O(t)$ is a *non-variable position* if $t|_p$ is not a variable, and let $\overline{O}(t)$ denote the set of all such non-variable positions.

A *term rewriting rule* is an expression of the form $l \to r$ for $l, r \in T(\Sigma, X)_s$ and $s \in S$ such that *Vars*$(r) \subseteq$ *Vars*(l) and a *term rewriting system* (TRS) R is a set of rewriting rules. If $\sigma_s : X_s \to X_s$ is a family of variable renamings then $\sigma(l) \to \sigma(r)$ is a *variant* of $l \to r$. The *rewrite relation* $\xrightarrow{R}$ associated with a TRS R is a binary relation on terms defined by $t \xrightarrow{R} t'$ if there exists a rule $l \to r \in R$, a position $p \in O(t)$ and a substitution σ such that $t|_p \equiv \sigma(l)$ and $t' \equiv t[\sigma(r)]_p$. We call $t \xrightarrow{R} t'$ a *rewrite step.* We let $\xrightarrow{R^*}$ denote the reflexive transitive closure of $\xrightarrow{R}$. A TRS R is *strongly normalising* if there is no infinite sequence of rewrite steps $t_0 \xrightarrow{R} t_1 \xrightarrow{R} t_2 \xrightarrow{R} \ldots$ and R is *confluent* (or *Church-Rosser*) if for any terms $t, t_1, t_2 \in T(\Sigma, X)_s$ if $t \xrightarrow{R^*} t_1$ and $t \xrightarrow{R^*} t_2$ then there exists $t' \in T(\Sigma, X)_s$ such that $t_1 \xrightarrow{R^*} t'$ and $t_2 \xrightarrow{R^*} t'$. A *complete* TRS is confluent and strongly normalising.

3 Mealy Automata over Abstract Data Types

In this section we formalise a general model of a reactive system as an extended Mealy automaton (EMA) over an abstract data type. We then introduce the syntax and semantics of a linear time temporal logic (LTL) as a language for expressing user requirements on EMA. Finally, we define a syntactic translation of LTL into equations and negated equations, and establish the soundness and completeness of this translation with respect to satisfiability.

We can model a Mealy automaton over an abstract data type as a many-sorted algebraic structure by considering inputs, states and outputs as distinguished data sorts (or types). The input and output types will be typically chosen from some well known data types such as *int*, *string*, *array*, *list* etc. that provide a high level of data abstraction.

Definition 1. *A* **signature** $(S,\ \Sigma,\ input,\ output)$ *for an extended Mealy automaton is a four-tuple, where* $S = \{\ state,\ s_1,\ \ldots,\ s_n\ \}$ *is a sort set,* Σ *is an* S*-sorted signature with distinguished constant and function symbols*

$$q^0 \in \Sigma_{\varepsilon, state},\ \delta \in \Sigma_{state\ input, state},\ \lambda \in \Sigma_{state\ input, output},$$

and $input,\ output \in \{\ s_1,\ \ldots,\ s_n\ \}$ *are distinguished input and output types.*

Definition 2. *Let* $(S,\ \Sigma,\ input,\ output)$ *be a signature for an EMA. An* **extended Mealy automaton** A *(of signature* Σ*) is an* S*-sorted* Σ *algebra* A*.*

As usual q^0_A is the initial state, $\delta_A : A_{state} \times A_{input} \to A_{state}$ is the state transition function, and $\lambda_A : A_{state} \times A_{input} \to A_{output}$ is the output function.

We define the extended state transition and output functions

$$\delta^*_A : A_{state} \times A^*_{input} \to A_{state}, \qquad \lambda^*_A : A_{state} \times A^+_{input} \to A_{output}$$

in the usual way for any $q \in A_{state}$, $\overline{i} \in A_{input}$ and $j \in A_{input}$ by $\delta^*_A(q,\ \varepsilon) = q$ and $\delta^*_A(q,\ \overline{i}\ .\ j) = \delta_A(\ \delta^*_A(q,\ \overline{i}),\ j)$, also $\lambda^*_A(q,\ \overline{i}\ .\ j) = \lambda_A(\ \delta^*_A(q,\ \overline{i}),\ j)$.

If A_{state} is finite then A is termed a *finite state EMA* , otherwise A is termed an *infinite state EMA*.

Next we introduce a linear time temporal logic (LTL) that can be used to express user requirements on EMA. For this it is necessary to integrate the underlying data type signature Σ in an appropriate way. In the sequel we assume that $(S,\ \Sigma,\ input,\ output)$ is a given EMA signature. Let $X = \langle X_s \mid s \in S - \{\ state\ \}\rangle$ be any indexed family of sets X_s of variable symbols of sort s. We assume that $in \in X_{input}$ and $out \in X_{output}$ are two distinguished variable symbols.

Definition 3. *The set* $LTL(\Sigma,\ X)$ *of all* **linear temporal logic formulas** *over* Σ *and* X *is defined to be the smallest set of formulas containing the atomic proposition* **true** *and all equations* $(t = t')$ *for each sort* $s \in S - \{\ state\ \}$ *and all terms* $t,\ t' \in T(\Sigma,\ X)_s$*, which is closed under negation* $\neg$*, conjunction* $\wedge$ *, disjunction* $\vee$ *, and the next* **X***, always future* **G***, sometime future* **F***, always past* $\mathbf{G^{-1}}$*, and sometime past* $\mathbf{F^{-1}}$ *temporal operators.*

As usual, $\mathbf{X}(\phi)$ denotes that ϕ is true in the next time instant, while $\mathbf{G}(\phi)$ (respectively $\mathbf{F}(\phi)$) denotes that ϕ is always (respectively at some time) true in the future of a run. On the other hand $\mathbf{G^{-1}}(\phi)$ (respectively $\mathbf{F^{-1}}(\phi)$) denotes that ϕ was always (respectively at some time) true in the past of a run. While not strictly necessary, including these *past operators* makes this LTL exponentially more succinct, as shown in [12]. This increases the efficiency of our narrowing model checker. We let $(\ \phi \implies \psi\ )$ denote the formula $(\ \neg\phi\ \vee\ \psi\ )$, and $t \neq t'$ denotes $\neg(t = t')$. Then for example, the formula

$$\mathbf{G}(\ (in = x)\ \wedge\ \mathbf{X}(\ (in = y) \implies \mathbf{X}(out = x + y)\ )\ )$$

is an LTL formula that expresses that at all times, if the current input is x and next input is y then in two time steps from now the output will be the sum $x + y$. So in this LTL we can express both control and data properties of reactive systems.

Definition 4. *Let A be an EMA, let $n \in \mathbb{N}$, let $\bar{i} = i_0, i_1, \ldots \in A_{input}^{\omega}$ be an infinite sequence of inputs for A, and let $Val_{A,\alpha} : T(\Sigma, X)_s \rightarrow A_s$ be the valuation mapping on terms given a variable assignment $\alpha : X \rightarrow A$. We define the* **satisfaction relation** *$A, n, \bar{i}, \alpha \models \phi$ for each formula $\phi \in LTL(\Sigma, X)$ by induction.*

(i) $A, n, \bar{i}, \alpha \models$ **true***.*

(ii) $A, n, \bar{i}, \alpha \models t = t'$ if, and only if, $Val_{A,\beta}(t) = Val_{A,\beta}(t')$, where

$$\beta = \alpha[\ in \mapsto i_n,\ out \mapsto \lambda_A(\delta_A^*(q_A^0, i_0, \ldots, i_{n-1}), i_n))\ ].$$

(iii) $A, n, \bar{i}, \alpha \models \neg\phi \ \Leftrightarrow\ A, n, \bar{i}, \alpha \not\models \phi$.

(iv) $A, n, \bar{i}, \alpha \models \phi \wedge \psi$ if, and only if, $A, n, \bar{i}, \alpha \models \phi$ and $A, n, \bar{i}, \alpha \models \psi$.

(v) $A, n, \bar{i}, \alpha \models \phi \vee \psi$ if, and only if, $A, n, \bar{i}, \alpha \models \phi$ or $A, n, \bar{i}, \alpha \models \psi$.

(vi) $A, n, \bar{i}, \alpha \models \mathbf{X}\phi$ if, and only if, $A, n+1, \bar{i}, \alpha \models \phi$.

(vii) $A, n, \bar{i}, \alpha \models \mathbf{G}\phi$ if, and only if, for all $k \geq n$ $A, k, \bar{i}, \alpha \models \phi$.

(viii) $A, n, \bar{i}, \alpha \models \mathbf{F}\phi$ if, and only if, for some $k \geq n$, $A, k, \bar{i}, \alpha \models \phi$.

(ix) $A, n, \bar{i}, \alpha \models \mathbf{G^{-1}}\phi$ if, and only if, for all $k \leq n$ $A, k, \bar{i}, \alpha \models \phi$.

(x) $A, n, \bar{i}, \alpha \models \mathbf{F^{-1}}\phi$ if, and only if, for some $k \leq n$ $A, k, \bar{i}, \alpha \models \phi$.

A formula $\phi \in LTL(\Sigma, X)$ is **satisfiable** *with respect to A if there exists an infinite sequence $\bar{i} \in A_{input}^{\omega}$ and an assignment $\alpha : X \rightarrow A$ such that $A, 0, \bar{i}, \alpha \models \phi$.*

As is well known, for every formula $\phi \in LTL(\Sigma, X)$ there exists a logically equivalent formula $\phi' \in LTL(\Sigma, X)$ in *negation normal form* (NNF) where negations only occur in front of atomic subformulas. To solve LTL formulas by narrowing we translate an NNF formula ϕ into a finite set $S = \{\ S_1, \ldots, S_n\ \}$ of *constraint sets*, where a constraint set S_i consists of equations and negated equations. This translation requires an additional set $\overline{X} = \{\ \overline{x_i} \mid x_i \in X_{input}\ \}$ of fresh variable symbols ranging over input sequence elements.

Definition 5. *Let A be an EMA, and let loopbound be the length of the longest loop-free path in A. For each NNF formula $\phi \in LTL(\Sigma, X)$ we define the* **satisfiability set** *$SatSet_n(\phi)$ as a finite collection of constraint sets by structural induction on ϕ.*

$$SatSet_n(t\ Q\ t') = \{\ \{\ (\theta_n(t)\ Q\ \theta_n(t'))\ \}\ \}$$

where $Q \in \{\ =, \neq\ \}$ and θ_n is the substitution defined by

$$\theta_n = \{\ in \rightarrow \overline{x}_n,\ out \rightarrow \lambda(\delta^*(q^0, \overline{x}_0, \ldots, \overline{x}_{n-1}), \overline{x}_n)\ \}$$

$$SatSet_n(\ \phi\ \wedge\ \psi\ ) = \{\ S_\phi \cup S_\psi\ \mid\ S_\phi \in SatSet_n(\phi),\ S_\psi \in SatSet_n(\phi)\ \}$$

$$SatSet_n(\ \phi\ \vee\ \psi\ ) = SatSet_n(\phi) \cup SatSet_n(\neg\phi\ \wedge\ \psi)$$

$$SatSet_n(\ \mathbf{X}(\phi)\ ) = SatSet_{n+1}(\phi)$$

$$SatSet_n(\ \mathbf{F}(\phi)\ ) = \bigcup_{k=0}^{loopbound} SatSet_{n+k}(\phi)$$

$$SatSet_n(\ \mathbf{F^{-1}}(\phi)\ ) = \bigcup_{k=0}^{loopbound} SatSet_{n-k}(\phi)$$

$$SatSet_n(\ \mathbf{G}(\phi)\ ) =$$

$$\bigcup_{h=0}^{loopbound} \bigcup_{l=1}^{loopbound} \{\ \ \{\ \overline{x}_{n+h+k.l+i} = \overline{x}_{n+h+i}\ \mid\ 1 \le k,\ 0 \le i \le l-1\ \}$$

$$\cup \{\ \delta^*(q^0,\ \overline{x}_0,\ \ldots,\ \overline{x}_{n+h+l-1}) = \delta^*(q^0,\ \overline{x}_0,\ \ldots,\ \overline{x}_{n+h-1})\ \}$$

$$\cup \bigcup_{i=0}^{h+l-1} S_i\ \mid\ S_i \in SatSet_{n+i}(\phi),\ 0 \le i \le h+l-1\quad \}$$

$$SatSet_n(\ \mathbf{G^{-1}}(\phi)\ ) = \{\ \bigcup_{i=0}^{n} S_i\ \mid\ S_i \in SatSet_i(\phi)\ \mathit{for}\ \ 0 \le i \le n\ \}$$

The translation $SatSet_n(\phi)$ preserves solutions of ϕ as follows.

Theorem 1. *Let A be an EMA, and loopbound be the length of the longest loop-free path in A. Let $\phi \in LTL(\Sigma,\ X)$ be in NNF, and let $n \in \mathbb{N}$.*

(i) (Soundness of Translation) For any assignment $\alpha : X \to A$ and input sequence $\overline{i} = i_0,\ i_1,\ \ldots \in A^\omega_{input}$ there exists $S \in SatSet_n(\phi)$ such that

$$A,\ n,\ \overline{i},\ \alpha \models \phi \implies A,\ \beta(\overline{i}),\ \alpha \models S,$$

where the assignment $\beta(\overline{i}) : \overline{X} \to A_{input}$ is given by $\beta(\overline{i})(\overline{x_n}) = i_n$.
(ii) (Completeness of Translation) For any assignments $\alpha : X \to A$ and $\beta : \overline{X} \to A_{input}$ if there exists $S \in SatSet_n(\phi)$ such that $A,\ \beta,\ \alpha \models S$ then there exists an input sequence $\overline{\beta} \in A^\omega_{input}$ such that

$$A,\ n,\ \overline{\beta},\ \alpha \models \phi.$$

Thus by Theorem 1, to solve an NNF formula ϕ it is necessary and sufficient to solve one of the constraint sets $S_1,\ \ldots,\ S_n \in SatSet_0(\phi)$. We will consider a method to solve constraint sets by narrowing in the next section.

4 Model Checking by Narrowing

The problem of finding solutions to a set $\{ \ t_1 = t'_1, \ \ldots, \ t_n = t'_n \ \}$ of equations is the well known unification problem about which much has been written (see e.g. [1]). More generally, in the case that a set $\{ \ t_1 = t'_1, \ \ldots, \ t_n = t'_n, \ u_1 \neq u'_1, \ \ldots, \ u_n \neq u'_n \ \}$ of equations and negated equations must be solved, this problem is known as the disunification problem (see e.g. [4]).

Let Σ be a many-sorted data type signature and E be an equational data type specification having a complete rewrite system R. Then the disunification problem is complicated by the fact that we seek solutions modulo R (and hence E) in the following sense.

Definition 6. *Let R be a term rewriting system. The relation of R-**conversion** denote by $=_R$ is the reflexive symmetric and transitive closure of $\xrightarrow{R}$. Let $S = \{ \ (t_i \ Q_i \ t'_i) \mid i = 1, \ \ldots, \ n, \ n \geq 0, \ Q_i \in \{ \ =, \ \neq \ \} \ \ \}$ be a disunification problem. A substitution $\sigma : X \rightarrow T(\Sigma, \ X)$ is an R-**unifier** of S if for all $1 \leq i \leq n$, if Q_i is $=$ then $\sigma(t_i) =_R \sigma(t'_i)$, and if Q_i is $\neq$ then $\sigma(t_i) \neq_R \sigma(t'_i)$. We let $U_R(S)$ denote the set of all R-unifiers of S.*

In the special case where $E = R = \emptyset$, these problems are known as syntactic unification and syntactic disunification, and both problems are decidable. However in many important cases, both the unification and disunification problems are undecidable. Nevertheless, these problems are semidecidable and one can consider complete search algorithms which always terminate when a solution is to be found. The method of narrowing gives such a complete search algorithm, and can be used whenever the data type specification E can be represented by a complete term rewriting system R.

The basic idea of narrowing is a systematic search of the space of possible solutions using the rules of R. If some equation $t_i = t'_i$ cannot be syntactically unified then we can apply a substitution $\sigma : X \rightarrow T(\Sigma, \ X)$ to t_i (or t'_i) such that the resulting term $\sigma(t_i)$ is not in R normal form and then reduce this in one step. This requires unifying t_i (or t'_i) with the left hand side l of a rule $l \rightarrow r$ in R, and replacing with a suitable instance of r so that a new equation is obtained. A similar process can be applied to negated equations, and can be iterated for all formulas until syntactic unification of the entire set becomes possible, though the narrowing process may not terminate. If it terminates, the resulting sequence of substitutions $\sigma_k : X \rightarrow T(\Sigma, \ X)$ can be composed together with the final syntactic unifier θ to yield an R-unifier.

Definition 7. *We say that a term t is R-**narrowable** into a term t' if there exists a non-variable position $p \in \overline{O}(t)$, a variant $l \rightarrow r$ of a rewrite rule in R and a substitution σ such that:*

(i) σ is a most general syntactic unifier of $t|_p$ and l, and

(ii) $t' \equiv \sigma(t[r]_p)$.

*We write $t \rightsquigarrow_{[p, l \rightarrow r, \sigma]} t'$ or simply $t \rightsquigarrow_{\sigma} t'$. The relation $\rightsquigarrow$ is called R-**narrowing**.*

The R-narrowing relation on terms can be extended to equations and negated equations in an obvious way. A formula $(t\ Q\ t')$ (where Q is $=$ or $\neq$) is R-narrowable into a formula $(u\ Q\ u')$ if there exists a variant $l \to r$ of a rewrite rule in R and a substitution σ such that either $t \rightsquigarrow_{[p, l \to r, \sigma]} u$ for some non-variable occurrence $p \in \overline{O}(t)$ or $t' \rightsquigarrow_{[q, l \to r, \sigma]} u'$ for some non-variable occurrence $q \in \overline{O}(t')$. We write $(t\ Q\ t') \rightsquigarrow_{[p, l \to r, \sigma]} (u\ Q\ u')$ or simply $(t\ Q\ t') \rightsquigarrow_{\sigma} (u\ Q\ u')$. Generalising the R-narrowing relation still further to sets of equations and negated equations we will write $S \rightsquigarrow_{[p, l \to r, \sigma]} S'$ or $S \rightsquigarrow_{\sigma} S'$.

We can relativise the concept of a substitution σ being more general than a substitution τ (c.f. Section 2) to R as follows. Let V be any S-indexed family of sets V_s of variables. We define $\sigma \leq_R \tau[V]$ if for some substitution δ, $\delta \circ \sigma(x) =_R \tau(x)$ for all $s \in S$ and $x \in V_s$. Now we can discuss the soundness and completeness of narrowing.

Theorem 2. *Let $S = \{\ (t_i\ Q_i\ t_i') \mid i = 1, \ldots, n,\ n \geq 0,\ Q_i \in \{\ =, \neq\ \}\ \}$ be a disunification problem.*

(i) (Soundness of Narrowing) Let

$$S \rightsquigarrow_{\sigma_1} S_1 \rightsquigarrow_{\sigma_2}, \ \ldots, \ \rightsquigarrow_{\sigma_n} S_n$$

be a terminated R-narrowing derivation such that S_n is syntactically unifiable by a substitution θ. Then $\theta \circ \sigma_n \circ \ \ldots \ \circ \sigma_1$ is an R-unifier of S.

(ii) (Completeness of Narrowing) If S is R-unifiable then let ρ be any R-unifier and V be a finite set of variables containing $Vars(S)$. There exists a terminated R-narrowing derivation

$$S \rightsquigarrow_{\sigma_1} S_1 \rightsquigarrow_{\sigma_2}, \ \ldots, \ \rightsquigarrow_{\sigma_n} S_n$$

such that S_n is syntactically unifiable. Let μ be a most general syntactic unifier of S_n then $\mu \circ \sigma_n \circ \ \ldots \ \circ \sigma_1 \leq \rho[V]$.

The search space of narrowing is large and narrowing procedures frequently fail to terminate. Many proposals have been made to increase the efficiency of narrowing. One important restriction on the set of occurrences available for narrowing, termed *basic narrowing*, was introduced in [11], and has since been widely studied, e.g. [18].

A basic narrowing derivation is very similar to a narrowing derivation as given in Definition 7 above. However, in a basic narrowing derivation, narrowing is never applied to a subterm introduced by a previous narrowing substitution. This condition is quite complex to define precisely, and the reader is referred to [11].

Theorem 4 of [11] can be used to show that basic narrowing for equations and negated equations is also sound and complete in the sense of Theorem 2. However, for basic narrowing [11] also establishes sufficient conditions to guarantee termination. This property is important in test case generation, where we need to know if a test case exists at all.

Theorem 3. *([11]) Let $R = \{ \ l_i \ \rightarrow \ r_i \ \mid \ i = 1, \ \ldots, \ n \ \}$ be a complete rewrite system such that any basic R-narrowing derivation from any of the r_i's terminates. Then every R-narrowing derivation terminates.*

Many examples of TRS satisfying Theorem 3 are known, including TRS for all finite ADTs. This general termination result can be applied to establish that basic R-narrowing yields a decision procedure for LTL model checking (i.e. basic R-narrowing is sound, complete and terminating) because of the following new result about the CGE symbolic learning algorithm.

Theorem 4. *Let $(R_n^{state}, R_n^{output})$ be the output of the CGE learning algorithm after a sequence of n observations of the I/O behavior of an EMA A. Then $R_n = R_n^{state} \cup R_n^{output}$ is a complete rewrite system and every R_n-narrowing derivation terminates.*

Proof. Proposition 4.5 of [14] establishes that R_n is complete. To establish termination, consider that every rule $l \ \rightarrow \ r \in R_n$ is ground by Definitions 3.12 and 4.4 of [14]. Hence the result is a special instance of Theorem 3 above and Example 3 in [11].

We have constructed an implementation of model checking by basic narrowing. We explain how this is integrated into learning-based testing in Section 5.

5 An LBT Architecture for Testing Reactive Systems

Learning-based testing (LBT) is a general paradigm for black-box specification-based testing that requires three basic components:

(1) a (black-box) *system under test* (SUT) S,

(2) a *formal requirements specification Req* for S, and

(3) a *learned model M* of S.

Given such components, the paradigm provides a *heuristic iterative method* to search for and automatically generate a sequence of test cases. The basic idea is to *incrementally learn* an approximating sequence of models M_i for $i = 1, \ 2, \ \ldots$ of the unknown SUT S by using test cases as queries. During this learning process, we model check each approximation M_i on-the-fly searching for counterexamples to the validity of *Req*. Any such counterexample can be confirmed as a true negative by taking it as the next test case. At step i, if model checking does not produce any counterexamples then to proceed with the iteration, the next test case is constructed by another method, e.g. randomly.

In [16], LBT was applied to testing reactive systems modeled as Boolean Kripke structures. In this paper we consider the case where the SUT S is a reactive system that can be modeled by an EMA over the appropriate abstract data types, and *Req* is an LTL formula over the same data types. Thus we extend the scope of our previous work to deal with both control and data by applying new learning algorithms and model checking technology.

For LBT to be effective at finding errors quickly, it is important to use an incremental learning algorithm. In [16] this was empirically demonstrated by using the IKL incremental learning algorithm for Boolean Kripke structures. However, learning algorithms for finite data types such as IKL do not extend to infinite data types. The CGE learning algorithm of [14] was designed to implement learning EMA over abstract data types. Furthermore, this algorithm is incremental since its output is a sequence of representations $R_1, R_2, \ldots$ of the hypothesis EMA $M_1, M_2, \ldots$ which are the approximations to S. Each representation R_i is a complete TRS that encodes M_i as the corresponding quotient of the prefix tree automaton. Details of this representation can be found in [14]. Furthermore, CGE has many technical advantages over IKL. For example, the number of queries (test cases) between construction of successive approximations R_k and R_{k+1} can be arbitrarily small and even just one query. By contrast, IKL and other table based learning algorithms usually have intervals of tens or hundreds of thousands of queries between successive approximations of large SUTs. As a consequence, model checking can only be infrequently applied.

The input to CGE is a series of pairs $(\overline{i_1}, \overline{o_1}), (\overline{i_2}, \overline{o_2}), \ldots$ consisting of a query string $\overline{i_k}$ for S and the corresponding output string $\overline{o_k}$ from S. In an LBT setting, the query strings $\overline{i_k}$ come from model checker counterexamples and random queries. Finite convergence of the sequence $R_1, R_2, \ldots$ to some TRS R_n can be guaranteed if S is a finite state EMA (see [14]) and the final hypothesis automaton M_n is behaviorally equivalent with S. So with an increasing number of queries, it becomes more likely that model checking will produce a true negative if one exists, as the unknown part of S decreases to nothing. By combining CGE with the narrowing model checker of Section 4, we arrive at a new LBT architecture for reactive systems shown in Figure 1.

Figure 1 illustrates the basic iterative loop of the LBT architecture between: (i) learning, (ii) model checking, (iii) test execution, and (iv) test verdict by an oracle. This iterative loop is terminated by an *equivalence checker*. This component can be used to detect that testing is complete when the SUT is sufficiently small to be completely learned. Obviously testing must be complete by the time we have learned the entire SUT, since model checking by narrowing is solution complete. The equivalence checker compares the current model representation R_k with S for behavioural (rather than structural) equivalence. A positive result from this equivalence test stops all further learning, after one final model check of R_k searches for any residual errors. In practical applications of LBT technology, real world SUTs are usually too large to be completely learned. It is this pragmatic constraint that makes incremental learning algorithms necessary for scalable LBT. In such cases the iterative loop must ultimately be terminated by some other method such as a time constraint or a coverage measure.

Figure 1 shows that the current model R_k is also passed from the CGE algorithm to the basic narrowing model checker, together with a user requirement represented as an LTL formula ϕ. This formula is fixed for a particular testing session. The model checker uses R_k to identify at least one counterexample to ϕ

as an *input sequence* $\overline{i_{k+1}}$ over the underlying input data type. If ϕ is a safety formula then this input sequence will usually be finite

$$\overline{i_{k+1}} = (\ i_1,\ \ldots,\ i_j\ ) \in T(\Sigma)^*_{input}.$$

If ϕ is a liveness formula then the input sequence $\overline{i_{k+1}}$ may be finite or infinite. Since infinite counterexamples to liveness formulas can be represented as infinite strings of the form $\overline{x}\ \overline{y}^{\omega}$, in this case $\overline{i_{k+1}}$ is truncated to a finite initial segment that would normally include at least one execution of the infinite loop $\overline{y}^{\omega}$, such as $\overline{i_{k+1}} = \overline{x}\ \overline{y}$. Observing the failure of infinite test cases is of course impossible, and the LBT architecture implements a compromise solution that executes the truncated input sequence only, and issues a warning rather than a definite test failure.

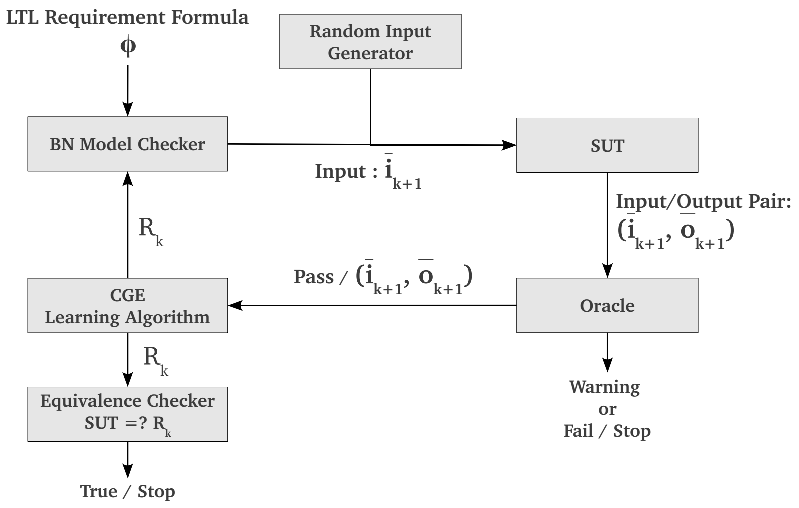


Fig. 1. A Learning-based Testing Architecture for Reactive Systems

If the next test case $\overline{i_{k+1}}$ cannot be constructed by model checking then in order to proceed with iterative testing a *random input string generator* (see Figure 1) is used to generate $\overline{i_{k+1}}$. During this random generation process, any random string that has been used as a previous test case is discarded to avoid redundant replicate tests.

Thus from one of two possible sources (model checking or random generation) a new test case $\overline{i_{k+1}}$ is constructed. Figure 1 shows that this new test case $\overline{i_{k+1}}$ is then executed on the SUT S to yield an *actual output sequence* $\overline{o_{k+1}} = o_1,\ \ldots,\ o_j$. The pair ($\overline{i_{k+1}}$, $\overline{o_{k+1}}$) is then passed to an oracle to compute the $k+1$-th test verdict.

The oracle we have developed for this LBT architecture is more powerful than the one described in [16], and is based on the following two step process.

Step 1. A test verdict can often be derived quickly and simply by computing a *predicted output* $\overline{p_{k+1}} = p_1, \ldots, p_j$ obtained by simulating the behavior of M_k on $\overline{i_{k+1}}$. This is easily derived by applying the TRS R_k to rewrite the input string $\overline{i_{k+1}}$ into its normal form, i.e. $\overline{i_{k+1}} \xrightarrow{R_k}^{*} \overline{p_{k+1}}$. Recall that R_k is a complete TRS, so this normal form is always well defined. We then implement a simple Boolean test $\overline{o_{k+1}} = \overline{p_{k+1}}$. If this equality test returns *true* and the test case $\overline{i_{k+1}}$ was originally a finite test case then we can conclude that the test case $\overline{i_{k+1}}$ is definately *failed*, since the behaviour $\overline{p_{k+1}}$ is by construction a counterexample to the correctness of ϕ. In this case we can decide to stop testing. If the equality test returns *true* and the test case $\overline{i_{k+1}}$ was finitely truncated from an infinite test case (a counterexample to a liveness requirement) then the verdict is weakened to a *warning* (but testing is not stopped). This is because the most we can conclude is that we have not yet seen any difference between the observed behaviour $\overline{o_{k+1}}$ and the incorrect behaviour $\overline{p_{k+1}}$.

Step 2. If the Boolean test $\overline{o_{k+1}} = \overline{p_{k+1}}$ in Step 1 returns *false* then more work is needed to determine a verdict. We must decide whether the observed output $\overline{o_{k+1}}$ is some other counterexample to the correctness of ϕ than $\overline{p_{k+1}}$. This situation easily occurs when the requirement ϕ is a loose specification of the SUT behavior, such as a constraint or value interval. In this case we can evaluate the requirement formula ϕ instantiated by the input and actual output sequences $\overline{i_{k+1}}$ and $\overline{o_{k+1}}$ to determine whether ϕ is true or false. For this we perform a translation similar to $SatSet_0(\phi)$ but with the variables $\overline{x}_n$ and *out* instantiated by the appropriate components of $\overline{i_{k+1}}$ and $\overline{o_{k+1}}$ respectively. We then evaluate all resulting sets of variable free equations and negated equations by rewriting. By Theorem 1, this approach will produce a correct verdict if $\overline{o_{k+1}}$ is a counterexample to ϕ.

Note that while Step 1 was already described in [16], Step 2 is an additional and more powerful step made possible by the translation of LTL into equational logic, and the use of term rewriting to implement the latter.

When the conditions of Theorem 3 are satisfied by the underlying data type then the LBT architecture of Figure 1 can be proven to terminate since the CGE algorithm correctly learns in the limit and basic narrowing is terminating and solution complete. A detailed analysis of this property will be published in an extended version of this paper. The argument is similar to that presented in [16].

6 A Case Study of LBT for Reactive Systems

Since the overhead of model checking an EMA by narrowing is high, it is important to study the performance of our LBT architecture in practice using case studies. Now many communication protocols can be modeled as EMA computing over (freely generated) symbolic data types. Thus the LTL decidability results of Section 4 apply to this class of examples.

The Transmission Control Protocol (TCP) is a widely used transport protocol over the Internet. We present here a performance evaluation of our LBT

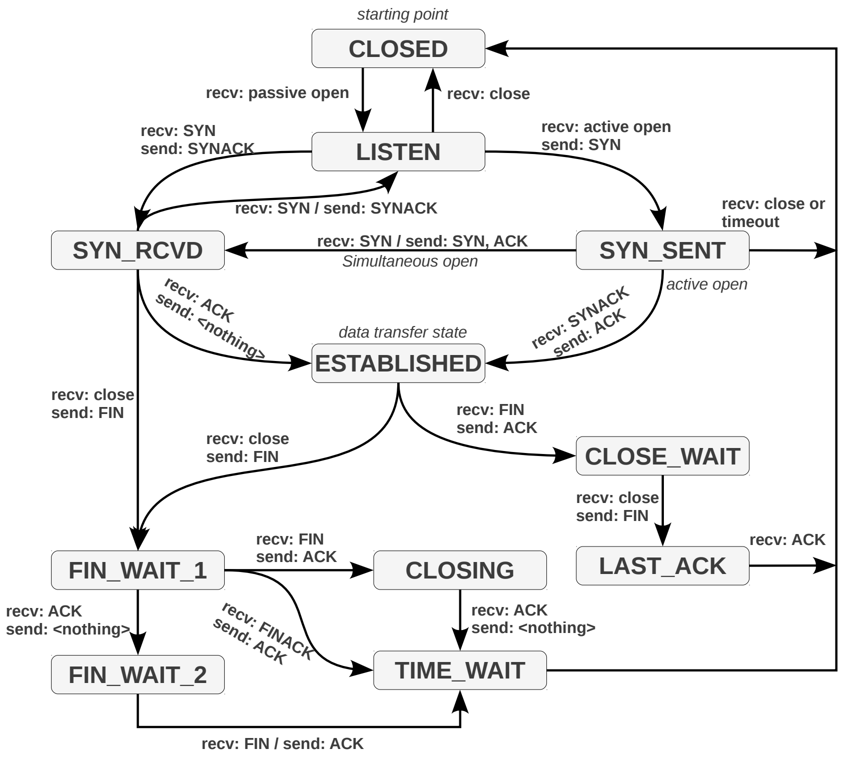


Fig. 2. TCP Mealy Machine Model

architecture applied to testing a simplified model of the TCP/IP protocol as the 11 state EMA shown in Figure 2.

In this performance evaluation, we considered the fault detection capability of LBT compared with random testing. A coverage comparison of learning-based testing with random testing is for example [21], which even considers the same TCP case study. The methodology for comparison was to start from the concrete model of TCP in Figure 2 and consider a variety of correctness requirements as LTL formulas (including use cases). We then injected transition mutations into the SUT which falsified each individual requirement separately. In this way, several different kinds of bugs were introduced into the protocol model, such as mutating the input/output on transitions, adding extraneous transitions or states and so on. Some of these artificial mutations reflect realistic defects that have been discovered in several TCP/IP implementations [10].

Below we informally define five requirements on the TCP/IP protocol and give an LTL formalization of each.

1. Use case. Whenever the entity receives an *active_ open* and sends out a *SYN*, the entity will send out a *SYNACK* if it receives a *SYN*, or send out an *ACK* if it receives a *SYNACK*, and send nothing when receiving other inputs.

$$G(\,(in = active\_open \wedge out = \textsc{syn}) \rightarrow$$

$$X((in = \textsc{syn} \rightarrow out = \textsc{synack}) \wedge (in = \textsc{synack} \rightarrow out = \textsc{ack}))\,)$$

2. Use case. Whenever the entity receives an *active_ open* and sends out a *SYN* and then receives a *SYNACK*, the entity will send out an *ACK* and then will send out an *ACK* if it receives a *FIN*.

$$G(\,(in = active\_open \wedge out = \textsc{syn} \wedge X\, in = \textsc{synack}) \rightarrow$$

$$(X\, out = \textsc{ack} \wedge X^2\,(in = \textsc{fin} \rightarrow out = \textsc{ack}))\,)$$

3. Use case. Whenever the entity performs the IO (*active_ open*, *SYN*) and receives *SYNACK* followed by *FIN* it will send out *ACK* followed by *ACK* and then send out *FIN* if it receives *CLOSE*.

$$G(\,(in = active\_open \wedge out = \textsc{syn} \wedge X\, in = \textsc{synack} \wedge X^2\, in = \textsc{fin}) \rightarrow$$

$$(X\, out = \textsc{ack} \wedge X^2\, out = \textsc{ack} \wedge X^3\,(in = close \rightarrow out = \textsc{fin}))\,)$$

4. Whenever the entity receives a *close* and sends out a *FIN*, or receives a *FIN* and sends out an *ACK*, the entity has either sent a *passive_ open* or received an *active_ open* before, and either sent or received a *SYN* before.

$$G(\,((in = close\ \wedge\ out = \textsc{fin}) \vee (in = \textsc{fin}\ \wedge\ out = \textsc{ack})) \rightarrow$$

$$(F^{-1}\,(in = pass\_open \vee in = active\_open) \wedge F^{-1}\,(in = \textsc{syn} \vee out = \textsc{syn}))\,)$$

5. Whenever the entity performs the IO (*FINACK*, *ACK*) it must have received or sent *SYN* in the past and performed the IO (*close*, *FIN*) in the past.

$$G(\,(in = \textsc{finack}\ \wedge\ out = \textsc{ack}) \rightarrow$$

$$(F^{-1}\,(in = \textsc{syn}\ \vee\ out = \textsc{syn}) \wedge F^{-1}\,(in = close\ \wedge\ out = \textsc{fin}))\,).$$

6.1 Results and Analysis

To compare LBT with random testing on the TCP/IP stack model, we measured two related parameters, namely: (i) the time t_{first} (in seconds), and (ii) the total number of queries (i.e. test cases) Q_{first} needed to *first discover an injected error in the SUT*. To conduct random testing, we simply switched off the CGE and model checker algorithms. The performance of LBT is non-deterministic due to the presence of random queries. Therefore each value of t_{first} and Q_{first} is an average obtained from over 1000 LBT runs using the same injected error.

Table 1. Random testing versus LBT: a performance comparison

Requirement	Random Testing		LBT				
	Q_{first}	$t_{first}(sec)$	Q_{first}	$t_{first}(sec)$	MCQ	RQ	Hyp_size
Req 1	101.4	0.11	19.11	0.07	8.12	10.99	2.2
Req 2	1013.2	1.16	22.41	0.19	9.11	13.3	2.8
Req 3	11334.7	36.7	29.13	0.34	10.3	18.83	3.1
Req 4	582.82	1.54	88.14	2.45	23.1	65.04	3.3
Req 5	712.27	2.12	93.14	3.13	31.8	61.34	4.1

The results of testing the requirements Req1 to Req 5 are listed in Table 1. Note that Q_{first} is the combined sum of the number of model checking queries MCQ and random queries RQ. These are also listed in columns 6 and 7 to provide deeper insight into the strengths and weaknesses of our method. In the final column, Hyp_size is the state space size of the learned hypothesis automaton at time t_{first}. Since Hyp_size is always considerably less than 11 (the state space size of our SUT), this confirms the advantages of using an incremental learning algorithm such as CGE.

We wish to draw two main conclusions from Table 1.

(i) At the level of *logical performance*, (comparing Q_{first} for LBT against Q_{first} for random testing) we see that LBT *always* finds errors with significantly fewer test cases ranging between 0.25% and 18% of the number required by random testing. Therefore, if the overheads of model checking and learning can be reduced then LBT also has the *potential* to outperform random testing in real-time performance.

(ii) At the level of *real-time performance* (comparing t_{first} for LBT against t_{first} for random testing) we see that LBT is *often but not always* significantly faster than random testing, ranging between 0.9% and 160% of the time required by random testing. This reflects the actual real-time overhead of performing both model checking and learning for the SUT and each requirement.

Looking more closely at the results for Reqs 4 and 5, where LBT is somewhat slower than random testing, we can gain deeper insight into these real-time performance issues. For Reqs 4 and 5 both the values MCQ and the ratios RQ/MCQ are significantly higher than for Reqs 1, 2 and 3. In these cases, basic narrowing is performing a large number of constraint solving tasks on unsatisfiable sets of constraints. However, basic narrowing fails very slowly when no solutions can be found. After this, random test cases are applied to proceed with the task of learning the SUT, but these do not necessarily test the actual requirements.

These preliminary results are nevertheless promising, and based on them we make some suggestions for how to further improve narrowing in Section 7. Thus we can improve the overall real-time performance of our current LBT architecture to achieve a real-time performance closer to the logical performance.

It should also be pointed out that as real-time measurement involves factors such as efficiency of implementation, there exists further scope for improvement on the implementation level.

7 Conclusions

In this paper we have shown how a model checker based on narrowing can be combined with a symbolic automaton learning algorithm such as CGE to give a new architecture for black-box specification-based testing using the learning-based testing (LBT) paradigm. We have benchmarked this LBT architecture against random testing, and shown that it compares favorably, with the potential for future improvement.

The results of Section 6.1 suggest that a pure narrowing procedure could be significantly improved by interleaving it with theorem proving techniques to detect unsatisfiability. This is because counterexamples to correctness may be sparse, in which case narrowing fails very slowly. Term rewriting could be applied to this problem too. Furthermore, it is known that basic narrowing modulo theories is incomplete and suggestions such as the *variant narrowing* of [7] could be considered. Finally, we observe that the CGE algorithm does not currently learn infinite state Mealy automata, and this is another extension of our work that must be considered for EMA.

Acknowledgement. We gratefully acknowledge financial support for this research from the Chinese Scholarship Council (CSC), the Swedish Research Council (VR) and the European Union under project HATS FP7-231620. We also acknowledge the helpful comments of the referees in producing the final version of this report.

References

1. Baader, F., Snyder, W.: Unification theory. In: Handbook of Automated Reasoning, pp. 447–531. Elsevier, Amsterdam (2001)
2. Chauhan, P., Clarke, E.M., Kukula, J.H., Sapra, S., Veith, H., Wang, D.: Automated abstraction refinement for model checking large state spaces using sat based conflict analysis. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517. Springer, Heidelberg (2002)
3. Clarke, E., Gupta, A., Kukula, J., Strichman, O.: Sat-based abstraction refinement using ilp and machine learning. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, p. 265. Springer, Heidelberg (2002)
4. Comon, H.: Disunification: a survey. In: Computational Logic: Essays in Honor of Alan Robinson, pp. 322–359. MIT Press, Cambridge (1991)
5. de Moura, L., Bjorner, N.: An efficient smt solver. In: Tools and Algorithms for the Construction and Analysis of Systems, TACAS (2008)
6. Escobar, S., Bevilacqua, V.: Symbolic model checking of infinite-state systems using narrowing. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 153–168. Springer, Heidelberg (2007)

7. Escobar, S., Meseguer, J., Sasse, R.: Variant narrowing and equational unification. In: Proc. WRLA 2008. ENTCS, vol. 238(3), Springer, Heidelberg (2009)
8. Fraser, G., Wotawa, F., Ammann, P.E.: Testing with model checkers: A survey. Tech. rep. 2007-p2-04, TU Graz (2007)
9. Groce, A., Peled, D., Yannakakis, M.: Adaptive model checking. Logic Journal of the IGPL 14(5), 729–744 (2006)
10. Guha, B., Mukherjee, B.: Network security via reverse engineering of tcp code: vulnerability analysis and proposed solutions. IEEE Network 11(4), 40–49 (2007)
11. Hullot, J.M.: Canonical forms and unification. In: Bibel, W. (ed.) CADE 1980. LNCS, vol. 87, pp. 122–128. Springer, Heidelberg (1980)
12. Markey, N.: Temporal logic with past is exponentially more succinct. EATCS Bulletin 79, 122–128 (2003)
13. Meinke, K.: Automated black-box testing of functional correctness using function approximation. In: ISSTA 2004: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 143–153. ACM, New York (2004)
14. Meinke, K.: Cge: A sequential learning algorithm for mealy automata. In: Sempere, J.M., García, P. (eds.) ICGI 2010. LNCS (LNAI), vol. 6339, pp. 148–162. Springer, Heidelberg (2010)
15. Meinke, K., Niu, F.: A learning-based approach to unit testing of numerical software. In: Petrenko, A., Simão, A., Maldonado, J.C. (eds.) ICTSS 2010. LNCS, vol. 6435, pp. 221–235. Springer, Heidelberg (2010)
16. Meinke, K., Sindhu, M.: Incremental learning-based testing for reactive systems. In: Gogolla, M., Wolff, B. (eds.) TAP 2011. LNCS, vol. 6706, pp. 134–151. Springer, Heidelberg (2011)
17. Meinke, K., Tucker, J.V.: Universal algebra. In: Handbook of Logic in Computer Science, vol. 1, pp. 189–411. Oxford University Press, Oxford (1993)
18. Middeldorp, A., Hamoen, E.: Completeness results for basic narrowing. Applicable Algebra in Engineering, Communication and Computing 5(3-4), 213–253 (1994)
19. Peled, D., Vardi, M.Y., Yannakakis, M.: Black-box checking. In: Formal Methods for Protocol Engineering and Distributed Systems FORTE/PSTV, pp. 225–240. Kluwer, Dordrecht (1999)
20. Raffelt, H., Steffen, B., Margaria, T.: Dynamic testing via automata learning. In: Yorav, K. (ed.) HVC 2007. LNCS, vol. 4899, pp. 136–152. Springer, Heidelberg (2008)
21. Walkinshaw, N., Bogdanov, K., Derrick, J., Paris, J.: Increasing functional coverage by inductive testing: a case study. In: Petrenko, A., Simão, A., Maldonado, J.C. (eds.) ICTSS 2010. LNCS, vol. 6435, pp. 126–141. Springer, Heidelberg (2010)

Monitoring Dynamical Signals While Testing Timed Aspects of a System

Goran Frehse[1], Kim G. Larsen[2], Marius Mikučionis[2], and Brian Nielsen[2]

[1] Verimag (UJF-CNRS-INPG), 2, av. de Vignate, 38610 Gieres, France
goran.frehse@imag.fr
[2] Department of Computer Science, Aalborg University,
Selma Lagerlöfs Vej 300, 9220 Aalborg Øst
{kgl,marius,bnielsen}@cs.aau.dk

Abstract. We propose to combine timed automata and linear hybrid automata model checkers for formal testing and monitoring of embedded systems with a hybrid behavior, i.e., where the correctness of the system depends on discrete as well as continuous dynamics. System level testing is considered, where requirements capture abstract behavior and often include non-determinism due to parallelism, internal counters and subtle state of physical materials. The goal is achieved by integrating the tools Uppaal [2] and PHAVer [3], where the discrete and hard real-time aspects are driven and checked by Uppaal TRON and strict inclusion of dynamical trajectories is verified by PHAVer. We present the framework, the underlying theory, and our techniques for integrating the tools. We demonstrate the applicability on an industrial case study.

1 Introduction

Timed automata (TA) is a convenient and expressive modelling language for expressing state- and time-dependent discrete behavior such as time constraints on event occurrences. In particular the Uppaal-language has proven useful and expressive in a large number of case studies. The editing, simulation and analysis of Uppaal-TA models is supported by the Uppaal model-checking tool. Similarly, (online) model-based testing is implemented in the Uppaal TRON tool [7].

However, TA cannot directly capture and describe continous behavior, which is normally abstracted away. When this cannot be done, a workaround may be to model discrete approximations; these may however be cumbersome, inaccurate and significantly degrade the performance of the analysis.

In contrast, (linear) hybrid automata ((L)HA) allows continuous evolutions (trajectories) to be described directly through (linear) differential equations associated with the locations of the automata. PHAVer [3] is a model-checker which provides exploration and analysis capabilities for a rich class of hybrid automata through incrementally refined over-approximation of the trajectories. However, for purely timed and sophisticated discrete behavior its performance cannot compete with timed automata tools like UPPAAL. Furthermore, its language contains none of the advanced feature of UPPAAL (different types of communication channels, committed locations, C-like instructions on transitions, etc.), which are highly useful for effective modelling of control software.

Testing of hybrid systems is not new, e.g. Reactis Validator is based on a hardware-in-the-loop simulation, where the requirements are expressed as assertions on a Simulink model [9]. Osch provides a formal framework [11] and a prototype tool based on TorX [10] architecture and hybrid χ simulator [12]. Both frameworks are based on a simulation, thus only a deterministic behavior with some relative deviation is allowed. Our framework is unique in a sense that it allows imprecision in timed and dynamical behavior (clock drifts, measurement imprecision, limited state observability, parallelism, abstract requirements) by means of non-determinism in a specification model, i.e. the user can explicitly specify ambiguity of any aspect: bounds on timing of events, dynamical variable values, their derivatives, internal/unobservable transitions. This approach is consistent in treating the imprecision in both time and continuous signals, so the ideas from UPPAAL TRON testing framework can be carried to PHAVER with just a few modifications. Consequently, the conformance check is as easy as inequality check of observed outputs with symbolic state bounds, the test is sound (if the test fails then IUT definitely does not conform to specification), but not necessarily exhaustive (some faults may escape detection due to discrete sampling and measurement imprecision).

Our goal is therefore to combine the UPPAAL TRON and PHAVER tools to construct a tool environment that supports effective and efficient testing and monitoring of control systems that contains both significant timed actions and continuous trajectories.

Model-based *monitoring* is a passive observation of a (black-box) system executing in its operating environment, and checking whether the observed behavior is permitted by (conforms-to) the specified behavior in the model. *Model-based testing* also uses the model to generate test input sequences to stimulate the system, and replaces (part of) the system's environment. Using *online testing* the system is stimulated and evaluated interactively and in real-time. Our goal is to extend the framework of online testing of real-time systems presented in [7] to hybrid systems. UPPAAL TRON simultaneously stimulates and evaluates the system behavior, and can be configured to perform either or both tasks, e.g., work solely as a monitor, while PHAVER provides sound over-approximation of continuous dynamics.

Fig. 1 shows a simple system setup where a plant is controlled by an embedded system. The controller is a digital device running an embedded program which inputs (samples) the sensor values and outputs actuations to the plant.

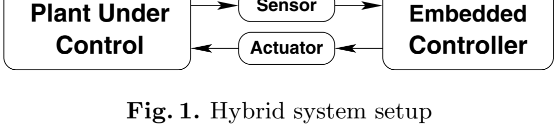


Fig. 1. Hybrid system setup

The proposed framework will typically be used to monitor and test the behavior of the controller, but can in principle equally well be used to evaluate whether the assumed behavior captured and represented by models of the plant are correct, as only a black-box is assumed.

Fig. 2 shows an arrangement of tools from [6] we use in our test setup: the emulator is a test generator that to the IUT plays the role of an environment (plant under control), the monitor is a test oracle that monitors the observable input/output interaction and decides whether the behavior of IUT is conforming.

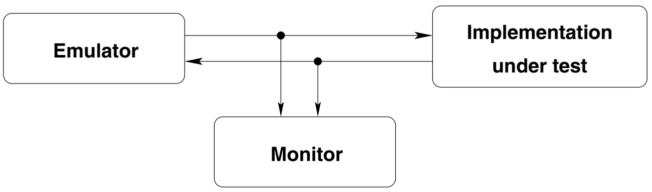


Fig. 2. Test setup

The main solution idea is to run the two tools in parallel, i.e., Uppaal TRON for test generation (environment emulation) and monitoring discrete and timed behavior and PHAVer monitoring the continuous behavior. Thus each tool evaluates a part of the overall model.

For this to work, the two tools must be synchronized with respect to real-time and observed events. Moreover, as the behavior in most complex models depends on both aspects, the sub-models also need to exchange information to trigger behavior changes. For instance, when the timed automaton executes an action or switches location the HA model may need to evaluate different trajectories. Similarly, the values (or crossing of thresholds) may be of importance to the possible actions of the TA.

Our contributions are as follows: monitoring dynamical behavior against nondeterministic models, a modeling pattern to keep Uppaal TA models in synchrony with LHA so that timed models would focus on discrete and timed aspects and hybrid automata mostly on dynamics, a test adapter framework that allows the tools to exchange synchronization events thus enabling the models to "communicate" during online testing, demonstration how our technique and tool can be applied to an industrial case study [8].

Requirement Models. The interaction of discrete events and continuous, time-driven dynamics can be efficiently modeled by a so-called *hybrid automaton* [1]. A hybrid automaton $H = (Loc, Var, Lab, Inv, Flow, Trans, Init)$ consists of a graph in which each vertex $l \in Loc$, also called *location* or *mode*, is associated via $Flow(l)$ with a set of differential equations (or inclusions) that defines the time-driven evolution of the continuous variables. A *state* $s \in Loc \times \mathbb{R}^{Var}$ consists of a location and values for all the continuous variables Var. The edges of the graph, also called *discrete transitions* $Trans$, allow the system to jump between locations, thus changing the dynamics, and instantaneously modify the values of continuous variables according to a *jump relation* μ. The jumps may only take place when the values of the variables are within the domain of μ. The system may only remain in a location l as long as the variable values are in a range called *invariant* $Inv(l)$ associated with the location. All behavior originates from one *initial state* $Init$.

An *execution* or *trajectory* of the automaton is a sequence of discrete jumps and pieces of continuous trajectories according to its dynamics, and originates from the initial state. *Linear Hybrid Automata* are hybrid automata in which invariants, guards and initial states are given by linear constraints over the variables, jump relations are given by linear constraints over the variables before and after the jump, and the flow constraints are linear constraints over the derivatives only (must not depend on the state variables). Fundamental properties such as reachability are undecidable for (linear) hybrid automata in general. Tools like PHAVer use polyhedra for symbolic state computations, and the operators used in reachability are of exponential complexity in the number of continuous variables.

A *Timed Automaton* is a hybrid automaton with the following restrictions:

- The continuous variables are *clocks*, i.e., their time-derivative is equal to one.
- Invariants and guards are conjunctions of constraints on a single clock, i.e., of the form $\bigwedge_i x_i \bowtie c_i$, where the x_i are clock variables, $\bowtie \in \{\leq, <, =, >, \geq\}$ and the c_i are integer constants.
- The jump relations are *resets*, i.e., each transition may set a subset of the variables to an integer value.

Timed automata have the advantage over general hybrid automata that fundamental properties such as reachability are decidable, and efficient (polynomial complexity) operators are known for symbolic state computation.

Conformance. The formal characterization of correctness is a natural extension of rtioco conformance relation from [7]. We refer to [12] for formal details about hybrid conformance and impact of discrete sampling.

Definition 1 specifies the relation between IUT p and a system specification s represented by states $\langle e, p\rangle$ and $\langle e, s\rangle$ which are composed of IUT model state s and environment model state e.

Definition 1. Relativized timed input/output conformance relation [7]. *$p, s \in \mathcal{S}$ and $e \in \mathcal{E}$ are input-output compatible:*

$$p \ \mathsf{rtioco}_e \ s \overset{def}{=} \forall \sigma \in \mathsf{TTr}(e).\mathsf{Out}\big(\langle e, p\rangle \ \mathsf{after} \ \sigma\big) \subseteq \mathsf{Out}\big(\langle e, s\rangle \ \mathsf{after} \ \sigma\big) \qquad (1)$$

where $\mathsf{TTr}(e)$ *is a set of timed traces generated by* e*, operator* after *denotes reachable states after a trace is executed,* $\mathsf{Out}(T)$ *denotes a set of possible outputs from a set of states* $T \subseteq \mathcal{E} \times \mathcal{S}$*.*

Intuitively the definition says that in order to establish the conformance between IUT state p and specification state s we have to do the following: generate a trace σ from the environment specification e, execute the trace σ on both IUT composed with environment ($\langle p, e\rangle$) and a complete system specification ($\langle s, e\rangle$), and then check whether the output produced by IUT is included in the outputs of specification. Explicit treatment of the environment model e allows to test the IUT in relation to specific environment.

For hybrid monitoring we adapt the hioco relation proposed by [11]:

Definition 2. Hybrid input/output conformance relation [12]. *Let $p, s \in \mathcal{S}$ be input-output compatible implementation and specification respectively, environment $e \in \mathcal{E}$, then:*

$$p \ \mathsf{hioco}_e \ s \overset{def}{=} \forall \sigma \in \mathit{traces}(\langle s, e\rangle) \ . \ \mathsf{Out}\big(\langle p, e\rangle \ \mathsf{after} \ \sigma\big) \subseteq \mathsf{Out}\big(\langle s, e\rangle \ \mathsf{after} \ \sigma\big) \wedge \tag{2}$$

$$\mathsf{traj}(\langle p, e\rangle \ \mathsf{after} \ \sigma) \subseteq \mathsf{traj}(\langle s, e\rangle \ \mathsf{after} \ \sigma) \tag{3}$$

The definitions means that in order to establish the conformance relation, one must consider all specification traces (discrete inputs and outputs intermingled with continuous time trajectories) and check that resulting responses from implementation (discrete outputs and trajectories) are included in the specification. The first part of definition (2) can be checked by UPPAAL TRON as it is concerned with discrete I/O and is compatible with rtioco where trajectories are replaced with time delays, and PHAVER is used to monitor the second part (3). We check (monitor) only the output trajectories, while the original hioco from [11] has additional operators for composing continuous input and output trajectories.

Online Test Algorithm. Algorithm 1 is an abstract version of the timed online testing algorithm which generates sound and exhaustive tests [7]. The algorithm is based on maintaining a current estimate of system state S and consists of three parts: generating an input action based on S, performing a time delay, checking that the output is consistent with the current state estimate, and potentially restarting the test.

2 Monitoring Trajectories

The monitoring of continuous dynamics is implemented by supplementing the action and delay cases of Algorithm 1 with PHAVER symbolic computations in a parallel process, where the continuous output signals are discretized by periodic sampling and checked that the values are included in a model behavior.

We formally define the operators included in PHAVER for monitoring and testing and discuss their implementation for linear hybrid automata. Let s, s' be hybrid states, i.e., each a pair of a discrete location and a valuation of the continuous variables. We consider the set of labels (actions) *Lab* of the automaton to consist of two disjoint sets of *observable actions* Σ_O and *unobservable actions* Σ_U.

We write $s \xrightarrow{a} s'$ if there is a discrete transition with label a that leads from s to s'. We write $s \xrightarrow{\delta} s'$ for $\delta \in \mathbb{R}^{\geq 0}$ if there is a continuous trajectory leading from

Algorithm 1. Test generation and execution, $OnlineTest(\mathcal{S}, \mathcal{E}, \mathsf{IUT}, T)$.

```
$S := \{\langle s_0, e_0\rangle\}$;                    // let the set contain an initial state
while $S \neq \varnothing \wedge \sharp iterations \leq T$ do
    switch Random$(\{action, delay, restart\})$ do
        case $action$                                  // offer an input
            if EnvOutput$(S) \neq \varnothing$ then
                randomly choose $i \in$ EnvOutput$(S)$;
                send $i$ to IUT, $S := S$ after $i$;
        case $delay$                                   // wait for an output
            randomly choose $d \in$ Delays$(S)$;
            sleep for $d$ time units or wake up on output $o$ at $d' \leq d$;
            if $o$ occurs then
                $S := S$ after $d'$;
                if $o \notin$ ImpOutput$(S)$ then return $fail$;
                else $S := S$ after $o$
            else $S := S$ after $d$;                   // no output within $d$ delay
        case $restart$                                 // reset and restart
            $S := \{\langle s_0, e_0\rangle\}$;
            reset IUT
if $S = \varnothing$ then return $fail$ else return $pass$
```

state s to s' without taking any discrete transitions in between. The notation extends to sets of states S, S' in a straightforward manner.

Let $Post_c(S) = \{s' \mid \exists s \in S, \delta \in \mathbb{R}^{\geq 0} : s \xrightarrow{\delta} s'\}$. Let $Post_d(S, a) = \{s' \mid \exists s \in S : s \xrightarrow{a} s'\}$. For a given alphabet $\Sigma \subseteq Lab$, let $Reach_\Sigma(S)$ be the smallest fixed point of the sequence $P_0 = Post_c(S)$, $P_{k+1} = P_k \cup \bigcup_{a \in \Sigma} Post_c(Post_d(P_k, a))$.

Some of the operators required for testing make explicit use of the time at which events occur. We extend the above operators to explicitly include time by modifying the hybrid automaton: We include a clock variable κ, which has derivative zero and does not change its value during transitions. For the sake of simplicity, we shall abuse notation a little and write (s, κ) for a state of the extended automaton where κ denotes the added clock variable. We annotate the operators on the extended automaton by adding the superscript κ, e.g., $Reach^\kappa_\Sigma(S, \kappa_0)$ denotes the reachable sets of states by taking only discrete transitions with a label in Σ, and starting from an extended state (s, κ) where $s \in S$ and $\kappa = \kappa_0$. We have implemented the following operators:

- $\mathsf{delayTop}(S, \Sigma) = max_\kappa Reach^\kappa_{Lab \setminus \Sigma}(S, 0)$ computes an upper bound on the time the automaton can execute without taking a transition with a label in Σ.
- $\mathsf{delayBottom}(S, \Sigma) = min_\kappa Reach^\kappa_{Lab \setminus \Sigma}(S, 0)$ computes a lower bound on the time the automaton can execute without taking a transition with a label in Σ.

– $\mathsf{transition}(a, x = c) = Post_d(S, a) \cap \{s \mid s(x) = c\}$ computes the states the automaton can be in after taking a transition with label a and having the variable x have the constant value c.
– $\mathsf{observation}(x = c) = S \cap \{s \mid s(x) = c\}$ computes the states the automaton can be in if the variable x has the constant value c.

We will now briefly discuss the complexity of the above operations as they are implemented in PHAVer. Recall that PHAVer represents sets of continuous states as sets of convex polyhedra. The complexity of the above *Post* operators is exponential in the number of continuous variables.

The above operators delayTop and delayBottom make use of the automaton extended with an additional clock. This increases the number of continuous variables in the system by one. Since the complexity of the post-operators grows exponentially with the number of variables, this might significantly affect performance. Once the *Reach* set of the extended system is computed, the upper and lower bounds on κ are found efficiently using linear programming.

If the hybrid automaton has bounded invariants and affine dynamics of the form $\dot{x} = Ax + b$, with A being a matrix and b a vector of rational coefficients, PHAVer overapproximates the behavior by transforming it into a LHA. The transformation splits each location into a number of equivalent locations whose invariants cover the original invariant, and where each invariant is smaller than a given size. For each of the ingoing and outgoing transitions of the original location, equivalent copies are added to the generated locations. In each of the generated locations l_i, the derivatives are overapproximated with the set $\dot{x} \in \{x' \mid \exists x \in Inv(l_i) : x' = Ax + b\}$, which brings the automaton to LHA form. By splitting all locations into parts that are small enough, an overapproximation of arbitrary accuracy can be achieved, albeit for the price of generating a potentially very large number of locations and transitions.

3 Modelling Pattern

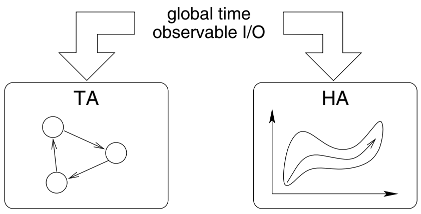
Our solution consists of running two tools in parallel: Uppaal TRON for test generation and monitoring discrete and timed behavior and PHAVer monitoring continuous behavior.

We set up both tools to keep track of the state estimate by using symbolic operations on the corresponding requirement models and declare failure if the state estimate of the IUT becomes empty, signaling that the observed behavior is outside the specification.

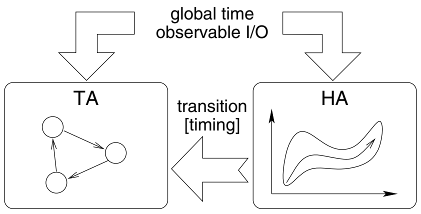
We propose to split the dynamic and timed/discrete aspects into two models:

– A timed automata model responsible for discrete and timely behavior.
– A hybrid automata model handling the dynamic features and using as few discrete aspects as possible.

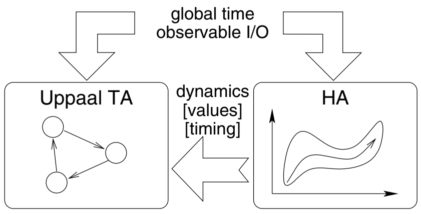
The reason for the separation is that the model-checking tools are optimised to analyze such aspects on their own. Moreover, performance wise, it is cheaper to



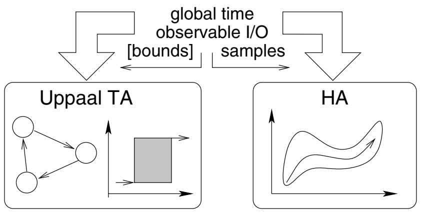
(a) Time and I/O.



(b) Time, I/O and discrete feedback.



(c) Time, I/O and dynamic values.



(d) Time, I/O and direct bounds.

Fig. 3. Various ways to synchronize TA and HA model state estimates

keep the two components separately in parallel than operate on a flat product of them which can be exponentially large.

Depending on the concrete system setup there are various possibilities on how to keep the models in synchrony as shown in Figure 3:

Time and I/O. Timed and dynamic aspects may be independent of each other (e.g. two parallel processes without interaction in between), and hence no special handling is needed. Fig. 3a shows that timed automata (TA) model state estimate is calculated independently from hybrid automata (HA) model but using the same input/output sequence synchronised on global time reference.

Event synchronization. When an important state change occurs in the HA model it is communicated via a dedicated internal event to TA model. Fig. 3b shows the same TA and HA model setup synchronised on the same timed sequence of input/output events, but in addition it is possible to transfer the information when certain discrete transitions (mode changes) in the HA can and should occur, thus effectively TA and HA models are synchronised on discrete transitions.

Data synchronization. In case of Uppaal TA models we may handle the dynamic variable values from HA model state estimate as discrete integer data. Fig. 3c shows that parts of HA dynamics is passed to TA model state estimation as discrete data values. However, a special care must be taken to ensure that the data is handled consistently between models.

Over-approximation. Finally, one can use methods from [5] to produce abstractions of hybrid automata in a form of stop-watch timed automata (Uppaal TRON can handle those too), thus we can use the resulting abstract model to handle timed and discrete aspects and use the hybrid model to monitor the continuous behavior more precisely. In a similar fashion we propose to model each continuous variable v from HA by two integer bounds v_{low} and v_{upper} in the UPPAAL TA model and update the bound variables. Fig. 3d shows that both models are synchronised on continuous signals: the HA model state estimate is updated with concrete sample values (measurements from the IUT), while UPPAAL TA is updated by bounds which approximate the sample values. In order to use this feature, the UPPAAL TA model needs to be adjusted to handle two boundary values for each continuous signal non-deterministically instead of a single concrete value. As a result, UPPAAL TA handles not just the timed automata aspects but also dynamical behavior in an abstract way: the continuous signal is represented by a rectangular region which is adjusted and synchronised with the rest of the model at runtime.

The over-approximation approach requires only a few changes in the adapter protocol but is general enough to support the other above techniques, thus we focus only on over-approximation. Note that the abstract bounds need not to be updated continuously with each sample if the new sample signal is still within bounds.

4 Architecture

The tools are connected using the UPPAAL TRON test adapter framework. The adapter is a software layer that takes care of translating and delivering input/output actions between tester and an implementation under test (IUT), thus the major parts of it is IUT-specific. In addition, every input/output action is translated into a script and fed into the PHAVER model-checker for monitoring dynamical signals. Fig. 4 shows the setup of tools:

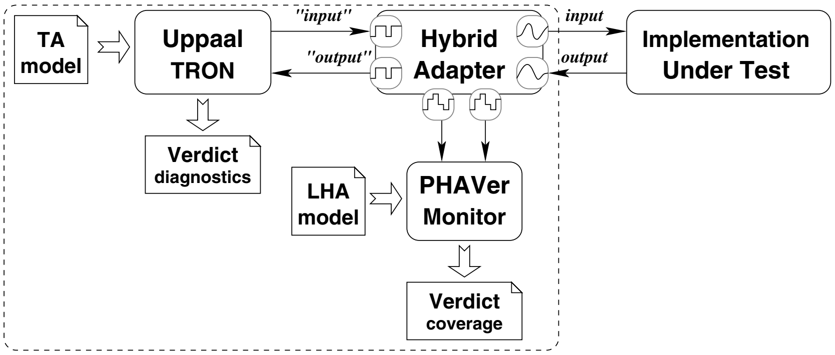


Fig. 4. Framework for online testing of hybrid systems

– Uppaal TRON reads a TA model, interacts by abstract representation of discrete input/output action sequences and delivers a verdict at the end of test. Some diagnostics are provided in case the test fails.
– A Hybrid Adapter translates the abstract discrete input and performs the actual input, continuously observes concrete output and translates it into abstract output. Both input and output are reported further to PHAVer in the form of PHAVer script which contain digitised timed input/output sequence.
– PHAVer reads a linear hybrid automata (LHA) model, computes the reachable set of states according to the script and produces a plot of reachable symbolic states until the state set becomes empty and the test consequently fails.

In Fig. 4 all entities in rounded rectangles contain a separate thread and communicates asynchronously so that input/output delivery would not block the test progress.

Fig. 5 shows an example of the events happening during delivery of input and processing of output. The Uppaal TRON and Hybrid Adapter time-stamp events independently of each other and PHAVer is just processing a stream of commands containing already time-stamped input/output events. Each input/output event is time-stamped by a time interval [from; till], where from is the earliest possible moment in time an action has happened, and till is the latest[1].

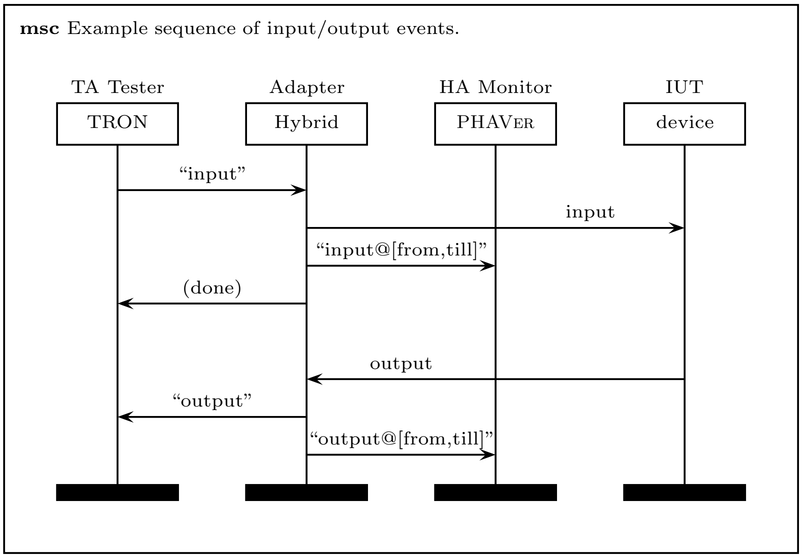


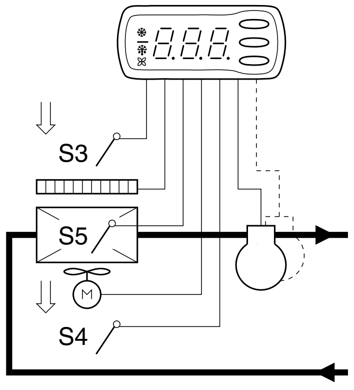
Fig. 5. Message sequence chart of input/output handling

[1] Inputs are time-stamped before and after sending an event and it is also possible to use communication latency when estimating the time-stamp of output.

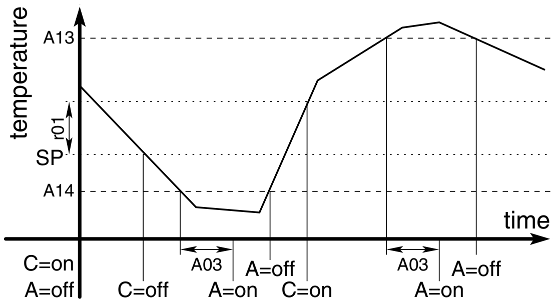
5 Application

We use electronic cooling controller EKC204A provided by Danfoss for case study [8] as a hybrid implementation which contain discrete, timely and dynamical aspects.

Temperature Controller. Fig. 6a shows how Danfoss EKC204A temperature controller can be deployed at industrial refrigeration systems. The device can be



(a) Sensor and actuator setup



(b) Temperature regulation principles

Fig. 6. Danfoss EKC204A temperature controller deployment

applied in many sensor and actuator configurations, in particular the controller supports the following connections shown in Figure 6a: sensor S3 for outside room air temperature, a relay for controlling defrost heater (electric or gas burner), sensor S5 for temperature on evaporator transferring the heat from air to evaporation of cooling liquid, a relay for fan motor to ensure air rotation around evaporator and/or entire room, sensor S4 for inside room air temperature, actuator relay for compressor used to push cooling liquid in the loop from evaporator to condenser and back. The sensors provide fixed precision number reading and actuators are binary relays having states of "on" and "off".

The controller can be configured to specific strategy using a register database. Fig. 6b shows the main principles behind temperature controller:

- The temperature can vary from $-50°C$ to $+50°C$. The device reads the temperature sensor data and calculates the displayed temperature.
- The objective is to control the temperature between SP and $SP + r01$.
- The compressor (relay C) is used to cool down the air temperature by turning it on ($C = on$) whenever temperature is higher than $SP + r01$ and turning it off ($C = off$) whenever temperature is below SP. Compressor has a requirement to stay on or off for at least some time in order minimise wear and tear of hardware.

– The controller should ring a temperature alarm ($A = on$)whenever the temperature is higher than register $A13$ or lower than register $A14$ for more than $A03$ amount of time and turn it off ($A = \mathit{off}$) when the temperature is within acceptable bounds $[A13; A14]$.

For our purposes we use only inside room air temperature sensor (S4). The controller assumes that the temperature sensors are not perfect (readings usually fluctuate), hence PID-like temperature algorithm is applied to filter and stabilize the room temperature estimate. The temperature is estimated and displayed using equation $T_n = \frac{4 \cdot T_{n-1} + T_s}{5}$, where T_s is a temperature reported by sensor and T_i is temperature estimate at moment i. The temperature is recalculated about once per each second and may vary slightly (the software is soft real-time). The filter is applied only if the temperature change is within $1°C$ and is adjusted immediately otherwise. All other aspects (compressor, alarm and defrost control) depend on the calculated temperature rather than raw sensor values, thus even TA model needs to be aware of this temperature calculation.

Model Setup. The requirements are modelled by a combination of UPPAAL timed automata and PHAVER hybrid automata.

[8] and later work resulted in an almost complete model of EKC204A aspects. In this paper we describe only the relevant part—temperature estimation—which is described by hybrid automata.[2] The UPPAAL model consists of the following processes modelling various aspects:

– The test is part of environment which controls how the temperature should be (e.g. decreased until it is below $-7°C$ and then increased again).
– The tempGen generates a sequence of concrete temperatures that are sent as raw temperature values injected into sensors.
– tempMonitor read the temperature sensor values and provides the calculated temperature values to the rest of the TA instances.
– The lowTempAlarm monitors the calculated temperature and triggers an alarm if the temperature is below the threshold for longer than allowed time bound.

The hybrid model consists of just two hybrid automata: TempMonitor monitors the calculated temperature values and TempRoom monitors whether the sensed temperature is changing reasonably (e.g. not too rapidly). The latter is optional, but it demonstrates that we can monitor the behavior of the tester as well as IUT.

Fig. 7 provides an overview on how the two models are synchronised together with IUT:

– $temp\_t(ENVTemp)$ is a signal generated from tempGen to tempMonitor which results in concrete action $inject\_temp(T)$ sent to the IUT and a $set\_temp(room\_temp)$ synchronization in hybrid model between TempRoom and TempMonitor.

[2] The implementation of the adapter for PHAVER, a generated script instance and requirement models can be downloaded at
`http://www.cs.aau.dk/$^\sim$marius/phaver`

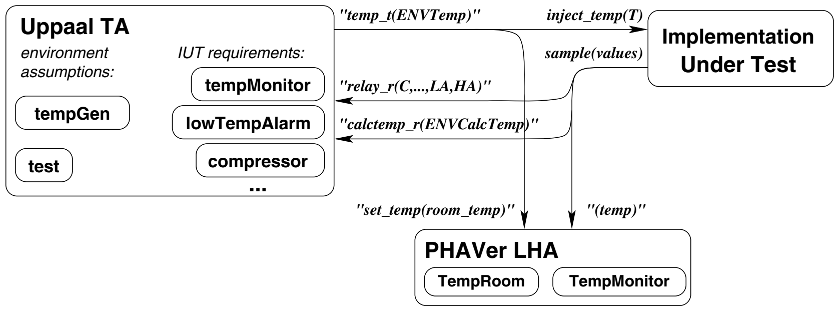


Fig. 7. Uppaal and PHAVer model setup for temperature controller

– $sample(values)$ is an output from IUT which is translated into relay changes ($relay\_r(C, F, D, A, HA, LA)$) to the Uppaal model and a calculated temperature snapshots $calctemp\_r(ENVCalcTemp)$ to the Uppaal model and ($temp$) update in the PHAVer model.

The Abstract Timed Model. In Uppaal model the tempMonitor is responsible for estimating and displaying the temperature value on the screen as well as notifying all other processes. The calculated temperature is estimated by $CalcTL$ and $CalcTU$ variables denoting lower and upper bounds.

Fig. 8 shows the temperature estimation abstraction when a new temperature value is injected into the sensors and the resulting Uppaal timed automaton which computes the abstract temperature bounds.

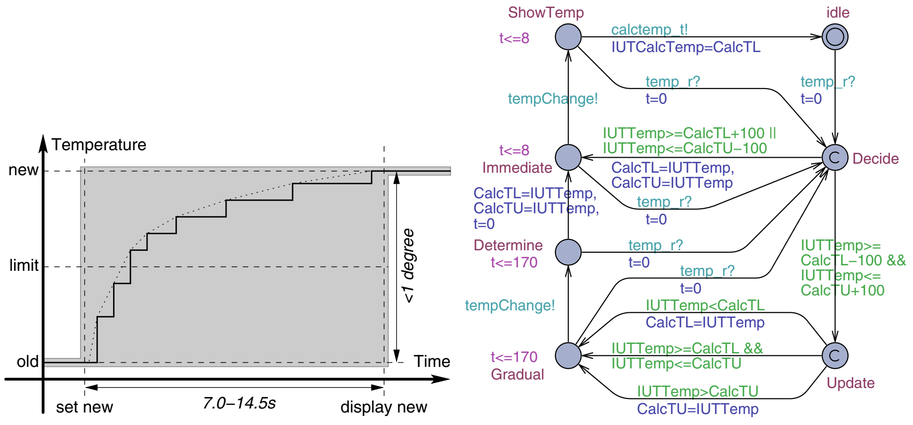


Fig. 8. Temperature estimation and its abstract model as tempMonitor process

LowTempAlarm is modelled as a parallel process (Fig. 9) interpreting the calculated temperature bounds $[CalcTL; CalcTU]$ using non-deterministic edges with guard comparisons to $LowTempLimit$.

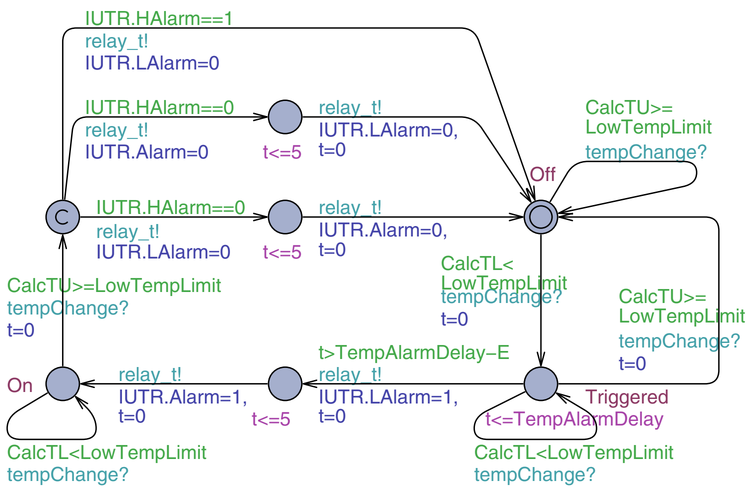


Fig. 9. Requirement model for low temperature alarm

Hybrid Model. Fig. 10 shows a hybrid automaton for temperature calculation which has four locations: *idle* – temperature is constant, *decide* – there is temperature change and controller instantaneously decides whether it is going up or down, *adjustUp* – the temperature is rising, *adjustDown* – the temperature is dropping. The current temperature estimate is kept in variable *temp* and the new sensor value is changed with *set_temp* event in variable *target*.

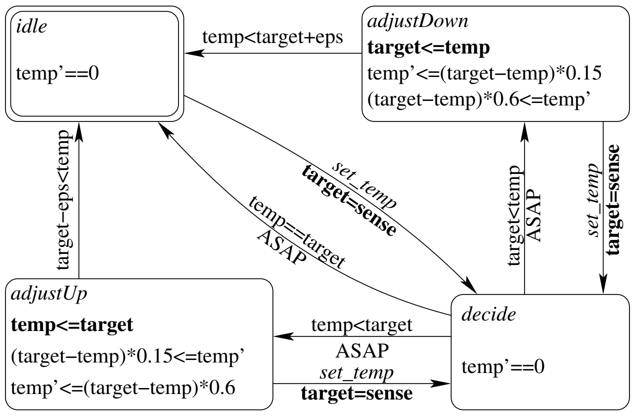


Fig. 10. Model of a controller temperature sensing and estimation

Results. Fig. 11 shows the temperature region computed from the model by an over-approximation for each instance of time during testing:

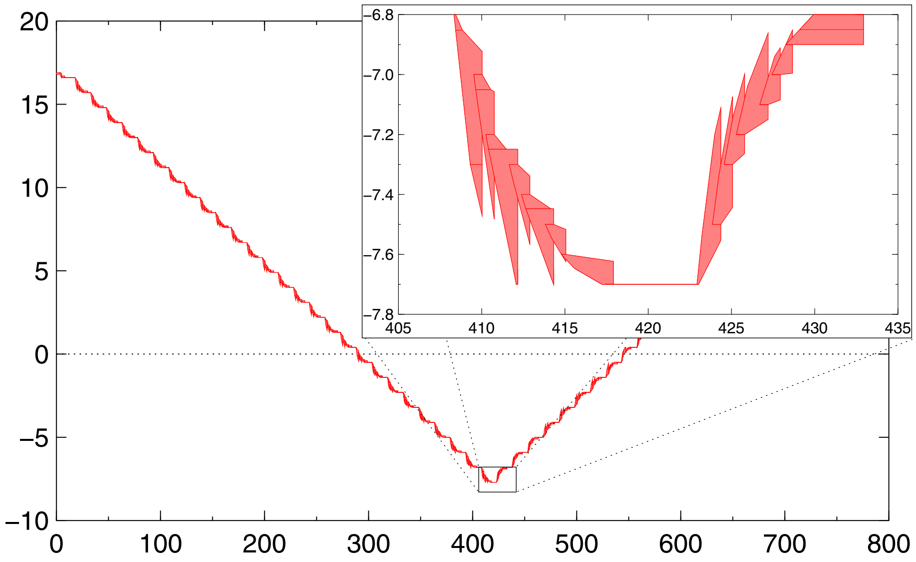


Fig. 11. Symbolic state evolution in PHAVer from test trace monitoring: time in seconds on horizontal axis, temperature in $°C$ on vertical axis

– Calculated temperature estimation starts from a single (known) point at $16.8°C$.
– Uppaal TRON then generates a new air temperature at $16.6°C$ and feeds it to IUT.
– PHAVer estimates that the calculated temp should fall withing first region.
– HybridAdapter samples a new calculated temperature reading and reports it to PHAVerAdapter, but does not report it yet to Uppaal TRON.
– PHAVer finds the observed temperature point (an interval in time) in the estimated region, updates it and recomputes future temperature region (second region).
– The process continues in this way until the calculated temperature reaches the injected temperature value (within 15s) where both Uppaal TRON and PHAVer are notified.
– The calculated temperature estimate collapses to one point in the PHAVer plot, until a new temperature injection is dictated by Uppaal TRON.

6 Conclusions

We have shown how timed automata and hybrid automata model-checkers can be combined to achieve online testing and monitoring of embedded software controller at different degrees of precision: timed/discrete and sampled continuous signals.

In the current setting, PHAVER can only function as a monitor, because it lags behind UPPAAL TRON (due to the high computation cost of computing state set updates). However, ongoing work on improvements on the PHAVER engine, and we are optimistic that it will be capable of also functioning as trajectory stimulator. Alternatively a simulator tool can be used to generate dynamical stimuli.

In the future we will implement the operators used in monitoring using reachability algorithms that are based on time-discretization and bounded time horizon, see [4].

References

1. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.-H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138(1), 3–34 (1995)
2. Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In: Alur, R., Sontag, E.D., Henzinger, T.A. (eds.) HS 1995. LNCS, vol. 1066, pp. 232–243. Springer, Heidelberg (1996)
3. Frehse, G.: Phaver: Algorithmic verification of hybrid systems past hytech. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 258–273. Springer, Heidelberg (2005)
4. Frehse, G., Ray, R.: Design principles for an extendable verification tool for hybrid systems. In: ADHS 2009: 3rd IFAC Conference on Analysis and Design of Hybrid Systems (2009)
5. Henzinger, T.A., Ho, P.-H.: Algorithmic analysis of nonlinear hybrid systems. In: Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, pp. 225–238. Springer, Heidelberg (1995)
6. Hessel, A., Larsen, K., Mikucionis, M., Nielsen, B., Pettersson, P., Skou, A.: Testing real-time systems using uppaal. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST. LNCS, vol. 4949, pp. 77–117. Springer, Heidelberg (2008)
7. Larsen, K.G., Mikucionis, M., Nielsen, B.: Online testing of real-time systems using uppaal. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 79–94. Springer, Heidelberg (2005)
8. Larsen, K.G., Mikucionis, M., Nielsen, B., Skou, A.: Testing real-time embedded software using uppaal-tron: an industrial case study. In: EMSOFT 2005: Proceedings of the 5th ACM International Conference on Embedded Software, pp. 299–306. ACM, New York (2005)
9. Sims, S., DuVarney, D.C.: Experience Report: The Reactis Validation Tool. In: Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming - ICFP 2007, vol. 42, p. 137. ACM Press, New York (2007)
10. Tretmans, J., Belinfante, A.: Automatic testing with formal methods. In: EuroSTAR 1999: 7th European Int. Conference on Software Testing, Analysis & Review, Barcelona, Spain, November 8-12 (1999); EuroStar Conferences, Galway, Ireland
11. van Osch, M.: Hybrid input-output conformance and test generation. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) FATES 2006 and RV 2006. LNCS, vol. 4262, pp. 70–84. Springer, Heidelberg (2006)
12. van Osch, M.: Automated Model-based Testing of Hybrid Systems. PhD thesis, Technische Universiteit Eindhoven (2009)

Model-Based Testing of Industrial Transformational Systems

Petur Olsen[1,⋆], Johan Foederer[2], and Jan Tretmans[3,4,⋆⋆]

[1] Department of Computer Science,
Centre for Embedded Software Systems,
Aalborg University,
Aalborg, Denmark
petur@cs.aau.dk

[2] Test Automation,
Océ-Technologies B.V.,
Venlo, The Netherlands
johan.foederer@oce.com

[3] Model-Based System Development,
Radboud University,
Nijmegen, The Netherlands
tretmans@cs.ru.nl

[4] Embedded Systems Institute,
Eindhoven, The Netherlands

Abstract. We present an approach for modeling and testing transformational systems in an industrial context. The systems are modeled as a set of boolean formulas. Each formula is called a clause and is an expression for an expected output value. To manage complexities of the models, we employ a modeling trick for handling dependencies, by using some output values from the system under test to verify other output values. To avoid circular dependencies, the clauses are arranged in a hierarchy, where each clause depends on the outputs of its children. This modeling trick enables us to model and test complex systems, using relatively simple models. Pairwise testing is used for test case generation. This manages the number of test cases for complex systems. The approach is developed based on a case study for testing printer controllers in professional printers at Océ. The model-based testing approach results in increased maintainability and gives better understanding of test cases and their produced output. Using pairwise testing resulted in measurable coverage, with a test set smaller than the manually created test set. To illustrate the applicability of the approach, we show how the approach can be used to model and test parts of a controller for ventilation in livestock stables.

⋆ Work performed while visiting Radboud University and Océ-Technologies B.V.

⋆⋆ This work has been supported by the EU FP7 under grant number ICT-214755: Quasimodo.

1 Introduction

Océ is a leading company in designing and producing professional printers. As the complexity of these printers grows, both due to features added and due to the requirement to support several input formats and backwards compatibility, the task of effectively testing the printer controller becomes very difficult. In this paper we present a model-based approach to improve the testing of the controller software of Océ printers.

We consider the part of the controller which processes input job descriptions and sends commands to the hardware. The system considered is in its abstract form a simple function. It takes a set of parameter values as input and computes a set of output parameter values. Input parameters are specific settings to a print job (number or pages, duplex/simplex, etc.), and the output is the description, in terms of output parameters, of the actually printed job. The dependencies between inputs and outputs are not trivial, and as the number of input parameters is over 100 and the number of output parameters is 45, the size of the system makes testing a difficult task.

The controller is modeled using a set of constraint clauses on the input parameter values, in the form of boolean formulas. Each clause relates a set of input values to an expected output value. The approach that we take in this paper is similar to that of QuickCheck [4] and Gast [9], both of which are automatic testing tools for functional programming languages, that generate random test cases. Both tools are less suited for use at Océ, since, being based on functional programming languages, they are cumbersome to integrate into existing test frameworks, whereas randomness makes structured generation of test cases and coverage determination more challenging. This led us to implement an internal prototype in Python to handle the testing.

Others have tried similar approaches to testing real world applications, such as Lozano et al. [11], who model a financial trading system with constraint systems. This leads us to believe that the approach is applicable to other types of systems as well. To further evaluate this approach we analyze how it can be used to model parts of the controller software for a ventilation system for livestock stables.

While the final goal is to detect faults in the SUT, this has not been the main focus in this project. Rather the focus has been to take steps toward creating a maintainable, large-scale model-based testing environment. It was not our aim to compare numbers of bugs found in model-based and manual testing.

This paper presents the problem of testing printer controller software. We present an approach to modeling the controller as a set of boolean formulas, including a modeling trick to enable us to make relatively simple models for the complex system. We present the testing process and how the desired coverage can be achieved. Additionally we present some discussions on using model-based testing in an industrial setting, and which benefits this approach has given Océ. Finally, to illustrate the applicability of our approach, we show how it can be adopted to model and test part of the controller software for a ventilation system for livestock stables.

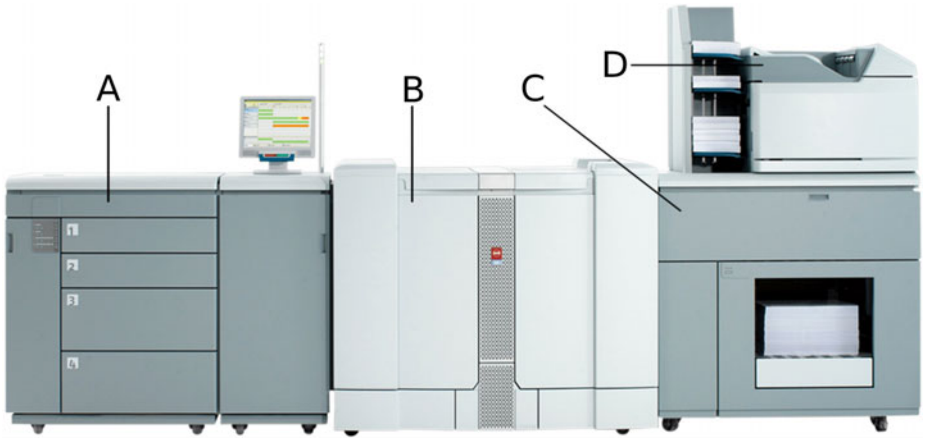


Fig. 1. VarioPrint 6250. A) Input module with four trays. B) Printer module. C) High capacity stacker output location. D) Finisher output location.

2 Problem Description

The problem is to test the controller of Océ printers. Océ produces professional printers, an example of which is shown in Figure 1.

– A is the input module with four input trays.
– B is the actual printer module.
– C is an output location called the High Capacity Stacker (HCS).
– D is an output location which supports stapling, called the Finisher.

This example is a small configuration of a printer. Several input modules can be attached and different output locations with different finishing options are supported.

The controller in these printers basically has two tasks: (*i*) handling the printing queues, and (*ii*) processing an input job description and sending the corresponding commands to the printing hardware. The part of the controller handling the printing queues can be seen as a reactive system, which continually monitors for job descriptions, sends them through the job processor to the printing hardware, and allows the user to perform actions on a user interface, for instance to cancel a job. This part can be modeled using some form of state machine. The part handling job processing, however, does not operate reactively. It accepts one job description at a time, and produces output for that job. Such a system can be seen in its abstract form as a simple, stateless function, accepting a set of input parameter values and returning a set of output parameter values. This is the part of the controller which this project focuses on, and which will be tested.

A *job description* consists of two parts: a document in a Printer Description Language (PDL) format and an optional *ticket* describing how to print the document. Several PDL and ticket formats are supported, each supporting different features and using different formats for expressing features. Some features are supported in both the PDL and the ticket, requiring the job processor to handle contradictions. Example input parameters include output location, stapling, and punching.

As output the job processor presents a set of parameter values for each sheet to be printed. These values are sent to the printer hardware which prints the job as specified. Example output parameters also include output location, stapling and punching, however the relationship between the inputs and outputs is not as simple as it might seem.

First, there are the contradictions. Stapling, for instance, can be specified both in the PDL and in the ticket, in which case the ticket will overrule the PDL. While stapling is enabled, an output location can be selected which does not support stapling, in which case the output location is overruled to one which does support stapling. However, there might not be any output locations attached to the printer which support stapling, in which case the stapling is disabled and the output location is as specified. Just for these two simple parameters we already have a lot of cases.

In addition to contradictions, there are different formats for specifying values. Specifying stapling in a ticket, for instance, has ten possible values: None, Top, TopLeft, Left, ..., and Saddle. The stapling output from the job processor only has five: None, Portrait, Landscape, Booklet, and Saddle (the orientation of the paper and the output location determine where the specific staple is located). Similarly other PDLs and ticket formats might have different formats for specifying stapling. Translating between these formats is not trivial.

On top of all these are the settings of the job processor. For instance the limit for the number of pages which the printer can staple can vary. Several other settings are available in the job processor.

It is clear that the job processor needs to handle all peculiarities in the input, as well as any settings of the printer. The job processor needs to support all configurations of printers, and needs to be able to print any job on any configuration, albeit possibly with some functionalities disabled. The configuration and settings of the printer can be seen as inputs to the job processor. Adding these to the PDL and ticket, and looking at all available parameters, the number of parameters for the job processor comes to well over 100. These facts make the job processor a very complex system, and testing such a system is not trivial.

2.1 Testing at Océ

The current testing process at Océ involves running a job description on a simulation of the hardware. When running the job processor on the simulated hardware, the output is presented in the form of a so-called APV file. This APV file contains all parameters for each printed sheet. Currently there are 45 parameters in the APV file.

The resulting APV file is analyzed manually. If deemed correct it is saved as a reference for future test runs. In subsequent automatic test runs the output can be checked against the reference and the result of the test can be determined.

There are several issues with this testing process. The first is maintainability when updating the job processor to support more parameters. This requires all test cases to be updated to support this parameter. Secondly, changing some requirements, which lead to failing test cases, requires the new APV file to be manually analyzed again. This manual analysis is very time consuming and error prone. It occurs that errors survive through the development process because of faulty analysis of the APV file.

The execution time of the test cases is also becoming an issue. Nightly runs, executing the complete set of test cases, have to finish in the morning, to present the results to the engineers. At the current number of test cases some of these runs do not complete in time. Due to expansions and new developments the number of test cases is expected to double, in the near future. This poses big requirements to the computer farm running the nightly tests, and requires expensive expansions. Therefore it is desirable to reduce the number of test cases, but the quality of the complete test set must not suffer.

Currently a *test case* is a Python script which sets up the printer, generates one or more job descriptions, and sends them to the controller in a specified order. These test cases are designed by test engineers who know the system intimately. The test cases are designed to find likely errors, and are very specifically designed, such that a failing test case gives some hint to where the error occurs. For instance, a test case might focus on stapling by generating several job descriptions with different stapling positions. If this test case fails the error is most likely in the stapling module. This gives very specific test cases and to get good coverage it requires a lot of test cases. This leads to the desire to have structurally generated test cases which have some measure of coverage, while minimizing the number of test cases.

In the current framework there is no uniform way of defining test cases. This stems from the different formats for PDLs and tickets, and the fact that current test cases are directly coded at a low level in Python. Since these are often generated in batches in for-loops, it can be difficult for other testers and developers to understand exactly what a test case does. This leads to problems with understanding test cases, and once a test case fails, it can also be troublesome to understand precisely what the parameters of the failing job were.

This presents four areas where Océ wants to improve their testing process:

- maintainability,
- execution time,
- coverage, and
- understanding of test cases.

We will improve on these aspects, by implementing a model-based approach to testing.

3 Modeling the Controller

The job processor is in its abstract form a simple, stateless function. It takes a number of input parameter values and computes a number of output parameter values. We modeled the job processor as a collection of Boolean formulas, where each formula specifies the value for one output parameter through an implication with on the left-hand side a conjunction of input parameter constraints, and on the right-hand side an output parameter constraint:

$$i_1 = v_1 \wedge i_2 = v_2 \wedge \cdots \wedge i_n = v_n \Rightarrow u_p = v_p \tag{1}$$

This formula expresses that the expected output of parameter u_p is value v_p if input parameters i_j have values v_j for $1 \leq j \leq n$, respectively. Each of these formulas is called a *clause* in the model.

For integer parameters we also allow comparisons like $i_j \leq v_j$, e.g., to refer to equivalence classes of input parameters. As an example, a simplified clause of the staple position could look like this:

$$\begin{aligned}(Staple = TopLeft \wedge SheetCount \leq 100) \Rightarrow \\ StaplePos = Portrait\end{aligned} \tag{2}$$

This specifies that the output parameter for Staple Position *StaplePos* has value *Portrait* if the input parameter *Staple* is *TopLeft* and there are less than 101 sheets.

For integer output parameters we allow the expected output to be calculated by a function on the input parameters. For instance if the *Plexity* is set to *Duplex* (printing on both sides of the paper), *SheetCount* becomes half of the number of printed pages: $SheetCount = \lceil Pages/2 \rceil$.

The actual job processor model has many more parameters and also more possible values for the parameters, resulting in many more clauses. A complete job processor model consisting of such a collection of clauses must first be verified for completeness and consistency, i.e., checked whether the collection indeed specifies a function from input parameters to output parameters, but such a verification is orthogonal to testing.

Satisfaction of the model has been used as oracle for our testing process. This means that the model is instantiated with actual output parameter values of the job processor implementation, together with the corresponding input parameter values. (Section 4 will deal with choosing input values). If all clauses hold the test passes; if a clause does not hold then the test fails and the output parameter specified in the false clause is wrong.

3.1 Dependencies

As seen above the model for staple position depends on two input values, and the complete model is even bigger. If we have a look at a simplified clause of the output location *OutputLoc*:

$$\begin{aligned}(TicketOutputLoc = HCS \wedge Staple = TopLeft \wedge \\ SheetCount \leq 100) \Rightarrow \\ OutputLoc = Finisher\end{aligned} \tag{3}$$

then we can see that it depends on the input parameters *Staple* and *SheetCount*. This is because the output location should only be overridden if a staple was requested, and the printer is actually able to staple. This causes a chain of dependencies, where the output location clause must contain all – transitive – dependencies in its clause. These chains clutter the clauses and make modeling cumbersome, since there are a lot of these type of dependencies. To simplify the clauses we can observe that a part of (3) can be substituted with (2). Substituting $(Staple = TopLeft \wedge SheetCount \leq 100)$ for $StaplePos = Portrait$ we get the simpler clause:

$$(TicketOutputLoc = HCS \wedge StaplePos = Portrait) \Rightarrow \\ OutputLoc = Finisher$$

We can see that the output location actually depends on the output parameter *StaplePos*. Formally, we allow $(i_j = v_j)$ from Equation 1 to refer to input- and output parameters.

One potential problem arises with this approach. If there are circular dependencies, we can not trust the results. To avoid circular dependencies we arrange all clauses in a hierarchy, where the leaves have no dependencies and parents depend on the parameters of their children. As long as this hierarchy is kept, it is safe to use some output parameters to verify other output parameter values. This approach simplifies the clauses significantly, and enables us to model these complex systems.

4 Testing

Testing a job processor implementation involves three steps:

- Selecting input values,
- executing the SUT with the input values, and
- verifying output from the SUT using the model.

Executing the SUT is done using the existing framework for automatic testing at Océ. Verifying the outputs is done using the model, as explained in the previous section. To select input values we look at the complete set of input parameters supported by the model, and the domains of these parameters. Instantiating each parameter constitutes a single test case. This can be done randomly, to generate a set of test cases, or it can be done structurally, based on some coverage criterion.

It has been shown in several projects [5, 6, 8, 2, 12] that most software errors occur at the interaction of a few factors, i.e. most errors are triggered by particular values for only a few input parameters, whereas the error is independent from the values of the other input parameter. Some projects report up to 70% of bugs found with two or fewer factors and 90% with three or fewer [5, 10], others show up to 97% of bugs found with only two factors [12]. This, combined with the fact that the number of test cases needs to be minimized, leads to combinatorial testing techniques, such as pairwise (or more generally n-wise) testing. The number of test cases in n-wise testing for fixed n grows logarithmically compared to exponential growth for testing all possible combinations.

The coverage of output parameters is not guaranteed by using the combinatorial testing technique. Pairwise testing does, however, tend to cover most of the unintended uses of the system, and many of the paths which lead to special cases, whereas manually generated test cases tend to focus on normal operation of the system. Once a test suite has been created, the output coverage can be analyzed, and the test suite can be updated to add any required coverage.

Test cases are generated based on a set of input parameters and their domains. A test case is an assignment of each input parameter to a single value from its domain. A *test specification* is a set of relations between input parameter name and the discrete domain of that parameter:

$$\begin{gathered}(P^1, \{v_1^1, v_2^1, \ldots, v_{n_1}^1\})\\(P^2, \{v_1^2, v_2^2, \ldots, v_{n_2}^2\})\\\vdots\\(P^m, \{v_1^m, v_2^m, \ldots, v_{n_m}^m\}).\end{gathered}$$

Given such a test specification, algorithms can generate a set of test cases which cover all pairs of values. That is, for every v_k^i and v_l^j where $i \neq j$, $P^i = v_k^i$ and $P^j = v_l^j$ for at least one test case.

For instance if we have three input parameters: *TicketOutputLoc*, *Staple*, and *SheetCount*, the test specification could be:

$$\begin{gathered}(\mathit{TicketOutputLoc}, \{\mathit{Finisher}, \mathit{HCS}\})\\(\mathit{Staple}, \{\mathit{None}, \mathit{Top}, \mathit{Left}\})\\(\mathit{SheetCount}, \{\leq 100, 101\})\end{gathered}$$

Each pairwise combination of values, e.g. $\mathit{TicketOutputLoc} = \mathit{Finisher}$ and $\mathit{Staple} = \mathit{Left}$, must be present in at least one test case. In this case six test cases are needed. Generating complete coverage would require 12 test cases.

4.1 Diagnosis

The automatic test case generation based on a job specification can be used as a tool in diagnosis. Reducing the domain of one or more parameter in the job specification, can provide information about the fault. As an example consider the job specification above, and consider a fault has been found with one of the generated test cases. Reducing the domain of *Staple* to $\{\mathit{None}\}$, and re-running test case generation and test case execution, can help locate the fault in the SUT. If, for instance, none of these new test cases finds the fault, it tells us that the fault only occurs when a staple is requested. This tells us that the error is either in the staple module, or occurs at the interaction between the staple module and some other part of the controller. Using this approach the test engineers can easily generate new sets of test cases to help locate bugs in the SUT.

5 Implementing the Test Tool

Other projects to improve the testing process, have previously been carried out at Océ. Experiences from these projects have shown that integrating tooling with existing frameworks can be difficult and cumbersome, and introducing new tools and frameworks requires some learning effort from the engineers. This led us to implement a prototype tool for this project by hand. The existing framework for automated testing has a lot of tooling and libraries for testing the job processor, therefore it was decided to integrate the prototype into this framework. The existing framework is written in Python, so Python is the language of choice.

The clauses are implemented as if-then-else statements. To give a logical grouping of clauses, all clauses pertaining to the same output parameter are grouped into the same Python class. This way a class is said to *check* an output parameter by implementing all clauses for that parameter. To ease implementation several output parameters can be checked by the same class.

The entire set of actual input values and output values is passed to each class. This enables the class to access any values needed in the clauses to verify their respective output value. The classes access the values they need and then go through a series of if-then-else statements that implement the clauses in the formal approach. Once an expected value is found, it is checked against the actual output value, and an appropriate response is added to the return set.

The return set contains *OK* or *Error* responses from each class, and is returned back to the test system. Here it can be analyzed and all errors found by the model, can be returned to the engineer.

Once an output value has been verified, it is removed from the set of output values, which is passed to subsequent classes. This feature has two effects. First, it enables us to check whether all output values have been verified, by examining if the set is empty when all classes have been invoked. Second, it enforces the hierarchy required to detect circular dependencies as explained in Section 3.1. This is enforced since a circular dependency would result in a missing value in the latter class in the circle. This also means that classes with dependencies need to be invoked first, and classes with no dependencies are invoked last.

5.1 Test Case Generation

N-wise testing has currently been implemented using the tool `jenny`[1]. `jenny` is an executable which accepts as input, the domain size of each input parameter and generates a set witnessing n-wise coverage of all input parameter values. Currently `jenny` is always executed for pairwise coverage. This could be extended, if the coverage requirements increase. A wrapper has been written around `jenny`. A test specification is passed to the wrapper, which generates a `jenny` query for the domain sizes of the parameters. `jenny` returns a set of test cases which are translated back into the specific values in the test specification. As an example consider the test specification:

[1] http://burtleburtle.net/bob/math/jenny.html

$$(TicketOutputLoc, \{Finisher, HCS\})$$
$$(Staple, \{None, Top, Left\})$$
$$(SheetCount, \{\leq 100, 101\})$$

The wrapper generates a query for `jenny` with three parameters with domain size two, three, and two respectively. `jenny` generates six test cases:

```
1a 2a 3b
1b 2c 3a
1b 2b 3b
1a 2b 3a
1b 2a 3a
1a 2c 3b
```

Each line represents a test case. The number represents the input parameter. The letter represents the value of the parameter. The test case `1b 2c 3a` is translated into $(TicketOutputLoc = HCS, Staple = Left, SheetCount = \leq 100)$.

This way test cases are generated based on the test specification as created by the test engineer. In the case of diagnosis the engineer can reduce the domain sizes in the test specification, and rerun the wrapper to get a new set of test cases.

5.2 Run Time

The time required to execute a single test case is highly dependent on the number of pages printed in that test case; depending on the hardware running the simulator, printing a single page can take up to half a second. This fact means that test cases with a lower number of pages are preferred. In the model for testing the stapling module the equivalence classes for the number of sheets are:

- 1 (too few sheets),
- 2 (too few sheets for duplex),
- 3 – 100,
- >= 101 (too many sheets).

Including all these in the pairwise test case generation would include many jobs with 100 pages or more. However, it can be observed that if the staple limit works for a single test case, it most likely works for all test cases. This observation leads us to implement the possibility to add manually generated jobs to the test set. By adding two jobs with 100 and 101 sheets printed, we reduce the equivalence classes for staple limit to three and by choosing a low value for the equivalence class 3 − 100 the number of sheets printed can be kept low in all other jobs. This dramatically reduces the time required to execute the test suite, while still keeping acceptable coverage.

5.3 Invalid Test Cases

Since the PDL and ticket formats support different features, it is possible to generate invalid test cases. For instance, one ticket format supports accounting options, so we need to have input parameters for accounting. However, if we generate a test case with accounting activated, while using a ticket format which does not support accounting, the test tool is unable to generate the job for the controller, since activating accounting in this ticket format is not possible. This is an invalid test case, since it can not be executed on the SUT.

These invalid combinations need to be removed from the pairwise coverage. This is because the pairs covered by an invalid test case are not executed on the system. Excluding simple combinations of parameter values is supported in the current version. More complex exclusions, such as employed by e.g. AETG [5], ATGT [3], or Godzilla [7] might be required in the future.

6 Status and Discussion

It was chosen to focus on modeling the stapling capabilities of the printers, as an initial step. The currently implemented models support four parameters: PDL format, page count, output location, and staple location. The four parameters have domain sizes two, three, three, and eight respectively. This set is pairwise covered in 27 test cases. To cover the upper page limits for stapling four test cases are manually added, bringing the total number of test cases generated from this project to 31.

It is difficult to say precisely how this coverage compares to the current set of test cases, as a lot of test cases touch stapling, while testing other areas of the controller as well. Of the manually generated test cases a total of 170 test cases use stapling. Looking at all the parameters and parameter values used in these 170 test cases, we observe 11 parameters with domains ranging from two to nine. Getting pairwise coverage of all these input parameters can be achieved with only 86 test cases. These parameters are not currently supported by the model, so the test cases can not be executed at this time. This shows us that once the models are extended, the number of test cases can be reduced. Determining if the fewer number of test cases will locate as many or more errors will require further work.

The choice of implementing a prototype in Python proved very useful. This also supports previous experience in similar projects. Integrating with the existing Océ framework was very easy. In the current version of the tool, implementing the models as Python classes is cumbersome and requires some copy-paste. With further development and refactoring, we expect to build a viable environment for developing models.

Using a model-based approach gives the engineers far better understanding of the test cases and their outcome. The new framework gives better overview of which parameter values are selected in each test case, and the models can be used to give a hint as to where an error is located in the code. The controllable test case generation can also be used for analysis, to find the precise interactions between parameters which cause the error.

The issues with maintenance are also improved, as making changes to the requirements only requires updating the model, then all test cases should work. Updating and implementing the models is not trivial, and errors in the model could cause false positives and false negatives. However, since the same models are used by all test cases, it is more likely that errors in the model will be found.

Currently only pairwise coverage is supported. As the usage of this approach grows in Océ, it will be seen how effective this coverage is in locating faults. It might be the case that the coverage needs to be supplemented by manually generated test cases, or replaced by a different coverage measure. Currently no analysis of output coverage is done. This requires engineers to manually examine the test cases, and possibly supplement with additional cases.

Even though the focus in this project was not to detect faults in the SUT, one unknown fault has been located. The unknown fault has not been located by the old set of manual test cases. This also indicates that the coverage criteria might be good.

Based on the advantages of the model-based approach, Océ has decided to continue development of this prototype, and extend models to continue improving the testing process.

7 Modeling a Livestock Stable Controller

Since our approach shows promising results for modeling and testing printer controllers, and the literature shows similar approaches used in other types of systems, we wanted to examine if our approach could be applied in other companies. We have initiated contact with a company designing and producing ventilation systems for livestock stables, to examine how the approach applies there. The controller for these systems is split into several components, each of which monitors some input sensors and controls some output actuators. The inputs are continuous measurements of e.g. temperature. The outputs are either *on/off* values or a percentage value describing how much power should be given to, for instance, a ventilator. Calculation of the two types of output are done in a standard way.

For *on/off*-type of controllers there are two important parameters: t, and T_δ. The value of t describes when the output should be activated. To avoid oscillation between *on* and *off*, T_δ describes how far below t the input has to fall before deactivating the output. It can be observed that between $t - T_\delta$ and t this controller shows nondeterministic behavior. This nondeterminism can be seen as an internal state in the controller, storing its previous output value. For inputs between t and $t - T_\delta$ the controller outputs the same value as previously. For inputs below $t - T_\delta$ the output is always *off*, and above t the output is always *on*. Figure 2 illustrates the possible values.

For percentage-type of controllers the output value is a linear function of the input. The function is described by two parameters: p and P_δ. The value of p

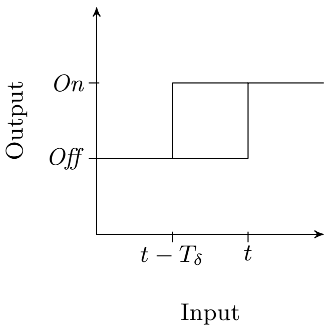


Fig. 2. Graph for *on/off* values

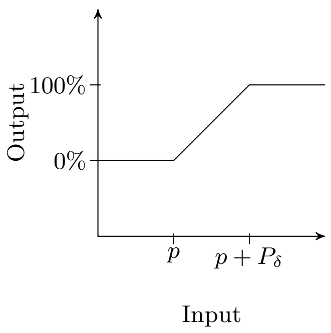


Fig. 3. Graph for percentage values

describes when the output must start increasing. P_δ describes how far above p the output must reach 100%. Below p the output is always 0%, above $p + P_\delta$ the output is always 100%. In between the output grows linearly from 0% to 100%. Figure 3 illustrates the function.

Components can have several inputs based on the same patterns, to form more complex components. For instance, a component can have a temperature reading and a humidity reading as input, and can activate a ventilation fan as output. The ventilation fan should be activated when the temperature reaches above some value while the humidity is below some value. Both inputs will have T_δ values for when the fan should be deactivated again.

Currently test cases for the system are generated manually. A test engineer examines the parameters of the component and generates a set of inputs generating an acceptable coverage, and generates corresponding expected outputs. Subsequently the test cases are executed automatically and the expected outputs are compared to the actual outputs.

This type of system can be modeled within our framework, with a single modification. We need to handle the nondeterministic behavior. This can be done by representing the model as a hybrid automaton[1] by handling the state as an input. We make a fresh input parameter to represent the state of the model. The domain of this parameter is the state space of the model. This way the clauses in the model can depend on the state the system is in and act accordingly. However, in this simple setting this seems like too complex a solution. We only need a single state variable; a Boolean. We only need to know the value of this variable one time step backward. This can be easily be handled in the current setting of transformational systems. The problem with adding this functionality is that pairwise test case generation will not work since test cases become traces, where each step depends on the previous one. Some work needs to be done to find a good way to generate test cases for this type of system. The test case generation needs to generate valid test cases and provide coverage of the data in input parameters as well as coverage of the state space of the model.

With this modification a large set of manually generated test cases, can be automatically generated from simple models. Future test cases can easily be created by instantiating the model with the required values.

This shows that the approach is indeed applicable for different types of systems, even though it was developed for testing printer controllers. The diversity of the SUTs shows that there are potentially several industrial areas where similar approaches could be applied to improve the testing process.

8 Conclusion

This paper has presented the initial steps towards a model-based testing framework for testing printer controllers at Océ. The approach has proved promising in improving the testing process and the quality of test cases. Advantages of the approach include: improved maintenance, reduced number of test cases, measurable coverage, and better understanding of the test results. While the approach seems promising at improving the testing process, further work is needed to state this for certain. While test cases can now be generated based on coverage requirements, it is unclear if the generated test set will locate as many errors as the old test cases. Also development of the model requires some significant effort, but it is expected to prove valuable in the long run. Based on the outcome of this project, Océ has decided to continue with the model-based approach. Finally, we illustrated that the approach is useful for modeling and testing controller software for ventilation systems in livestock stables. This diverse usefulness of our approach indicates that several other industrial areas could benefit from similar approaches.

The connection to the current way of working at Océ, and usability of the methods by Océ in their current environment, were among the main requirements and starting points of this project. This did not always lead to the most sophisticated, or theoretically new solutions, and sometimes even to ad-hoc solutions, e.g. to establish the connection to the existing Python tooling. Future work will include the use of more sophisticated approaches, such as using SAT-solvers or SMT tooling to solve, check, and manipulate Boolean formulas.

References

[1] Alur, R., Courcoubetis, C., Henzinger, T., Ho, P.: Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In: Grossman, R.L., Ravn, A.P., Rischel, H., Nerode, A. (eds.) HS 1991 and HS 1992. LNCS, vol. 736, pp. 209–229. Springer, Heidelberg (1993)

[2] Burr, K., Young, W.: Combinatorial test techniques: Table-based automation, test generation and code coverage. In: Proceedings of the Intl. Conf. on Software Testing Analysis and Review, pp. 503–513. West (1998)

[3] Calvagna, A., Gargantini, A.: A logic-based approach to combinatorial testing with constraints. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 66–83. Springer, Heidelberg (2008)

[4] Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of haskell programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP 2000, pp. 268–279. ACM, New York (2000)
[5] Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C.: The aetg system: An approach to testing based on combinatorial design. IEEE Trans. Softw. Eng. 23, 437–444 (1997)
[6] Cohen, D.M., Dalal, S.R., Parelius, J., Patton, G.C.: The combinatorial design approach to automatic test generation. IEEE Software 13(5), 83–88 (1996)
[7] DeMillo, R.A., Offutt, A.J.: Constraint-based automatic test data generation. IEEE Transactions on Software Engineering 17(9), 900–910 (1991)
[8] Dunietz, I.S., Ehrlich, W.K., Szablak, B.D., Mallows, C.L., Iannino, A.: Applying design of experiments to software testing: experience report. In: Proceedings of the 19th International Conference on Software Engineering, ICSE 1997, pp. 205–215. ACM, New York (1997)
[9] Koopman, P., Alimarine, A., Tretmans, J., Plasmeijer, R.: Gast: generic automated software testing. In: Peña, R., Arts, T. (eds.) IFL 2002. LNCS, vol. 2670, pp. 84–100. Springer, Heidelberg (2003)
[10] Kuhn, R., Reilly, M.: An investigation of the applicability of design of experiments to software testing. In: Proceeding of the 27th NASA/IEEE Software Engineering Workshop. IEEE, Los Alamitos (2002)
[11] Lozano, R.C., Schulte, C., Wahlberg, L.: Testing continuous double auctions with a constraint-based oracle. In: Cohen, D. (ed.) CP 2010. LNCS, vol. 6308, pp. 613–627. Springer, Heidelberg (2010)
[12] Wallace, D.R., Kuhn, D.R.: Failure modes in medical device software: an analysis of 15 years of recall data. In: ACS/ IEEE International Conference on Computer Systems and Applications, pp. 301–311 (2001)

A Real-World Benchmark Model for Testing Concurrent Real-Time Systems in the Automotive Domain

Jan Peleska[1], Artur Honisch[3], Florian Lapschies[1], Helge Löding[2], Hermann Schmid[3], Peer Smuda[3], Elena Vorobev[1], and Cornelia Zahlten[2]

[1] Department of Mathematics and Computer Science, University of Bremen, Germany
{jp,florian,elenav}@informatik.uni-bremen.de
[2] Verified Systems International GmbH, Bremen, Germany
{hloeding,cmz}@verified.de
[3] Daimler AG, Stuttgart, Germany
{artur.honisch,hermann.s.schmid,peer.smuda}@daimler.com

Abstract. In this paper we present a model for automotive system tests of functionality related to turn indicator lights. The model covers the complete functionality available in Mercedes Benz vehicles, comprising turn indication, varieties of emergency flashing, crash flashing, theft flashing and open/close flashing, as well as configuration-dependent variants. It is represented in UML2 and associated with a synchronous real-time systems semantics conforming to Harel's original Statecharts interpretation. We describe the underlying methodological concepts of the tool used for automated model-based test generation, which was developed by Verified Systems International GmbH in cooperation with Daimler and the University of Bremen. A test suite is described as initial reference for future competing solutions. The model is made available in several file formats, so that it can be loaded into existing CASE tools or test generators. It has been originally developed and applied by Daimler for automatically deriving test cases, concrete test data and test procedures executing these test cases in Daimler's hardware-in-the-loop system testing environment. In 2011 Daimler decided to allow publication of this model with the objective to serve as a "real-world" benchmark supporting research of model based testing.

1 Introduction

Model-based testing. Automated *model-based testing (MBT)* has received much attention in recent years, both in academia and in industry. This interest has been stimulated by the success of model-driven development in general, by the improved understanding of testing and formal verification as complementary activities, and by the availability of efficient tool support. Indeed, when compared to conventional testing approaches, MBT has proven to increase both quality and efficiency of test campaigns; we name [13] as one example where quantitative evaluation results have been given. In this paper the term model-based testing is used in the following, most comprehensive, sense: the behavior of the *system under test (SUT)* is specified by a model elaborated in the same style as a model serving for development purposes. Optionally, the SUT model can be paired with an environment model restricting the possible interactions of the environment with the SUT. A *symbolic test case generator* analyzes the model and specifies *symbolic test cases* as logical formulae identifying model computations suitable for a certain *test purpose* (also called *test objective*). Symbolic test cases may be represented as LTL formulas of the form $\mathbf{F}\phi$, expressing that finally the test execution should produce a computation fragment where the test purpose specified by ϕ is fulfilled.

Constrained by the transition relations of SUT and environment model, a *solver* computes concrete model computations which are *witnesses* of symbolic test cases $\mathbf{F}\phi$. More formally, solvers elaborate solutions of so-called *bounded model checking instances*

$$tc(c, G) \equiv_{\text{def}} \bigwedge_{i=0}^{c-1} \Phi(\sigma_i, \sigma_{i+1}) \wedge G(\sigma_0, \dots, \sigma_c) \tag{1}$$

In this formula σ_0 represents the current model state and Φ the transition relation associated with the given model, so any solution of (1) is a valid model computation fragment of length c. Intuitively speaking, $tc(c, G)$ tries to solve LTL formula $\mathbf{F}\phi$ within c computation steps, starting in model pre-state σ_0, so that each step is a valid model transition, and test purpose ϕ is encoded in $G(\sigma_0, \dots, \sigma_c)$.

The inputs to the SUT obtained from these computations are used in the test execution to stimulate the SUT. The SUT behavior observed during the test execution is compared against the *expected* SUT behavior specified in the original model. Both stimulation sequences and *test oracles*, i. e., checkers of SUT behavior, are automatically transformed into *test procedures* executing the concrete test cases in a software-in-the-loop or hardware-in-the-loop configuration.

Observe that this notion of MBT differs from "weaker" ones where MBT is just associated with some technique of graphical test case descriptions. According to the MBT paradigm described here, the focus of test engineers is shifted from test data elaboration and test procedure programming to modeling. The effort invested into specifying the SUT model results in a return of investment, because test procedures are generated automatically and debugging deviations of observed against expected behavior is considerably facilitated because the observed test executions can be "replayed" against the model.

Objectives and Main Contributions. The main objective of this article is to present a "real-world" example of a model used in the automotive industry for system test purposes. The authors have experienced the "validation powers" of such models with respect to realistic assessment of efforts in model development, and with respect to the tool capabilities required to construct concrete model computations – i. e., test data – for given symbolic test cases. We hope to stimulate a competition of alternative methods and techniques which can be applied to the same benchmark model in order to enable objective comparison of different approaches. For starting such a competition we also give an overview of the methods and algorithms applied in our tool and present performance values, as well as test generation results to be compared with the results obtained using other methods and tools.

To our best knowledge, comparable models of similar size, describing concurrent real-time behavior of automotive applications and directly derived from industrial applications are currently not available to the public, at least not for application in the MBT domain. As a consequence, no systematic approach to benchmark definitions has been made so far. We therefore suggest a simple classification schema for such benchmarks, together with a structuring approach for the test suites to be generated. While this article can only give an overview of the model, detailed information are publicly available on the website [17] (`www.mbt-benchmarks.org`).

Overview. In Section 2 we present an introductory overview over the benchmark model. In Section 3 the methods applied in our test generator are sketched, with the objective to stimulate discussions about the suitability of competing methods. The representation of symbolic test cases as *constraint solving problems (CSP)* is described, and we sketch how these CSPs are solved by the *test generation engine* in order to obtain concrete test stimulations to be passed from the test environment to the SUT.

In Section 4 we propose a classification of benchmarks which are significant for assessing the effectiveness and performance of model-based testing tools. This classification induces a structure for reference test suites. In Section 5 a test generation example is presented.

Since the complete model description and the detailed explanation of algorithms used for test case generation and CSP solving is clearly beyond the page restriction of this submission, interested readers are referred to [17], where the material is presented in more comprehensive form, and the model can be downloaded in XMI format and as a model file for the EnterpriseArchitect CASE tool [23] which was used to create the model and its different export formats. Additionally an archive may be downloaded whose files allow to browse through the model in HTML format. The symbolic test cases used in the performance results are also available, so that the sequence of CSP solutions created by our test case generator can be repeated by other tools. The current benchmark evaluation results, including test generations performed with our tool are also published there.

Related Work. Benchmarking has been addressed in several testing domains. For "classical" software testing the so-called *Siemens benchmarks* [9] provide a collection of C programs with associated mutations rated as representative for typical programming bugs. A more comprehensive discussion and review of available software testing benchmarks is given in [14]. In [15] an initiative for event-driven software testing benchmarks has been launched.

In the field of model-based embedded systems testing only very few benchmarks are currently available, and none of them appear to describe comprehensive industrial control applications. In [10] a Matlab/Simulink model for a flight control system has been published. According to our classification proposed in Section 4 it addresses the benchmark category *test strength benchmarks*: A set of test cases is published which have been generated using random generation techniques inspired by Taguchi methods. To analyze the strength of test cases, several mutants of the model have been provided which may either be executed in Simulink simulation mode or as C programs generated from the mutant models. Since random test generation techniques on the input interface to the SUT are used, the model coverage achieved is only analyzed after test suite execution. As a consequence, no *test generation benchmarks* suggested in Section 4 are discussed. All existing benchmarks we are aware of may be classified as test strength benchmarks. Our proposition of test generation benchmarks seems to be a novel concept.

While our test generation approach relies on constraint solvers to find test-input-data, *search-based testing* techniques use randomized methods guided by optimization goals. In [1] the use of random testing, adaptive random testing and genetic algorithms for use in model-based black-box testing of real-time systems is investigated. To this end, the test-environment is modeled in UML/MARTE while the design of the SUT is not modeled at all, since all test data are derived from the possible environment behavior. An environment simulator is derived from the model that interacts with the SUT and provides the inputs selected by one of the strategies. The environment model also serves as a test oracle that reports errors as soon as unexpected reactions from the SUT are observed. This and similar approaches are easier to implement than the methods described in this paper, because there is no need to encode the transition relation of the model and to provide a constraint solver, since concrete test data is found by randomized model simulations. We expect, however, that the methods described in [1] do not scale up to handle systems of the size presented here, where the concurrent nature of the SUT requires to consider the interaction between several components in real-time (the model would be too large to construct a single large product automaton from the many smaller ones describing the component behavior). To the best knowledge of the authors there is no work on using search based testing on synchronous parallel real-time systems in order to achieve a high degree of SUT coverage, let alone to find test input data to symbolic test-cases.

The solutions presented here have been implemented in the RT-Tester test automation tool which provides an alternative to TRON [16,6] which supports timed automata test models and is also fit for industrial-strength application. TRON is complementary to RT-Tester, because it supports an interleaving semantics and focuses on event-based systems, while RT-Tester supports a synchronous semantics with shared variable interfaces. RT-Tester also competes with the Conformiq Tool Suite [5], but focuses stronger on embedded systems testing with hard real-time constraints.

2 Model Overview

General. Our MBT benchmark model specifies the *turn indicator functions* available in Mercedes Benz cars; this comprises left-/right turn indication, emergency flashing, crash flashing, theft flashing and open/close flashing. The level of detail given in the model corresponds to the observation level for system testing. To provide the full functionality, several automotive controllers cooperate using various communication busses (CAN and LIN). The signals exchanged between controllers can be observed by the testing environment; additionally the environment can stimulate and monitor discrete and analogue interfaces between SUT and peripherals, such as switches, buttons, indicator lights and various dashboard indications. Capturing this functionality in a formal way requires a concurrent real-time system semantics.

System Interface. In Fig. 1 the interface between system under test (SUT) and testing environment (TE) is shown. Due to the state-based nature of the hardware interfaces (discretes, periodic CAN or LIN bus messages repeatedly sending state information) the modeling formalism handles interfaces as shared variables written to by the TE and read from by the SUT or vice versa.

The TE can stimulate the SUT via all interfaces affecting the turn indication functionality in the operational environment: $\mathsf{in\_CentralLockingRM} \in \{0, 1, 2\}$ denotes the remote control for opening and closing (i. e. unlocking and locking) cars by means of the central locking system. Signal $\mathsf{in\_CrashEvent} \in \{0, 1\}$ activates a crash impact simulator which is part of the TE, and $\mathsf{in\_EmSwitch} \in \{0, 1\}$ simulates the "not pressed"/"pressed" status of the emergency flash switch on the dashboard. Signal $\mathsf{in\_IgnSwitch} \in \{0, \dots, 6\}$ denotes the current status of the ignition switch, and $\mathsf{in\_TurnIndLvr} \in \{0, 1, 2\}$ the status of the turn indicator lever (1 = left, 2 = right). In special-purpose vehicles (SPV), such as taxis or police cars, additional redundant interfaces for activation of emergency flashing and turn indicators exist (e. g., $\mathsf{in\_EmSwitchSPV} \in \{0, 1\}$). Observe that these redundant interfaces may be in conflicting states, so that the control software has to perform a priority-dependent resolution of conflicts. Inputs to the SUT marked by OPTION specify different variants of vehicle style and equipments, each affecting the behavior of the turn indication functions. In contrast to the other input interfaces to the SUT, options remain stable during execution of a test procedure, since their change requires a reset of the automotive controllers, accompanied by a procedure for loading new option parameters. If the TE component does not contain any behavioral specifications, the test generator will create arbitrary timed sequences of input vectors suitable to reach the test goals, only observing the range specifications associated with each input signal. This may lead to unrealistic tests. Therefore the TE may be decomposed into concurrent components (typically called *simulations*) whose behavior describe the admissible (potentially non-deterministic) interaction of the SUT environment on some or all interfaces. The test generator interprets these simulations as additional constraints, so that only sequences of input vectors are created, whose restrictions to the input signals controlled by TE components comply with the transition relations of these simulations.

SUT outputs are captured in the SignalsOut interface (Fig. 1 shows only a subset of them). The indicator lights are powered by the SUT via interfaces pwmRatio_FL, pwmRatio_FR, $\ldots \in \{0, \ldots, 120\}$ where, for example, FL stands for "forward left" and RR for "rear right". The TE measures the percentage of the observed power output generated by the lamp controllers, 100% denoting identity with the nominal value. System integration testing is performed in grey box style: apart from the SUT outputs observable by end users, the TE also monitors bus messages produced by the cooperating controllers performing the turn indication service. Message tim_EFS $\in \{0, 1\}$, for example, denotes a single bit in the CAN message sent from a central controller to the peripheral controllers in order to indicate whether the emergency flash switch indicator on the dashboard should be activated, and tim_FL $\in \{0, 1\}$ is the on/off command to the controller managing the forward-left indicator light.

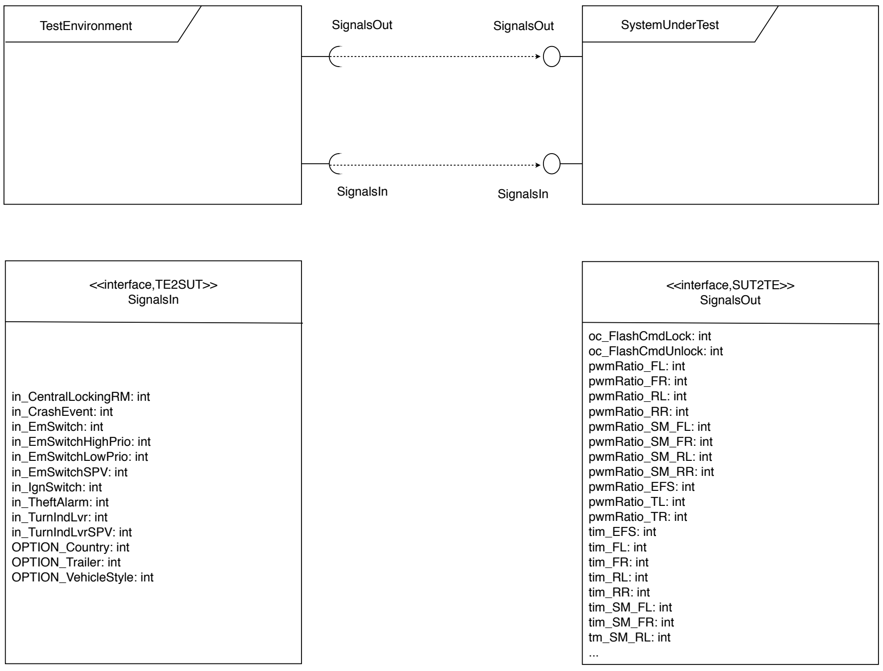


Fig. 1. Interface between test environment and system under test

First-Level SUT Decomposition. Fig. 2 shows the functional decomposition of the SUT functionality. Component NormalAndEmerFlashing controls left/right turn indication, emergency flashing and the dependencies between both functions (see below). Component OpenCloseFlashing models the indicator-related reactions to the locking and unlocking of vehicles with the central locking system. CrashFlashing models indications triggered by the crash impact controller. TheftFlashing controls reactions triggered by the theft alarm system. These functions interact with each other, as shown in the interface dependencies depicted in Fig. 2: the occurrence of a crash, for example, affects the emergency flash function, and opening a car de-activates a theft alarm. The local decisions of the above components are fed into the PriorityHandling component where conflicts between indication-related commands are resolved: if, for example, the central locking system is activated while emergency flashing is active, the open/close flashing patterns (one time for open, 3 times for close) are not generated; instead, emergency flashing continues. Similarly, switching off the emergency switch has no effect if the high-priority emergency interface ($\mathsf{in\_EmSwitchHighPrio} \in \{0, 1\}$) is still active. Priority handling identifies the function to be performed and relays the left-hand/right-hand/both sides flashing information to the components OnOffDuration and AffectedLamps. The former determines the durations for switching lights on and off, respectively, during one flashing period. These durations depend both on the status of the ignition switch and the function to be performed. The latter specifies which lamps and dashboard indications have to participate in the flashing cycles. This depends on the OPTION_VehicleStyle which determines, for example, the existence of side marker lamps (interfaces pwmRatio_SM_FL, FR, RL, RR), and on the OPTION_Trailer which indicates the existence of a trailer coupling, so that the trailer turn indication lamps (pwmRatio_TL, TR) have to be activated. Moreover, the affected lamps and indications depend on the function to be performed: open-close flashing, for example, affects indication lamps on both sides, but the emergency flash switch indicator (pwmRatio_EFS) is not activated, while this indicator is affected by emergency, crash and theft flashing. The MessageHandling component transmits duration and identification of affected lamps and indicators on a bus and synchronizes the flash cycles by re-transmission of this message at the beginning of each flashing cycle. Finally, component LampControl comprises all output control functions, each function controlling the flashing cycles of a single lamp or dashboard indicator.

Behavioral Semantics. Model components behave and interact according to a concurrent synchronous real-time semantics, which is close to Harel's original micro-step semantics of Statecharts [8]. Each leaf component of the model is associated with a hierarchic state machine. At each step starting in some model pre-state σ_0, all components possessing enabled state machine transitions process them in a synchronous manner, using σ_0 as the pre-state. The writes of all state machine transitions affect the post-state σ_1 of the micro-step. Two concurrent components trying to write different values to the same variable in the same micro-step cause a *racing condition* which is reflected by deadlock of the transition relation and – in contrast to interleaving semantics – considered as a modeling error. Micro-steps are discrete transitions performed in zero time. Inputs to the SUT remain unchanged between discrete transitions. If the system is in a stable state, that is, all state machine transitions are disabled, time passes in a delay transition, while the system state remains stable. The delay must not

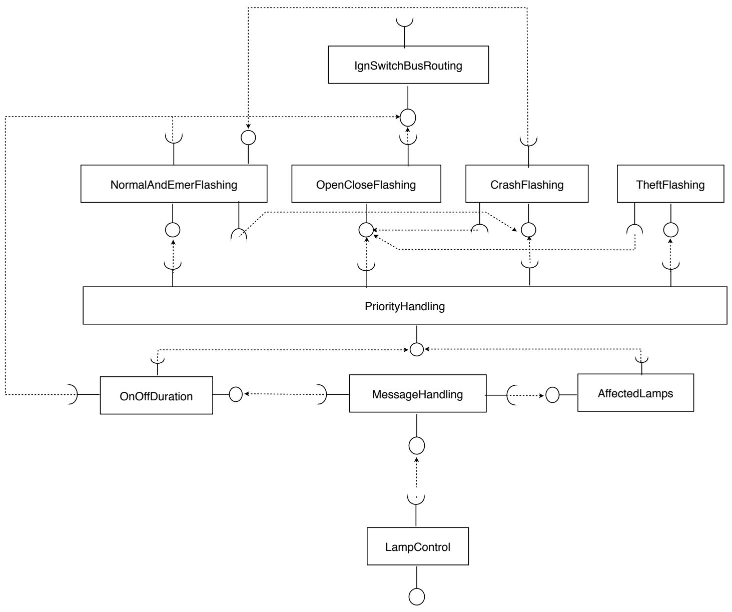


Fig. 2. First-level decomposition of system under test

exceed the next point in time when a discrete transition becomes enabled, due to a timeout condition. At the end of a delay transition, new inputs to the SUT may be placed on each interface. The distinction between discrete and delay transitions is quite common in concurrent real-time formalisms, and it is also applied to interleaving semantics, as, for example, in Timed Automata [21]. The detailed formal specification of the semantic interpretation of the model is also published on the website given above [19].

3 Benchmark Reference Tool

Tool Components and Basic Concepts. The reference data for the benchmarks have been created using our model-based testing tool RT-Tester. It consists of a parser front-end transforming textual model representations (XMI export provided by the CASE tool) into internal representations of the abstract model syntax.

A constraint generator derives all model coverage goals from the abstract syntax tree and optionally inputs user-defined symbolic test cases. Users may select which symbolic test cases should be discharged in the same test procedure. A transition relation generator traverses the model's abstract syntax tree and generates the model transition relation Φ needed for expressing computation goals according to Equation (1). During the test data and test procedure generation process, the constraints associated with these symbolic test cases are passed on to an abstract interpreter. The interpreter performs an abstract conservative approximation of the model states that are reachable from the current model state within a pre-defined number n of steps. The goals which may be covered within n steps according to the abstract interpretation are passed on in disjunctive form to an SMT solver. The solver unrolls the transition relation in a step-by-step manner and tries to solve at least one of the goals. If this succeeds, a timed sequence of input vectors to the SUT is extracted from the solution provided by the SMT solver. Starting from the current model state, a concrete interpreter executes this sequence until a new stable state is reached where further inputs may be generated to cover the remaining goals. If the solver cannot discharge any goal within n steps, random simulations and/or backtracking to model states already visited can be performed in order to identify other model states from where the next goal may be reached. Constraint generator, interpreters and solver represent the core of the tool, called the *test generation engine*. Its components only depend on the abstract syntax representation of the model and its transition relation, but not on the concrete modeling syntax and the syntax required by the test execution environment for the test procedures.

At the end of the generation process a multi-threaded test procedure is generated which stimulates the SUT according to the input sequences elaborated by the solver and simultaneously checks SUT reactions with respect to consistency with the model. In the sections below we highlight the most important features of the tool; a detailed description is given in [19].

SMT-Solver. The constraint solving problems of type (1) may contain linear and non-linear arithmetic expressions, bit-operations, array-references, comparison predicates and the usual Boolean connectives. Data types are Booleans, signed and unsigned integers, IEEE-754 floating-point numbers and arrays.

Our SMT-solver SONOLAR uses the classical bit-blasting approach that transforms a formula to a propositional satisfiability problem and lets a SAT-solver try to find a solution [11,2]. Variables in the formula are translated to vectors of propositional variables (i. e., *bit vectors*). The lengths of these bit vectors correspond to the bit width of the respective data types. Operations are encoded as propositional constraints relating input to output bit vectors. Since we reason on bit-level, this enables us to precisisely capture the actual semantics of all operations. Integer arithmetic takes potential overflows into account and each floating point operation is correctly rounded to the selected IEEE-754 rounding-mode.

To this end, the formula is first represented as an acyclic expression graph, where each variable and each operation of the formula is represented as a node. Using structural hashing on these nodes, identical terms are shared among expressions. This representation allows us to perform word-level simplifications, normalization and substitutions. The expression graph is then bit-blasted to an *And-Inverter Graph (AIG)*. AIGs are used by several SMT solvers to synthesize propositional formulas [11,2,12]. Each node of an AIG is either a propositional variable or an *and*-node with two incoming edges that may optionally be inverted, i.e. negated. The AIG is structually hashed and enables us to perform bit-level simplifications. Readers are referred to [7,3] for more information on logic synthesis using AIGs. The AIG is then translated to CNF using the standard Tseitin encoding and submitted to a SAT solver.

In order to handle the extensional theory of arrays we adopted the approach described in [4]. Instead of bit-blasting all array expressions to SAT up-front, array expressions that return bit-vectors associated with array-reads or checks for array equality are replaced by fresh variables. This results in an over-abstraction of the actual formula since the array axioms are left out. If the SAT solver is able to find a solution to this formula the model is checked for possible array inconsistencies. In this case, additional constraints are added on-demand to rule out this inconsistency. This process is repeated until either the SAT solver finds the refined formula to be unsatisfiable or no more array inconsistencies can be found. While unrolling the transition relation constraints are incrementally added to the SMT solver.

Abstract Interpretation. Our abstract interpreter has been developed to compute over-approximations of possible model computations in a fast way. Its main application is to determine lower bounds of the parameter c in Formula (1) specifying the number of times the transition relation Φ must be unrolled before getting a chance to solve $tc(c, G)$. This considerably reduces generation time, because (a) the SMT solver can skip solution trials for $tc(c, G)$ with values of c making a solution of $tc(c, G)$ infeasible, and (b) the abstract interpretation technique provides the means for non-chronological backtracking in situations where it is tried to solve $tc(c, G)$ from a former model state already visited (see [18] for a more detailed description).

The abstract interpreter operates on abstract domains: interpretation of model behavior is performed using an abstract state space Σ_A instead of the concrete one. Σ_A is obtained by replacing each concrete data type D of the concrete state space Σ with an adequate abstract counterpart $L(D)$. Functions defined over concrete data types $D_0, \ldots, D_n$ are lifted to the associated abstract domains $L(D_0), \ldots, L(D_n)$. In order to be able to reason about concrete computations while computing only the abstract ones, concrete and abstract states are related to one another by Galois connections. A Galois connection is a tuple of mappings $(\triangleright : \mathbb{P}(\Sigma) \rightarrow \Sigma_A, \triangleleft : \Sigma_A \rightarrow \mathbb{P}(\Sigma))$ defining for any set of concrete states the associated abstract state and vice versa, see [18] for additional details. Finally, each abstract domain $L(D)$ is equipped with a join operator $\sqcup : L(D) \times L(D) \rightarrow L(D)$ which establishes the basis for the join operator over two abstract states $\sqcup : \Sigma_A \times \Sigma_A \rightarrow \Sigma_A$. This operator is essential as it allows to reduce the complexity usually arising when interpreting large models where the computation of all reachable states would otherwise be infeasible, due to the number of states and the number of control decisions.

The abstract interpreter uses the interval, Boolean and power set lattices as abstract domains for numerical data types, Booleans and state machine locations, respectively. The interpretation of a given model is parametrized by an initial abstract state $\sigma_A^0 \in \Sigma_A$, an integer c_{max} denoting the maximal number of transition steps to be interpreted and a test case goal G to be checked for satisfiability. Starting in the given initial state, the interpreter computes a sequence of up to c_{max} abstract states $\langle \sigma_A^1, \sigma_A^2, \dots \rangle$ where each state σ_A^{i+1} is guaranteed to "include"[1] every concrete state σ^{i+1} reachable from any of the concrete states represented by σ_A^i. The interpretation stops as soon as either the maximal number of steps has been reached or the test case goal G evaluates to `true` or $\top$[2]. In the latter case the actual step number is returned.

4 MBT Benchmark Classification

We propose to classify MBT benchmarks according to the following characteristics, denoted by *test strength benchmarks* and *test generation benchmarks.*

Test strength benchmarks investigate the error detection capabilities of concrete test cases and test data generated by MBT tools: even if two MBT tools produce test suites of equivalent model coverage, they will usually possess different strength, due to different choices of symbolic test cases, representatives of equivalence classes, boundary values and timing of input vectors passed to the SUT, or due to different precision of the test oracles generated from the model. Mutation testing is an accepted approach to assessing test suite strength; therefore we suggest to generate model mutants and run test suites generated from the unbiased model as model-in-the-loop tests against these mutants. The evaluation criterion is the percentage of uncovered mutations for a fixed set of mutant models.

Test generation benchmarks input symbolic test cases as introduced in Section 3 and measure the time needed to generate concrete test data. We advocate a standard procedure for providing these test objectives for a given model, structuring symbolic test cases into several sets. The first sets should be related to model coverage criteria [24], such as (1) control state coverage (every control state of the SUT model is visited by at least one test), (2) state machine transition coverage (every transition of every state machine is taken at least once) and (3) MC/DC coverage (conditions of the type $\phi_1 \wedge \phi_2$ are tested at least once for each of the valuations $(\phi_1, \phi_2) = (\texttt{false}, \texttt{true}), (\texttt{true}, \texttt{false}), (\texttt{true}, \texttt{true})$, and conditions of the type $\phi_1 \vee \phi_2$ are tested at least for $(\phi_1, \phi_2) = (\texttt{false}, \texttt{false})$, $(\texttt{true}, \texttt{false}), (\texttt{false}, \texttt{true})$).

Conventional model coverage criteria as the ones listed in (1 — 3) do not possess sufficient strength for concurrent real-time systems, because the dependencies between state machines operating in parallel are not sufficiently addressed. Since models as the one under consideration are too large to consider coverage of all state vector combinations, a pragmatic compromise is to maximize the coverage of all basic control state pairs of interacting components C_1, C_2, combined with pairs of input equivalence classes of signals influencing C_1, C_2.

[1] In the sense that $\sigma^{i+1} \in (\sigma_A^{i+1})^{\triangleleft}$.

[2] $\top$ is the Boolean lattice value representing the concrete value set $\{\texttt{false}, \texttt{true}\}$.

As a consequence we suggest symbolic test cases consisting of (a subset of) these combinations as a forth set. As a fifth set of symbolic test cases it is proposed to define application-specific test cases of specific interest.

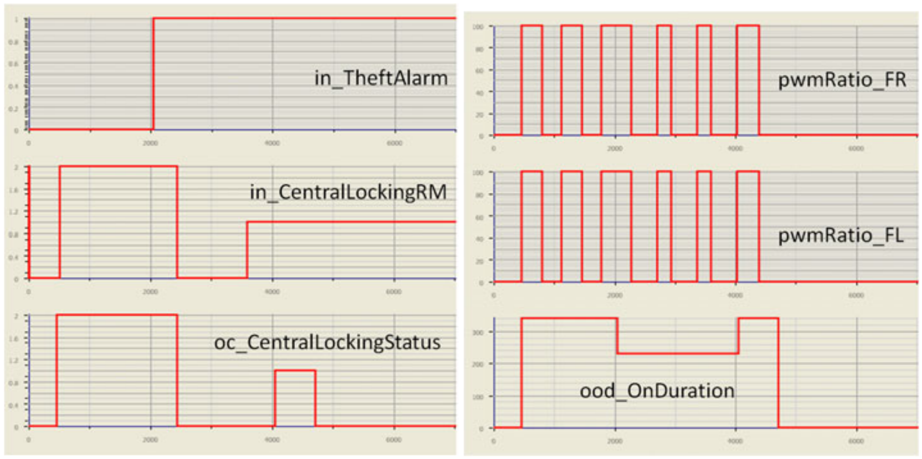
Given these classes for a specific model, this induces 5 test suites to realize a comprehensive test generation benchmark.

Evaluation criteria for test generation benchmarks. Apart from the time needed to generate concrete test data, the number of SUT resets involved in the resulting test procedures should be minimized as well: since SUT resets usually consume significant time when testing embedded systems, hardware-in-the-loop tests avoiding resets significantly reduce the test suite execution time. Moreover, test executions covering many test cases drive the SUT into more internal states than test executions resetting the SUT between two or only a small number of test cases. As a consequence, the error detection capabilities of test procedures are usually increased with the number of test cases they cover between resets. Avoiding resets is adverse to the reduction of generation time: if some goal $\mathbf{F}\phi$ is very time consuming to reach from a given model state σ_0, backtracking to a former model state from where computation fragments fulfilling $\mathbf{F}\phi$ can be reached more easily frequently helps to reduce the generation time in a significant way. Since the SUT is usually unable to roll back into a previous state, backtracking enforces a SUT reset, after which the new computation can be exercised. For comparing the performance of tools we suggest to calculate Pareto frontiers of pairs *(generation time,number of resets)* for each competing tool, and compare the tool-dependent frontiers.

Significance of test generation benchmarks. According to standards applicable to safety-critical systems verification [22,20] the error detection strength of a test suite is just one aspect to be addressed when justifying the adequateness of test cases. Complementary to that the standards require to account for sufficient coverage on the levels of requirements, design and code. As a consequence, the capability of test automation tools to generate sufficient test cases to achieve such coverage has significant impact on verification and certification efforts.

5 Test Generation Example

The benchmark website [17] contains two classes of symbolic test cases: (1) *user-defined test cases* reflect specific test purposes identified by test engineers. They serve either to test more complex requirements which cannot be traced in the model to simple sets of basic control states or transitions to be covered, or they are used to explore SUT reactions in specific situations where a failure is suspected. (2) *Model-defined test cases* aim at covering certain parts of the model according to pre-defined strategies, such as basic control state coverage, state machine transition coverage and MC/DC coverage. They are automatically derived by our tool from the abstract syntax representation of the model.



(a) Generated inputs and internal model state `oc_CentralLockingStatus`.

(b) Expected outputs and internal model state `ooo_OnDuration`.

Fig. 3. Generation results of theft alarm test procedure

In this section a test generation based on user-defined test cases from [17, Test UD 003] is presented. The underlying test purpose is to investigate the interaction between theft alarm and open/close flashing: theft alarm flashing is only enabled when the doors are locked. As a reaction to alarm-sensor activation the turn indicator lights shall start flashing on both sides. Pressing the remote control key to unlock the doors automatically shuts off the alarm flashing.

Instead of explicitly determining the sequence of timed input vectors to the SUT which is suitable for covering the test purpose described, we specify simpler and shorter symbolic test cases that may also refer to internal model states and leave it up to the tool's generation engine to calculate the concrete test input data and its timing. Symbolic test case[3]

```
TC-turn_indication-THEFT_ALARM-0001;
   [ SystemUnderTest.TheftFlashing.TheftFlashing.THEFT_ALARM_ACTIVE.ALARM_OFF
   && ! in_TheftAlarm ]
   Until
   [ _timeTick >= 2000 && in_TheftAlarm ]
```

refers to SUT inputs (theft alarm-sensor `in_TheftAlarm`) the model execution time (`_timeTick`) and basic control states of state machines which are part of the SUT model (`SystemUnderTest.TheftFlashing...ALARM_OFF`). The LTL formula is a directive to the test generation engine to find a computation which finally reaches a model state where theft alarm flashing is enabled but not yet active (this is the case when the basic control state `...ALARM_OFF` is reached), and the theft alarm-sensor should remain passive until 2000ms have passed since

[3] The symbols used in the test cases below are taken from the turn indicator model published in [17]. A detailed description of inputs, outputs and internal model variables can be found there.

start of test (the leading finally operator is always omitted in our symbolic test case specifications). The inputs derived by the generation engine to cover this test case drive the SUT into a state where an alarm is signaled and the SUT has to react by activating theft alarm flashing. The next symbolic test case to be processed is

```
TC-turn_indication-THEFT_ALARM-0002;
  [ SystemUnderTest.TheftFlashing.TheftFlashing.THEFT_ALARM_ACTIVE.ALARM_ON
    && in_TheftAlarm ]
  Until
  [ _timeTick >= 4000 && IMR.in_TheftAlarm &&
    SystemUnderTest.oc_CentralLockingStatus == 1 ]
```

This formula is a directive to stay in the theft alarm state for at least another 2 seconds after which a model state is to be reached where the internal model variable `oc_CentralLockingStatus` has value 1 (= *"unlocked"*), indicating that an *"unlock doors"* command has been given via remote key control. Again, the associated inputs and timing is calculated by the test generation engine. The final symbolic test case to be processed by the generator is

```
TC-turn_indication-THEFT_ALARM-0003;
  [ SystemUnderTest.TheftFlashing.TheftFlashing.THEFT_ALARM_OFF ]
  Until
  [ _timeTick >= 6000 &&
  SystemUnderTest.TheftFlashing.TheftFlashing.THEFT_ALARM_OFF ]
```

It is a directive to stay in the *"theft alarm disabled"* state `...THEFT_ALARM_OFF` for at least another two seconds, so that it can be observed that after one flash period signaling that the doors have been unlocked, no further alarm indications are made. The signal flow associated with this test (inputs and expected SUT outputs) is depicted in Fig. 3. The generator created a sequence of input vectors where first doors are closed by means of the remote key control input `in_CentralLockingRM` (2 = lock, 1 = unlock). This triggers three flashing periods for left and right indicator lamps (`pwmRatio_FR, pwmRatio_FL`, see Fig. 3b). For open/close flashing the on-duration of a flashing period is 340ms; this is captured in the internal model variable `ood_OnDuration` whose contents will be transmitted by the SUT via CAN bus and can therefore be observed and checked by the threads running on the test engine and acting as test oracles. After two seconds an alarm is raised by setting `in_TheftAlarm = 1`. This changes the on-duration of the flashing period to 220ms. Theft alarm flashing is switched off at model execution time stamp 4000, approx. 500ms after the unlock-doors signal has been given (`in_CentralLockingRM=1`); the change of the internal `oc_CentralLockingStatus` from 0 to 1 indicates that the *"doors unlocked"* status has now been realized (see Fig. 3a). One flashing period signals "doors unlocked" (again with on-duration 340ms), after which no further alarm indications occur.

6 Conclusion

We have presented a model of an automotive control application, covering the full functionality related to turn indication, emergency flashing, crash, theft and open/close flashing. The model is a 1-1-transcription of the one currently used by Daimler for system testing of automotive controllers. As the only adaptation we have presented the model in pure UML 2.0 style, while Daimler uses a specific UML profile optimized for their hardware-in-the-loop testing environment. The model is made available to the public in complete form through the website [17], together with benchmark test suites and performance values achieved with our reference tool. Additionally, a classification of benchmarks for embedded systems test tools has been suggested which takes into account both the test strength and the performance for automated test suite generation.

The underlying methods of the MBT tool used by the authors for performing embedded systems test have been described, in order to facilitate the comparison of competing techniques applied to the benchmarks in the future. The tool is currently applied in industrial projects in the automotive, railway and avionics domains.

References

1. Arcuri, A., Iqbal, M.Z., Briand, L.: Black-box system testing of real-time embedded systems using random and search-based testing. In: Petrenko, A., Simão, A., Maldonado, J.C. (eds.) ICTSS 2010. LNCS, vol. 6435, pp. 95–110. Springer, Heidelberg (2010)
2. Brummayer, R.: Efficient SMT Solving for Bit-Vectors and the Extensional Theory of Arrays. Ph.D. thesis, Johannes Kepler University Linz, Austria (November 2009)
3. Brummayer, R., Biere, A.: Local two-level and-inverter graph minimization without blowup. In: Proceedings of the 2nd Doctoral Workshop on Mathematical and Engineering Methods in Computer Science, MEMICS 2006 (2006)
4. Brummayer, R., Biere, A.: Lemmas on Demand for the Extensional Theory of Arrays. In: Proc. 6th Intl. Workshop on Satisfiability Modulo Theories (SMT 2008). ACM, New York (2008)
5. Conformiq Tool Suite (2010), http://www.conformiq.com
6. David, A., Larsen, K.G., Li, S., Nielsen, B.: Timed testing under partial observability. In: Proc. 2nd International Conference on Software Testing, Verification and Validation (ICST 2009), pp. 61–70. IEEE Computer Society, Los Alamitos (2009)
7. Eén, N., Mishchenko, A., Sörensson, N.: Applying logic synthesis for speeding up SAT. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 272–286. Springer, Heidelberg (2007)
8. Harel, D., Naamad, A.: The statemate semantics of statecharts. ACM Transactions on Software Engineering and Methodology 5(4), 293–333 (1996)
9. Harrold, M.J., Rothermel, G.: Siemens programs, hr variants, http://ww.cc.gatech.edu/aristotle/Tools/subjects
10. Jeppu: A benchmark problem for model based control systems tests - 001, http://www.mathworks.com/matlabcentral/fileexchange/28952-a-benchmark-problem-for-model-based-control-system-tests-001
11. Jha, S., Limaye, R., Seshia, S.: Beaver: Engineering an Efficient SMT Solver for Bit-Vector Arithmetic. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 668–674. Springer, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-02658-4_53
12. Jung, J., Sülflow, A., Wille, R., Drechsler, R.: SWORD v1.0. Tech. rep. (2009); SMTCOMP 2009: System Description

13. Löding, H., Peleska, J.: Timed moore automata: test data generation and model checking. In: Proc. 3rd International Conference on Software Testing, Verification and Validation (ICST 2010). IEEE Computer Society, Los Alamitos (2010)
14. Lu, S., Li, Z., Quin, F., Tan, L., Zhou, P., Zhou, Y.: Bugbench: Benchmarks for evaluating bug detection tools. In: Workshop on the Evaluation of Software Defect Detection Tools (2005)
15. Memon, A.M.: http://www.cs.umd.edu/~atif/newsite/benchmarks.htm
16. Nielsen, B., Skou, A.: Automated test generation from timed automata. International Journal on Software Tools for Technology Transfer (STTT) 5, 59–77 (2003)
17. Peleska, J., Honisch, A., Lapschies, F., Löding, H., Schmid, H., Smuda, P., Vorobev, E., Zahlten, C.: Embedded systems testing benchmark (2011), http://www.mbt-benchmarks.org
18. Peleska, J., Vorobev, E., Lapschies, F.: Automated test case generation with SMT-solving and abstract interpretation. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 298–312. Springer, Heidelberg (2011)
19. Peleska, J., Vorobev, E., Lapschies, F., Zahlten, C.: Automated model-based testing with RT-Tester. Tech. rep. (2011), http://www.informatik.uni-bremen.de/agbs/testingbenchmarks/turn_indicator/tool/rtt-mbt.pdf
20. RTCA, SC-167: Software Considerations in Airborne Systems and Equipment Certification, RTCA/DO-178B. RTCA (1992)
21. Springintveld, J.G., Vaandrager, F.W., D'Argenio, P.R.: Testing timed automata. Theoretical Computer Science 254(1-2), 225–257 (2001)
22. European Committee for Electrotechnical Standardization: EN 50128 – Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems. CENELEC, Brussels (2001)
23. Systems, S.: Enterprise architect 8.0 (2011), http://www.sparxsystems.de
24. Weißleder, S.: Test Models and Coverage Criteria for Automatic Model-Based Test Generation with UML State Machines. Doctoral thesis, Humboldt-University Berlin, Germany (2010)

Adaptive Testing of Deterministic Implementations Specified by Nondeterministic FSMs

Alexandre Petrenko[1] and Nina Yevtushenko[2]

[1] CRIM, Centre de recherche informatique de Montréal, 405 Ogilvy Avenue, Suite 101
Montréal (Québec) H3N 1M3, Canada
petrenko@crim.ca
[2] Tomsk State University, 36 Lenin Street, Tomsk, 634050, Russia
ninayevtushenko@yahoo.com

Abstract. The paper addresses the problem of adaptive testing of a deterministic FSM which models an implementation under test using a nondeterministic FSM as its specification. It elaborates a method for deriving test fragments, combining and executing them in adaptive way such that the implementation passes the test if and only if it is a reduction of the specification. Compared to the existing methods, it uses adaptive test fragments needed to reach as well as to distinguish states.

Keywords: nondeterministic FSM, conformance testing, test generation, adaptive testing.

1 Introduction

There exists a significant body of work devoted to the development of methods for test generation from a given FSM to guarantee the "full" fault coverage. Such coverage of resulting tests means the following. Given a specification machine with n states and a fault domain containing all FSMs with at most m states, $m \geq n$, the full fault coverage is achieved by a so-called m-complete test suite, which detects all faults in any implementation that can be modelled by an FSM in the fault domain. An implementation has a fault if it does not respect a chosen conformance relation, typically trace equivalence or inclusion. To derive m-complete tests the existing methods (with reset operation) use the following three test fragments:

- transfer sequences/strategies to reach states in the specification FSM;
- traversal sequences to extend the transfer sequences; in case of $m = n$ they ensure the transition coverage of the specification and implementation machines and in case of $m > n$ additionally check for the existence of extra states in the implementation machines;
- state identification or distinguishing sequences/strategies to check states reached by prefixes of the above sequences.

In the case of deterministic specifications, several methods are elaborated, such as the W, Wp, HSI, H, and more recently the SPY method [10]. While differing in the types of state identifiers, they require the same traversal set of all input sequences of length

$m - n + 1$ be applied after each state of the specification. However, as the results of [9] show, different traversal sets should be used when the specification has undistinguishable states. Moreover, it is no longer required to reach with transfer sequences each and every state of such specifications.

When the specification can be nondeterministic and a conforming implementation FSM is its reduction, it can have fewer traces than the specification, and not all the states of the specification can be matched with the states of the implementation. The implication for deriving m-complete test suites is that the value of m can even be smaller than that of n. This fact has to be taken into account in determining each of the three test fragments and the way they are composed to yield m-complete test for a nondeterministic FSM. State reachability and distinguishability can be achieved in testing more efficiently using adaptive execution of inputs, where the choice of a next input depends on a current output, as early work of [1, 8] indicates.

The main contribution of this paper is a method for deriving test fragments, combining and executing them in an adaptive manner against a given deterministic implementation FSM with at most m states such that the resulting verdict is pass if and only if the implementation is a reduction of the specification. The method allows to avoid the derivation of preset m-complete tests, as they could be voluminous. At the same time, we prove that the latter is the union of the tests executed for each implementation with at most m states.

The remaining of this paper is organized as follows. Section 2 defines the basic notions for state machines. Section 3 explains how the test fragments for reaching and distinguishing states as well as traversal sets can be derived for a given nondeterministic FSM and defines an m-complete test as an FSM. Section 4 details the method for adaptive testing. The proposed method is illustrated in details using an example. The related work is discussed in Section 5 and Section 6 concludes the paper.

2 General Definitions

A *Finite State Machine* (FSM) $\mathcal{S}$ is a 5-tuple $(S, s_0, I, O, h_\mathcal{S})$, where S is a finite set of states with the initial state s_0; I and O are finite non-empty disjoint sets of inputs and outputs, respectively; $h_\mathcal{S}$ is a transition relation $h_\mathcal{S} \subseteq S \times I \times O \times S$, where a 4-tuple $(s, i, o, s') \in h_\mathcal{S}$ is a transition.

Sometimes we will consider instead of s_0 another state s of $\mathcal{S}$ as the initial state; such FSM $(S, s, I, O, h_\mathcal{S})$ is denoted $\mathcal{S}/s$.

Input sequence $\alpha \in I^*$ is a *defined* input sequence in state s of $\mathcal{S}$ if it labels a sequence of transitions starting in state s. A *trace* of $\mathcal{S}$ in state s is a string of input-output pairs which label a sequence of transitions starting in state s. Let $Tr(\mathcal{S}/s)$ or $Tr_\mathcal{S}(s)$ denote the set of all traces of $\mathcal{S}$ in state s, while $Tr(\mathcal{S})$ or $Tr_\mathcal{S}$ denote the set of traces of $\mathcal{S}$ in the initial state. Given a sequence $\beta \in (IO)^*$, we use $Pref(\beta)$ to denote the set of all prefixes of β which are in the set $(IO)^*$, and let $Pr(\beta)$ denote $Pref(\beta) \setminus \{\varepsilon\}$. Given sequence $\beta \in (IO)^*$, the *input* (*output*) *projection* of β, denoted $\beta_{\downarrow I}$ ($\beta_{\downarrow O}$), is a sequence obtained from β by erasing symbols in O (I).

We define various types of machines as follows.

FSM $\mathcal{S} = (S, s_0, I, O, h_\mathcal{S})$ is

- *trivial* if $h_\mathcal{S} = \emptyset$;
- *completely specified* (a complete FSM) if for each pair $(s, i) \in S \times I$ there exists $(o, s') \in O \times S$ such that $(s, i, o, s') \in h_\mathcal{S}$;
- *partially specified* (a partial FSM) if for some pair $(s, i) \in S \times I$, input i is undefined in state s, i.e., $(s, i, o, s') \notin h_\mathcal{S}$ for all $(o, s') \in O \times S$;
- *deterministic* (DFSM) if for each pair $(s, i) \in S \times I$ there exists at most one transition $(s, i, o, s') \in h_\mathcal{S}$ for some $(o, s') \in O \times S$;
- *nondeterministic* (NFSM) if for some pair $(s, i) \in S \times I$, there exist at least two transitions $(s, i, o_1, s_1), (s, i, o_2, s_2) \in h_\mathcal{S}$, such that $o_1 \neq o_2$ or $s_1 \neq s_2$;
- *observable* if for each two transitions $(s, i, o, s_1), (s, i, o, s_2) \in h_\mathcal{S}$ it holds that $s_1 = s_2$;
- *single-input* if in each state there is at most one defined input, i.e., if for each two transitions $(s, i_1, o_1, s_1), (s, i_2, o_2, s_2) \in h_\mathcal{S}$ it holds that $i_1 = i_2$;
- *output-complete* if for each pair $(s, i) \in S \times I$ such that the input i is defined in the state s, there exists a transition from s with i for every output;
- *acyclic* if $Tr_\mathcal{S}$ is finite.

Given input sequence α defined in state s, let $out_\mathcal{S}(s, \alpha)$ denote the set of output sequences which can be produced by $\mathcal{S}$ in response to α at state s, that is $out_\mathcal{S}(s, \alpha) = \{\beta_{\downarrow O} \mid \beta \in Tr(\mathcal{S}/s) \text{ and } \beta_{\downarrow I} = \alpha\}$. Given an observable FSM $\mathcal{S}$, for a trace $\beta \in Tr(\mathcal{S}/s)$, s-after-β denotes the state reached by $\mathcal{S}$ when it executes the trace β from state s. If s is the initial state s_0 then instead of s_0-after-β we write $\mathcal{S}$-after-β. The FSM $\mathcal{S}$ is *initially connected*, iff for any state $s \in S$ there exists a trace β such that $\mathcal{S}$-after-$\beta = s$. A state is a *deadlock* state if no input is defined in the state and trace $\beta \in Tr(\mathcal{S})$ is a *completed* trace of $\mathcal{S}$ if $\mathcal{S}$-after-β is a deadlock state.

In this paper, we consider only complete initially connected observable specification machines; one could use a standard procedure for automata determinization to convert a given FSM into observable one. We define in terms of traces several relations between states of a complete FSM.

Given states s_1, s_2 of a complete FSM $\mathcal{S} = (S, s_0, I, O, h_\mathcal{S})$,

- s_1 and s_2 are (*trace-*) *equivalent*, if $Tr_\mathcal{S}(s_1) = Tr_\mathcal{S}(s_2)$;
- s_2 is *trace-included* into (is a *reduction* of) s_1, $s_2 \leq s_1$, if $Tr_\mathcal{S}(s_2) \subseteq Tr_\mathcal{S}(s_1)$;
- s_1 and s_2 are *r-compatible*, $s_1 \simeq s_2$, if there exists a state of a complete FSM that is a reduction of both states s_1 and s_2;
- s_1 and s_2 are *r-distinguishable*, $s_1 \not\simeq s_2$ if no state of any complete FSM can be a reduction of both states s_1 and s_2.

We also use relations between machines. Given FSMs $\mathcal{S} = (S, s_0, I, O, h_\mathcal{S})$ and $\mathcal{P} = (P, p_0, I, O, h_\mathcal{P})$, FSM $\mathcal{P}$ is a *reduction* of $\mathcal{S}$ if $Tr_\mathcal{P}(p_0) \subseteq Tr_\mathcal{S}(s_0)$; FSM $\mathcal{P}$ is a *submachine* of $\mathcal{S}$ if $P \subseteq S$, $p_0 = s_0$ and $h_\mathcal{P} \subseteq h_\mathcal{S}$.

To characterize the common behavior of two machines (states) we use the operation of the intersection. The *intersection* $\mathcal{S} \cap \mathcal{P}$ of two machines $\mathcal{S}$ and $\mathcal{P}$ (also known as the product) is an FSM $(Q, q_0, I, O, h_{\mathcal{S}\cap\mathcal{P}})$ with the state set $Q \subseteq S \times P$, the initial state $q_0 = s_0p_0$, and the transition relation $h_{\mathcal{S}\cap\mathcal{P}}$ such that Q is the smallest state set obtained by using the rule $(sp, i, o, s'p') \in h_{\mathcal{S}\cap\mathcal{P}} \Leftrightarrow (s, i, o, s') \in h_{\mathcal{S}}$ & $(p, i, o, p') \in h_{\mathcal{P}}$. The intersection FSM $\mathcal{S} \cap \mathcal{P}$ preserves only common traces of the two machines, in other words, for each state sp of $\mathcal{S} \cap \mathcal{P}$ we have $Tr_{\mathcal{S}\cap\mathcal{P}}(sp) = Tr_{\mathcal{S}}(s) \cap Tr_{\mathcal{P}}(p)$ and thus, $Tr_{\mathcal{S}\cap\mathcal{P}} = Tr_{\mathcal{S}} \cap Tr_{\mathcal{P}}$.

3 Deriving Test Fragments

In this section, we first establish properties of states of a specification NFSM that can be reached in an adaptive way and present a method for deriving FSM models, called state preambles, for adaptive strategies needed to reach such states. Then we consider adaptive distinguishability of states and provide a precise characterization of all possible strategies by an FSM, called a canonical separator, obtained from a self-product of the specification machine. State separators are submachines of a canonical separator. We also explain how traversal sets are derived for a given NFSM and conclude by defining a test as an FSM and its completeness.

3.1 State Preambles

All the existing test generation methods rely on tests which reach the states of the specification and match the states of the implementation and specification machines. Each such test for DFSM is completely defined by an input sequence. Once a specification is an NFSM and an implementation is allowed to have fewer traces than the specification not all the states of the specification can be matched with the states of the implementation. Indeed, a reduction of the specification FSM may have fewer states. This observation leads to the question which state of the specification FSM must be "implemented" in any correct implementation? Intuitively, it is a state such that any reduction of the specification FSM should have a trace which takes the specification FSM from the initial state to the state in question. This intuition is reflected in the following definition.

Definition 1. *Given an FSM $\mathcal{S} = (S, s_0, I, O, h_{\mathcal{S}})$, state $s \in S$ is* definitely reachable *if any reduction of $\mathcal{S}$ has a trace which takes $\mathcal{S}$ into the state s.*

This property can be established as follows.

Proposition 1. *State s of an FSM $\mathcal{S}$ is definitely reachable if and only if $\mathcal{S}$ has a single-input acyclic submachine $\mathcal{S}'$ with the only deadlock state s such that for each input defined in some state of $\mathcal{S}'$, the state has all the transitions of $\mathcal{S}$ labeled with this input.*

Such a submachine can be used in testing to adaptively bring a given machine into a definitely reachable state and is called a preamble for that state.

Definition 2. *Given a definitely reachable state $s \in S$, a single-input acyclic submachine of $\mathcal{S}$ with the only deadlock state s such that for each input defined in some state of the submachine, the state has all the outgoing transitions of $\mathcal{S}$ labeled with this input is a* preamble *for state s, denoted $\mathcal{P}_s = (R, r_0, I, O, h_{\mathcal{P}_s})$.*

A state for which there exists a preamble with a single input projection for all completed traces was called deterministically reachable state in [8]. In a deterministic (initially connected) machine each state is deterministically reachable; any nondeterministic machine has at least one deterministically, thus definitely, reachable, state, namely, the initial state.

We present a method to check whether state s is definitely reachable and, if it is, to derive a preamble $\mathcal{P}_s$.

```
**Algorithm 1.** for constructing a preamble for a given state
**Input:** An FSM $\mathcal{S}$ and $s \in S$, $s \neq s_0$.
**Output:** a preamble if the state $s$ is definitely reachable.
Construct an FSM ($R$, $r_0$, $I$, $O$, $h_{\mathcal{R}_s}$) as follows
$R := \{s\}$;
$h_{\mathcal{R}_s} := \varnothing$;
While there exist a state $s' \notin R$ and a set of inputs $I_{s'}$, such that for each input $i \in I_{s'}$, $(s', i, o, s'') \in h_{\mathcal{S}}$, $s'' \in R$ for all $o \in out_{\mathcal{S}}(s', i)$
        $R := R \cup \{s'\}$;
        $h_{\mathcal{R}_s} := h_{\mathcal{R}_s} \cup \{(s', i, o, s'') \mid i \in I_{s'}$ and $o \in out_{\mathcal{S}}(s', i)\}$;
End While;
If $s_0 \notin R$ then return the message "the state $s$ is not definitely reachable" and stop
Else let ($R$, $r_0$, $I$, $O$, $h_{\mathcal{R}_s}$), where $r_0 := s_0$, be the obtained FSM;
Starting from the initial state, remove from each state with several defined inputs all outgoing transitions with the same input until each state has a single defined input thus to obtain a single-input submachine with the only deadlock state $s$;
Delete states which are unreachable from the initial state;
Return the obtained machine as a preamble for the state $s$ and stop. ♦
```

The idea is that analyzing the backward reachability, we first choose all inputs which may form a preamble, since at that stage we do not know those leading to the required state from the initial state and then analyzing the forward reachability, we retain just a single input at each state. Since any preamble is a submachine of the specification machine with n states, then the length of any trace in a preamble never exceeds the number of states of the specification; in other words, one needs at most n - 1 inputs to transfer to a definitely reachable state.

Given a preamble $\mathcal{P}_s$, let $T(\mathcal{P}_s)$ be the set of its completed traces. Given a set K of definitely reachable states of the specification FSM $\mathcal{S}$ such that the initial state $s_0 \in K$, the union U_K of all the completed traces over all the preambles of these states is a *cover* for K. The cover will be used in constructing another test fragment, traversal sets, as explained in Section 3.3.

Example. Consider the FSM $\mathcal{S}$ in Figure 1(a). It has four states 1, 2, 3, and 4 with state 1 as the initial state; three inputs a, b, and c; and two outputs 0 and 1.

The initial state is deterministically reachable, it is reached with the empty input sequence, so its preamble is a trivial FSM. Each state is definitely reachable, thus the set K contains all the states. Figures 2 and 3 present preambles and intermediate machines constructed using the above given method. A cover for all the states is the union of all completed traces of the obtained preambles, $U_K = \{\varepsilon, a1, a0c1b1, c0, c1a1, c1, c0c1\}$. ♦

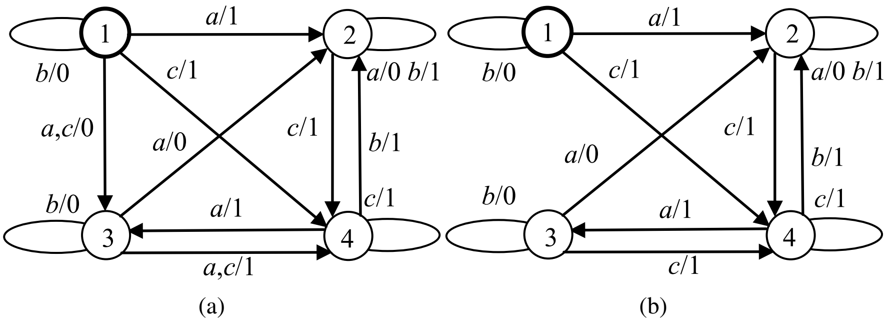


Fig. 1. (a) FSM $\mathcal{S}$ and (b) FSM $\mathcal{B}$ (initial states are in bold)

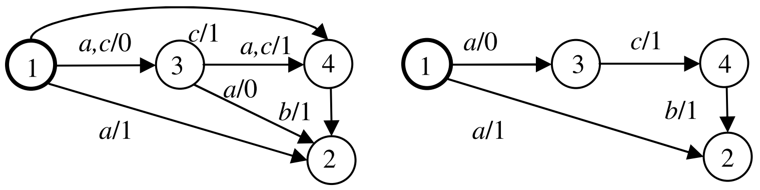


Fig. 2. Constructing a preamble for state 2

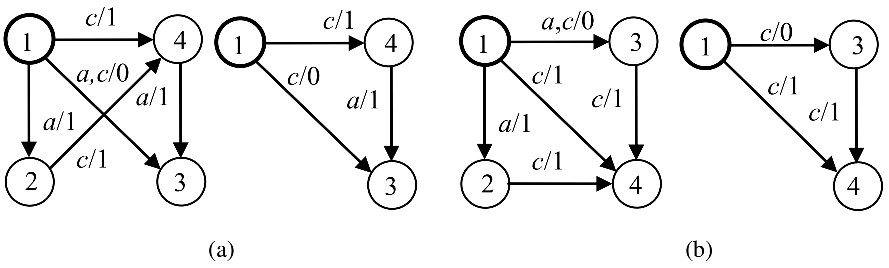


Fig. 3. Constructing preambles for state 3 (a) and state 4 (b)

3.2 State Separators

By the definition, if two states of a given machine are r-compatible, there exists a state of some complete FSM which is a reduction of both states. Such a state is the initial state of the intersection of two instances of a given machine initialized in different states (a self-product), since the intersection represents all the common traces of the two states. On the other hand, if the two states are *r*-distinguishable, the intersection is not a complete FSM. This fact is stated in the following.

Proposition 2. *Given two states s_1 and s_2 of a complete FSM $\mathcal{S} = (S, s_0, I, O, h_{\mathcal{S}})$ and the intersection $\mathcal{S}/s_1 \cap \mathcal{S}/s_2 = (Q, s_1s_2, I, O, h_{\mathcal{S}/s_1 \cap \mathcal{S}/s_2})$, states s_1 and s_2 are r-compatible if and only if the intersection has a complete submachine.*

Corollary 1. *States s_1 and s_2 are r-distinguishable if and only if the intersection has no complete submachine, i.e., each submachine has an input undefined in some state.*

The existence of a complete submachine can be checked by iterative removal from the intersection each state that has undefined input along with its incoming transitions. If at the end, the initial state is also removed then the two given states are *r*-distinguishable, otherwise they are *r*-compatible. The procedure is similar to the one for checking the existence of an adaptive (s_1, s_2)-distinguishing strategy considered in [1]. While that work focuses on the existence of such a strategy, it does not provide a means to characterize all the strategies and a method to obtain one. Such a method was first elaborated in [8], and now we offer an exact characterization of all state distinguishing strategies in the form of an FSM, called a canonical separator, which is obtained from the intersection as follows.

Definition 3. *Given r-distinguishable states s_1 and s_2 of an FSM $\mathcal{S}$ and the intersection $\mathcal{S}/s_1 \cap \mathcal{S}/s_2 = (Q, s_1s_2, I, O, h_{\mathcal{S}/s_1 \cap \mathcal{S}/s_2})$, an FSM $\mathcal{P} = (P, s_1s_2, I, O, h_{\mathcal{P}})$ such that $P = Q \cup \{s_1, s_2\}$ and $h_{\mathcal{P}} = h_{\mathcal{S}/s_1 \cap \mathcal{S}/s_2} \cup \{(ss', i, o, s_1) \mid ss' \in Q, o \in out(s, i) \setminus out(s', i)\} \cup \{(ss', i, o, s_2) \mid ss' \in Q, o \in out(s', i) \setminus out(s, i)\}$, is a* canonical separator *of states s_1 and s_2.*

In other words, a canonical separator for two *r*-distinguishable states s_1 and s_2 of $\mathcal{S}$ is the intersection extended by including two designated deadlock states s_1 and s_2 such that the completed traces distinguish the two possible initial states of $\mathcal{S}$.

A canonical separator contains all the traces which separate *r*-distinguishable states. For testing, it is sufficient to use acyclic traces which do not branch on inputs. This leads to the following definition.

Definition 4. *Given r-distinguishable states s_1 and s_2 of an FSM $\mathcal{S}$, a single-input acyclic submachine of the canonical separator, such that the only deadlock states are s_1 and s_2 and for each input defined in some state of the submachine, the state has all the outgoing transitions of the canonical separator labeled with this input is a* separator *of states s_1 and s_2, denoted $\mathcal{R}(s_1, s_2)$.*

By the definition, canonical separator for given states is unique, but it may contain several separators as its submachines. The procedure for determining a separator from a given canonical one is similar to Algorithm 1; it includes backward analysis and iterative removal of all defined inputs, but one for each state, as well as cycles such that the deadlock states of the submachine are reachable from all the other states and is omitted here.

Example. Figure 4(a) shows a fragment of the canonical separator of states 1 and 3 for the FSM $\mathcal{S}$ in Figure 1 (we do not show the part which starts in state 44, as states 1 and 3 cannot be reached from it). Figure 4(b) shows a separator obtained from the canonical one. The separators for other states are shown in Figure 4(c). ♦

Notice that the intersection $S/s_1 \cap S/s_2$ has no more than n^2 states, if S is an observable machine with n states. Then the length of any trace in a separator never exceeds this bound; in other words, one needs at most n^2 inputs to adaptively distinguish two states.

We can use separators of pairs of states to identify a given state among a set of possible states. Given a subset of states $L \subseteq S$, let $Id(s, L) = \{R(s, s') \mid s' \in L \text{ and } s' \not\simeq s\}$. The set $Id(s, L)$ is in some sense a generalization of a concept of a state identifier in a subset of states used in testing from deterministic FSMs. More precisely, the set $Id(s, L)$ allows one to identify that a given state cannot be a reduction of any state in the set L but state s. Such sets will be used in the main algorithm in Section 4.

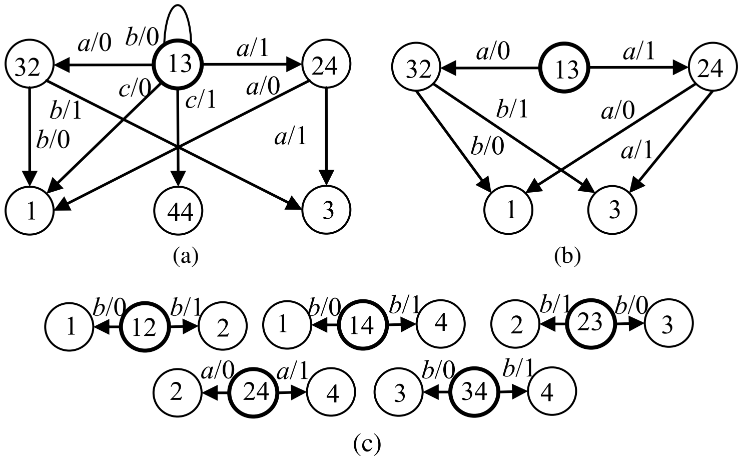


Fig. 4. (a) a fragment of the canonical separator of states 1 and 3, (b) the separator of states 1 and 3, (c) the separators for other states

3.3 Traversal Sets

The construction of traversal sets is based on the approach elaborated in [6] and we refer the reader to that work for more detail. The basic idea is to count states of the specification FSM traversed by a trace and to terminate the trace as soon it becomes cyclic in any conforming implementation FSM with at most m states. The termination rule is formulated in terms of partial orders on the traces, defined by the reduction relation between the states. The improvement compared to [6] is the use of definitely reachable states instead of deterministically reachable states which shortens the traversal sets.

Given a specification FSM S, a cover U_K of the set K of all definitely reachable states, states $s, s' \in K$, a preamble P_s, and a non-empty trace $\beta \in Tr_S(s)$, we define a (strict) partial order $\leq_{s'}$ on the set $T(P_s)Pr(\beta) \cup U_K$, where $T(P_s)Pr(\beta) = \{\alpha\gamma \mid \alpha \in T(P_s) \ \& \ \gamma \in Pr(\beta)\}$, such that

1) for $\omega, \omega' \in \alpha Pr(\beta)$, $\alpha \in T(P_s)$, $\omega \leq_{s'} \omega'$ if $|\omega| < |\omega'|$ and S-after-$\omega \leq S$-after-$\omega' \leq s'$; and
2) for $\omega \in U_K$, $\omega' \in T(P_s)Pr(\beta)$, $\omega \leq_{s'} \omega'$ if $\omega \neq \omega'$ and S-after-$\omega \leq S$-after-$\omega' \leq s'$.

Let $C(T(\mathcal{P}_s)Pr(\beta), U_K, s')$ be a longest chain of the poset $(T(\mathcal{P}_s)Pr(\beta) \cup U_K, \leq_{s'})$, and $|C(T(\mathcal{P}_s)Pr(\beta), U_K, s')|$ be its length. Given a state $s \in K$, trace $\beta \in Tr_{\mathcal{S}}(s)$ is a *traversal* trace for the preamble $\mathcal{P}_s$ if $\Sigma_{s' \in R}|C(T(\mathcal{P}_s)Pr(\beta), U_K, s')| = m + 1$ for some set $R \in \mathcal{R}_{\mathcal{S}}$, where $\mathcal{R}_{\mathcal{S}}$ denotes the set of all maximal sets of pairwise r-distinguishable states of $\mathcal{S}$. For each traversal trace β we select one among such sets R in $\mathcal{R}_{\mathcal{S}}$ and denote it R_β.

The set of all possible traversal traces for the preamble $\mathcal{P}_s$ is a *traversal set* $N(U_K, \mathcal{P}_s)$. We illustrate the construction of traversal sets using our running example.

Example. We assume that any implementation machine has at most four states, i.e., $m = 4$. All the states of $\mathcal{S}$ are pairwise r-distinguishable, $\mathcal{R}_{\mathcal{S}} = \{\{1, 2, 3, 4\}\}$.

Consider state 1. The set of all completed traces of $\mathcal{P}_1$ contains just the empty word ε. To determine the traversal set for state 1, we start by considering all the traces of length one of this state and iteratively increasing their length until the above condition is satisfied. We have initially the traces $a0$, $a1$, $b0$, $c0$, $c1$ of state 1. For state $s = 1$ and trace $\beta = a0$, we construct the set $T(\mathcal{P}_s)Pr(\beta) \cup U_K$, where $T(\mathcal{P}_s)Pr(\beta) = \{a0\}$, $U_K = \{\varepsilon, a1, a0c1b1, c0, c1a1, c1, c0c1\}$, thus $T(\mathcal{P}_s)Pr(\beta) \cup U_K = \{\varepsilon, a0, a1, a0c1b1, c0, c1a1, c1, c0c1\}$. The longest chain $C(T(\mathcal{P}_s)Pr(\beta), U_K, 1)$ of the poset $(T(\mathcal{P}_s)Pr(\beta) \cup U_K, \leq_1)$, is $\{\varepsilon\}$, a longest chain $C(T(\mathcal{P}_s)Pr(\beta), U_K, 2)$ of the poset $(T(\mathcal{P}_s)Pr(\beta) \cup U_K, \leq_2)$ is a singleton, so is one for $C(T(\mathcal{P}_s)Pr(\beta), U_K, 4)$. A longest chain $C(T(\mathcal{P}_s)Pr(\beta), U_K, 3)$ is $\{a0, c0\}$, since $\mathcal{S}$-after-$a0$ = $\mathcal{S}$-after-$c0$ = 2, thus, $|C(T(\mathcal{P}_s)Pr(\beta), U_K, 3)| = 2$. For the set $R = \{1, 2, 3, 4\}$, we obtain $\Sigma_{s' \in R}|C(T(\mathcal{P}_s)Pr(\beta), U_K, s')| = 1 + 2 + 1 + 1 = 5$. Therefore, the trace $a0$ is a traversal trace for the preamble $\mathcal{P}_1$. Similarly, we conclude that the remaining traces of length one, $b0$, $c0$, $c1$ are also traversal traces for state 1. Thus, for the preamble $\mathcal{P}_1$, the traversal set $N(U_K, \mathcal{P}_1)$ becomes $\{a0, a1, b0, c0, c1\}$.

The traversal sets for the remaining preambles are constructed as above. $N(U_K, \mathcal{P}_2) = \{a0, b1, c1\}$, $N(U_K, \mathcal{P}_3) = \{a0, a1, b0, c1\}$ and $N(U_K, \mathcal{P}_4) = \{a1, b1, c1\}$.

Given a traversal trace $\beta \in N(U_K, \mathcal{P}_s)$, $s' \in R_\beta$, the set $Id(s', R_\beta)$ is used to identify that a given state cannot be a reduction of any state of the set R_β but state s'. In our example, we have that $Id(s, S) = Id(s, R_\beta)$ for each state s, since $R_\beta = \{1, 2, 3, 4\}$ for each trace β in the traversal sets obtained above. ♦

3.4 FSM Tests

We use FSMs to model tests from an implementation under test perspective: inputs (outputs) of the specification machine are also inputs (outputs) of a test. Thus, a tester, executing the test, applies inputs to an implementation FSM, observes its outputs and produces a corresponding verdict defined by a final state, *pass* or *fail*, reached by the test.

Definition 5. *An acyclic output-complete FSM $\mathcal{U} = (U \cup \{pass, fail\}, u_0, I, O, h_{\mathcal{U}})$, where pass and fail are designated deadlock states, is a* test *if $(u, i, o, fail) \in h_{\mathcal{U}}$ implies that u-after-$io'\beta = pass$ for some $o' \neq o$ and $\beta \in Tr_{\mathcal{U}}(u\text{-after-}io')$.*

Given a test $\mathcal{U}$, we further refer to traces which take $\mathcal{U}$ from the initial state to the *fail* state as *fail* traces and denote $Tr_{\mathcal{U}}^{\text{fail}}$ the set of all fail traces. *Pass* traces are defined as follows, $Tr_{\mathcal{U}}^{\text{pass}} = Tr_{\mathcal{U}} \setminus Tr_{\mathcal{U}}^{\text{fail}}$. Note that while fail traces are completed traces, pass traces can be proper prefixes of other traces. Test $\mathcal{U}$ is a *trivial* test if it is a trivial FSM with the initial *pass* state.

A test may have transitions with different inputs from a same state. In several work, this is not allowed in order to ensure the controllability of test execution. We leave the choice of inputs to the tester; assuming the following about the tester. If in a current state of the test, several inputs are defined, then executing such a test the tester simply selects one among alternative inputs during a particular test run. To execute another input defined in this state, the tester first uses a reset operation assumed to be available in any implementation to reset it to its initial state and then re-executes the preamble to this state. Test execution continues until no more unexecuted inputs in the test are left. Moreover, test execution is *adaptive*: depending on the observed output reaction to the previous input, the tester either chooses a next input to execute or just terminates the test run when an unexpected output is observed and it reaches the state *fail*.

To characterize conformance in this paper, we use the reduction relation and assume that the specification can be nondeterministic while implementations are deterministic, but both are complete machines. Given a complete FSM $\mathcal{S} = (S, s_0, I, O, h_{\mathcal{S}})$, let $\Im(\mathcal{S})$ be a set of complete deterministic (implementation) machines over the input alphabet I and the output alphabet O, called a *fault domain*. FSM $\mathcal{B} \in \Im(\mathcal{S})$ is a *conforming* implementation machine of $\mathcal{S}$ w.r.t. the reduction relation if $\mathcal{B} \leq \mathcal{S}$.

Definition 6. *Given the specification FSM $\mathcal{S}$, a test $\mathcal{U} = (U, u_0, I, O, h_{\mathcal{U}})$, and an implementation FSM $\mathcal{B} \in \Im(\mathcal{S})$,*

- $\mathcal{B}$ passes *the test $\mathcal{U}$, if the intersection $\mathcal{B} \cap \mathcal{U}$ has no state, where the test $\mathcal{U}$ is in the state fail. The test $\mathcal{U}$ is* sound *for FSM $\mathcal{S}$ in $\Im(\mathcal{S})$ w.r.t. the reduction relation, if any $\mathcal{B} \in \Im(\mathcal{S})$, which is a reduction of $\mathcal{S}$, passes the test $\mathcal{U}$.*
- $\mathcal{B}$ fails *$\mathcal{U}$ if the intersection $\mathcal{B} \cap \mathcal{U}$ has a state, where the test $\mathcal{U}$ is in the state fail. The test $\mathcal{U}$ is* exhaustive *for FSM $\mathcal{S}$ in $\Im(\mathcal{S})$ w.r.t. the reduction relation, if any $\mathcal{B} \in \Im(\mathcal{S})$, which is not a reduction of $\mathcal{S}$, fails the test $\mathcal{U}$.*
- *The test $\mathcal{U}$ is* complete *for FSM $\mathcal{S}$ in $\Im(\mathcal{S})$ w.r.t. the reduction relation, if it is sound and exhaustive in $\Im(\mathcal{S})$ w.r.t. the reduction relation.*

The set $\Im(\mathcal{S})$ that contains all complete deterministic FSMs with at most m states is denoted $\Im_m(\mathcal{S})$. A test is *m-complete* if it is complete in the fault domain $\Im_m(\mathcal{S})$.

4 Adaptive Testing

In this section, we propose an algorithm for adaptive testing of a complete deterministic implementation FSM $\mathcal{B}$ with at most m states; the algorithm yields the verdict pass if $\mathcal{B}$ is a reduction of a given specification FSM $\mathcal{S}$ and verdict fail if it is not a reduction of $\mathcal{S}$.

Test fragments defined in previous sections are used in the proposed method as in all the existing methods; namely, we reach states with preambles, apply traversal sets after reached states and check states with separators. However, preambles have to be executed in an adaptive way to reach states of the implementation FSM which match definitely reachable states of the specification machine (and only they, as opposed to methods for DFSMs), while the execution of separators is in general also adaptive. The key differences are related to traversal sets. First, since a conforming FSM may not implement all the traversal traces, we need to determine those present in a given implementation FSM. This is achieved by executing the input projections of the traversal traces. Second, learning these traces during test execution allows determining separators to check r-distinguishable states, traversed by them.

The algorithm includes also the construction of an FSM which represents all the traces observed during test execution. This machine is then used to show that an FSM which contains the observed traces for all implementation FSMs in $\Im_m(\mathcal{S})$ is an m-complete test for the FSM $\mathcal{S}$.

Algorithm 2. for adaptive testing of a deterministic implementation FSM

Input. Complete FSM $\mathcal{S}$, a set K of definitely reachable states, the set $Id(s, S)$ and preamble $\mathcal{P}_s$ for each $s \in K$, traversal sets $N(U_K, \mathcal{P}_s)$, sets $Id(s', R_\beta)$ for each $s' \in R_\beta$ and $\beta \in N(U_K, \mathcal{P}_s)$, and an implementation (black box) which behaves as a deterministic FSM $\mathcal{B}$.

Output. Verdict pass if $\mathcal{B}$ is a reduction of $\mathcal{S}$ or verdict fail if $\mathcal{B}$ is not a reduction of $\mathcal{S}$ and the FSM $\mathcal{G}_{\mathcal{B}}$ that contains all the observed traces of $\mathcal{B}$.

```
Initialize the two sets $T^{pass}$ and $T^{fail}$ of traces as the empty sets;
While there exists an unexecuted separator $\mathcal{R}(s, s')$ in the set $Id(s, S)$ for some $s \in K$
    Apply reset;
    Execute the preamble $\mathcal{P}_s$ until the observed trace is not in $\mathcal{P}_s$ or the designated
    state $s$ of the preamble is reached, let $\alpha$ be the observed trace;
      If the trace $\alpha$ is not in $\mathcal{P}_s$ add $\alpha$ to the set $T^{fail}$ and terminate with the verdict fail;
      Otherwise, let the observed completed trace of $\mathcal{P}_s$ be $\alpha_s$; add the trace $\alpha_s$ to $T^{pass}$;
      Mark "executed" all the separators in $Id(s, S)$ which have the same set of
      traces to the state $s$ as $\mathcal{R}(s, s')$ and execute the separator $\mathcal{R}(s, s')$, let $\beta$ be the
      observed trace;
          If the designated state $s$ of the separator is not reached add the trace $\alpha_s\beta$
          to $T^{fail}$ and terminate with the verdict fail;
          Otherwise, add the trace $\alpha_s\beta$ to $T^{pass}$;
End While;
While there exists an unexplored state $s \in K$
    While there exists an unexecuted trace in the traversal set $N(U_K, \mathcal{P}_s)$, let $\gamma$ be
    the input projection of a longest unexecuted traversal trace
      Apply reset;
      Execute the preamble $\mathcal{P}_s$ (and observe the completed trace $\alpha_s$ of $\mathcal{P}_s$);
```

```
            Apply the inputs of $\gamma$ one by one until the observed trace is not in $Tr(S/s)$ or the trace is in $N(U_K, P_s)$; let $\beta$ be the observed trace with the input projection $\nu$;
                If the trace $\beta \notin Tr(S/s)$ add the trace $\alpha_s\beta$ to $T^{fail}$ and terminate with the verdict fail;
                Otherwise, i.e., if $\beta \in N(U_K, P_s)$ then mark "executed" each trace in $N(U_K, P_s)$ whose input projection has the prefix $\nu$;
                While for some prefix $\sigma$ of $\beta$ such that $\alpha_s\sigma \in C(T(P_s)Pr(\beta), U_K, s')$, $s' \in R_\beta$, there exists an unexecuted separator $R(s', s'')$ in the set $Id(s', R_\beta)$
                    Apply reset;
                    Execute the preamble $P_s$ (and observe the completed trace $\alpha_s$ of $P_s$);
                    Apply the input projection of $\sigma$, let $\eta$ be the observed trace;
                    Add to $T^{pass}$ the observed trace $\alpha_s\eta$;
                    Mark "executed" all the separators in $Id(s', R_\beta)$ which have the same set of traces to the designated state $s'$ as the unexecuted separator $R(s', s'')$ and execute the separator, let $\kappa$ be the observed trace;
                    If the designated state $s'$ of the separator is not reached then add the trace $\alpha_s\eta\kappa$ to $T^{fail}$ and terminate with the verdict fail;
                    Otherwise, add the trace $\alpha_s\eta\kappa$ to $T^{pass}$;
                End While;
        End While;
    Mark the state $s$ "explored";
    End While;
    Terminate with the verdict pass;
Derive an FSM $G_B$ with the set of traces $pref(T^{pass}) \cup T^{fail}$, such that each completed trace in $T^{pass}$ takes the FSM $G_B$ to the deadlock state pass and each trace in $T^{fail}$ takes $G_B$ to the deadlock state fail. ♦
```

Theorem 1. *Given a deterministic FSM $B = (B, b_0, I, O, h_B)$ with at most m states, let Algorithm 2 be used for the adaptive testing of FSM B against a given specification FSM S. Then the verdict pass is produced if B is a reduction of S while the verdict fail is produced if B is not a reduction of S.*

Before proving Theorem 1 we first illustrate Algorithm 2 with our running example.

Example. Assume we are given the deterministic implementation FSM B in Figure 1(b). For simplicity the FSM B has the state labels as in the FSM S; it is then a submachine of that FSM, differing in the transitions (1, *a*, 0, 3) and (3, *a*, 1, 4) of S which are absent in the FSM B.

We initialize the two sets T^{pass} and T^{fail} as the empty sets and consider state 1 in the set K. The preamble P_1 is a trivial FSM, so we just add the trace ε to T^{pass}. $Id(1, S)$ includes the separators: $R(1, 2)$, $R(1, 3)$, and $R(1, 4)$ which we execute by applying first input *b* to the implementation machine, observing the completed trace *b*0 and then, after reset, input *a* which results in output 1, followed by input *a* observing the completed trace *a*1*a*0. We add the traces *b*0 and *a*1*a*0 to T^{pass}. Next, we execute the preamble $\mathcal{P}_2$ and observe the completed trace $a1$ of this preamble, which we add to T^{pass}. Notice that from now on, executing the preamble $\mathcal{P}_2$ reduces to applying input a and observing trace $a1$. To execute the separators in $Id(2, S)$, we apply input a followed by b, observing the trace $a1b1$; finally a followed by a, observing the trace $a1a0$, which we add to T^{pass}. Executing the preamble $\mathcal{P}_3$, we obtain the trace $c1a1$ and add it to T^{pass}. After the subsequent execution of three separators, we add to T^{pass} the traces $c1a1a0b1$ and $c1a1b0$. Executing the preamble $\mathcal{P}_4$ we obtain the trace $c1$ and add it to T^{pass}. After the subsequent execution of the three separators, we add to T^{pass} the traces $c1a1$ and $c1b1$. Completing the execution of the preambles along with the separators, we obtain the following pass traces $T^{pass} = \{ \varepsilon, b0, a1a0, a1, a1b1, c1a1, c1a1a0b1, c1a1b0, c1b1\}$.

Next, we have to execute traversal sets after their corresponding preambles followed by separators. Consider state 1 and $N(U_K, \mathcal{P}_1) = \{a0, a1, b0, c0, c1\}$. We apply a and observe the trace $a1$; executing the separators in $Id(2, S)$ we observe the traces which are already in T^{pass}, namely, $a1b1$ and $a1a0$. We apply b and observe the trace $b0$; executing the separators in $Id(1, S)$ we observe the traces $b0b0$ and $b0a1a0$. Finally we apply c and observe the trace $c1$; executing the separators in $Id(4, S)$ we observe the traces which are already in T^{pass}, namely, $c1a1$ and $c1b1$.

Consider state 2 and $N(U_K, \mathcal{P}_2) = \{a0, b1, c1\}$. We obtain the following observations: $a1a0a0$, $a1a0b1$, $a1b1a0$, $a1b1b1$, $a1c1a1$, and $a1c1b1$. For state 3 and $N(U_K, \mathcal{P}_3) = \{a0, a1, b0, c1\}$, we have $c1a1a0a0$, $c1a1a0b1$, $c1a1b0a0b1$, $c1a1b0b0$, $c1a1c1a1$, $c1a1c1b1$. Finally, for state 4 and $N(U_K, \mathcal{P}_4) = \{a1, b1, c1\}$ we obtain the following observations: $c1a1a0b1$, $c1a1b0$, $c1b1a0$, $c1b1b1$, $c1c1a1$, $c1c1b1$.

The resulting set of completed pass traces becomes $\{a1a0a0, a1a0b1, a1b1a0, a1b1b1, a1c1a1, a1c1b1, b0a1a0, b0b0, c1a1a0a0, c1a1a0b1, c1a1b0a0b1, c1a1b0b0, c1a1c1a1, c1a1c1b1, c1b1a0, c1b1b1, c1c1a1, c1c1b1\}$. The set of fail traces T^{fail} is empty, since the FSM $\mathcal{B}$ is a reduction of the FSM $\mathcal{S}$. The algorithm terminates with the verdict pass. The obtained acyclic FSM $\mathcal{G}_\mathcal{B}$ has the set of traces $Pref(T^{pass})$ and each completed trace takes the FSM $\mathcal{G}_\mathcal{B}$ to the deadlock state *pass*. ♦

Compare now the obtained result with a preset test suite returned by the method from [6]. That method constructs traversal sets only for deterministically reachable states. As a result the traces in these traversal sets are longer since each deterministically reachable state is also definitely reachable, but the converse is not true. Since there are only two deterministically reachable state in the specification FSM in Figure 1, the test suite will contain all the input sequences of length three appended with corresponding separators, i.e., the total length of a test suite is more than $5 \cdot 3^3$. Here we notice that the method for constructing a complete test suite in [2] also uses only deterministically reachable states.

Now we return to the proof of the theorem.

Proof of Theorem 1. By construction, if $\mathcal{B}$ is a reduction of the specification FSM $\mathcal{S}$, then only the verdict pass can be produced by Algorithm 2. In fact, according to the algorithm, each observed trace is a trace of $\mathcal{S}$ and is a pass trace of the test, so the verdict fail cannot be produced.

Consider now FSM $\mathcal{B} \in \Im_m(\mathcal{S})$ that is not a reduction of $\mathcal{S}$ and for an observed completed trace α_s of the preamble $\mathcal{P}_s$, $s \in K$, and for each observed trace $\alpha_s\sigma \in C(T(\mathcal{P}_s)Pr(\beta), U_K, s')$, $\beta \in N(U_K, \mathcal{P}_s)$, $s' \in R_\beta$, no verdict fail was produced. We now show that in this case, there exist $s \in K$ and $\beta \in N(U_K, \mathcal{P}_s) \cap Tr(\mathcal{B}\text{-after-}\alpha_s)$ such that for the observed completed trace α_s of the preamble $\mathcal{P}_s$, there exists a set of sequences $M = \{\mu_1, ..., \mu_{m+1}\} \subseteq \cup_{s' \in R_\beta} C(T(\mathcal{P}_s)Pr(\beta), U_K, s')\}$ with the following property. The set $\{(\mathcal{S} \cap \mathcal{B})\text{-after-}\mu_j \mid j = 1, \ldots, m+1\}$ contains states (s', b) and (s'', b) such that $s' \leq s_1$ and $s'' \leq s_2$ for some states $s_1, s_2 \in R_\beta$.

Let Q' be the set of states that are reachable in the intersection $\mathcal{S} \cap \mathcal{B}$ via observed completed traces of all preambles in the cover U_K, i.e., $Q' = \{(\mathcal{S} \cap \mathcal{B})\text{-after-}\alpha \mid \alpha \in U_K\}$. Let also ν be a shortest trace from a state of the set Q' to a state $q = sb$ such that some input i is not defined in state q under i (Corollary 1), i.e., the output of $\mathcal{B}$ at state b under input i is not in the set of outputs at state s of the specification FSM $\mathcal{S}$. The property of ν being a shortest such trace means that for each trace $\rho\delta \in Tr_{\mathcal{S}}$, where ρ is a completed trace of a preamble $\mathcal{P}_{s'}$, $s' \in K$, and δ possessing the same property, i.e., such that the state $(\mathcal{S} \cap \mathcal{B})\text{-after-}\rho\delta$ has an undefined input, it holds that $|\delta| \geq |\nu|$. Since $\mathcal{B}$ is not a reduction of $\mathcal{S}$, the intersection $\mathcal{S} \cap \mathcal{B}$ is initially connected, and the set K has the initial state, such a trace ν exists. By definition of traversal sets $N(U_K, \mathcal{P}_s)$ and since no verdict fail was produced for observed traces of FSM $\mathcal{B}$, there exist a preamble $\mathcal{P}_s$ and an observed trace $\alpha_s\beta$, where $\alpha_s \in T(\mathcal{P}_s)$, and β is a prefix of ν, with the property that there exists $R_\beta \in \mathrm{R}_{\mathcal{S}}$, such that $\mathcal{S}\text{-after-}\alpha_s\beta \in R_\beta$ and $\Sigma_{s' \in R_\beta}|C(T(\mathcal{P}_s)Pr(\beta), U_K, s')| = m + 1$.

For each state $s' \in R_\beta$, consider a longest chain of the poset $(\alpha_s Pr(\beta) \cup U_K, \leq_{s'})$; let $M = \{\mu_1, \ldots, \mu_k, \mu_{k+1}, ..., \mu_{m+1}\}$. Without loss of generality, we assume that $\mu_1, \ldots, \mu_k$ are the observed completed traces of the set U_K while $\mu_{k+1}, ..., \mu_{m+1}$ are sequences of the set $\alpha_s Pr(\beta)$.

Consider the corresponding $m + 1$ states of FSM $\mathcal{B}$, $\mathcal{B}\text{-after-}\mu_1, \ldots, \mathcal{B}\text{-after-}\mu_{m+1}$. Since $\mathcal{B}$ has at most m states there exist $1 \leq j < r \leq m + 1$ such that $\mathcal{B}\text{-after-}\mu_j = \mathcal{B}\text{-after-}\mu_r$. By the definition of the poset, either $\mathcal{S}\text{-after-}\mu_j \leq \mathcal{S}\text{-after-}\mu_r$ or $\mathcal{S}\text{-after-}\mu_j \not\leq \mathcal{S}\text{-after-}\mu_r$. Let $\mathcal{S}\text{-after-}\mu_j \leq \mathcal{S}\text{-after-}\mu_r$. By definition of the poset $(\alpha_s Pr(\beta) \cup U_K, \leq_{s'})$, two cases are possible, either $\mu_j \in U_K$ and $\mu_r \in \alpha_s Pr(\beta)$ or $\mu_j, \mu_r \in \alpha_s Pr(\beta)$.

Consider the case when $\mu_j \in U_K$ and $\mu_r \in \alpha_s Pr(\beta)$. As Algorithm 2 produces the verdict pass when the implementation $\mathcal{B}$ is tested, the following holds. The trace β' obtained from $\alpha_s\beta$ by deleting the prefix μ_r, is a trace of FSM $\mathcal{B}$ at state $\mathcal{B}\text{-after-}\mu_r = \mathcal{B}\text{-after-}\mu_j$. If β' is not a trace of FSM $\mathcal{S}$ at state $\mathcal{S}\text{-after-}\mu_j$ then a proper prefix β'' of the trace β' takes the FSM $\mathcal{S} \cap \mathcal{B}$ from state $(\mathcal{S}\text{-after-}\mu_j, \mathcal{B}\text{-after-}\mu_j)$ to a state $(\mathcal{S}\text{-after-}\mu_j\beta'', \mathcal{B}\text{-after-}\mu_j\beta'')$ such that the state $(\mathcal{S}\text{-after-}\mu_j\beta'', \mathcal{B}\text{-after-}\mu_j\beta'')$ of the intersection $\mathcal{S} \cap \mathcal{B}$ has an undefined input. If β' is a trace of FSM $\mathcal{S}$ at state $\mathcal{S}\text{-after-}\mu_j$ then β' is a trace of $\mathcal{S} \cap \mathcal{B}$ at state $(\mathcal{S} \cap \mathcal{B})\text{-after-}\mu_j$ and thus, a shorter trace ν_r obtained from ν by deleting the prefix μ_r is a trace of FSM $\mathcal{S} \cap \mathcal{B}$ to a state $(\mathcal{S}\text{-after-}\mu_j\nu_r, \mathcal{B}\text{-after-}\mu_j\nu_r)$ with an undefined input.

Consider now the case when μ_j, $\mu_r \in \alpha_s Pr(\beta)$. Similar to the previous case, the trace β could also be shortened by deleting the part between two states $\mathcal{S}$-after-μ_j and $\mathcal{S}$-after-μ_r, as $\mathcal{S}$-after-$\mu_j \leq \mathcal{S}$-after-μ_r.

Both cases contradict the fact that the trace ν that contains β as a prefix is a shortest trace from a state of the set Q' to a state $q = sb$ with an undefined input i.

Thus, $\mathcal{S}$-after-$\mu_j \not\simeq \mathcal{S}$-after-$\mu_r$, i.e., the set $\{(\mathcal{S} \cap \mathcal{B})\text{-after-}\gamma \mid \gamma \in M\}$ contains states (s, b) and (s', b) such that $s \leq s_1$ and $s' \leq s_2$ for some states s_1, $s_2 \in R_\beta$, thus $s \not\simeq s'$. According to Algorithm 2, for the observed completed trace α_s of the preamble $\mathcal{P}_s$, $s \in K$, after each $\gamma \in C(T(\mathcal{P}_s)Pr(\beta), U_K, s')$, $s' \in R_\beta$, there will be an identifier $Id(s', R_\beta)$ executed. Correspondingly, an identifier $Id(\mathcal{S}\text{-after-}\mu_j, R_\beta)$ after the trace μ_j and an identifier $Id(\mathcal{S}\text{-after-}\mu_r, R_\beta)$ after the trace μ_r will be executed. Since $\mathcal{S}$-after-$\mu_j \not\simeq \mathcal{S}$-after-$\mu_r$, $\mathcal{B}$ is deterministic and $\mathcal{B}$-after-$\mu_j = \mathcal{B}$-after-μ_r, at least for one of the separators the designated state would not be reached, i.e., the verdict fail will be produced. ♦

Finally, we demonstrate how the obtained result is related to the problem of m-complete test generation. Suppose that all the deterministic FSMs with at most m states in the fault domain $\Im_m(\mathcal{S})$ can be explicitly enumerated and for each such FSM $\mathcal{B}$, an FSM $\mathcal{G}_\mathcal{B}$ is derived by Algorithm 2. Let the FSM $\mathcal{U}$ be the union of all FSMs $\mathcal{G}_\mathcal{B}$ which, if it is not output-complete, is completed as follows. For each trace αio of $\mathcal{U}$ if there exists $o' \in O$ such that $\mathcal{U}$ has no trace $\alpha io'$ then a completed pass trace $\alpha io'$ is added if $\alpha io'$ is a trace of the specification FSM $\mathcal{S}$; otherwise, a fail trace $\alpha io'$ is added. The reason is that such a trace $\alpha io'$ does not belong to any machine in $\Im_m(\mathcal{S})$; it has to be added since a test is by definition output-complete. In fact, a test can be output-completed in arbitrary way, since those traces will never be observed in testing deterministic FSMs with at most m states.

Theorem 2. *Given an FSM $\mathcal{S}$ and a test $\mathcal{U}$ derived by the above procedure, the test $\mathcal{U}$ is m-complete for $\mathcal{S}$, i.e., it is complete in the fault domain $\Im_m(\mathcal{S})$ of complete deterministic FSMs.*

Proof. A trace of test $\mathcal{U}$ is a pass trace if and only if this trace is a trace of the specification FSM $\mathcal{S}$. Consider FSM $\mathcal{B} \in \Im_m(\mathcal{S})$ that is not a reduction of $\mathcal{S}$; by construction, an FSM $\mathcal{G}_\mathcal{B}$ derived by the above algorithm has a fail trace that is a trace of $\mathcal{B}$ and thus, test $\mathcal{U}$ has a fail trace that is a trace of $\mathcal{B}$. ♦

Such a characterization of the relationship between the executed test and m-complete tests is of a more theoretical than practical interest, as in testing one usually deals with a given implementation and not with an arbitrary collection of them.

5 Related Work

Most of the previous work in test generation from NFSMs relies on the execution of all test fragments forming the complete tests, see e.g., [4, 6]. The proposed method requires adaptive execution avoiding tests which are not related to a given implementation. It also differs in the way test fragments are treated. The construction of traversal sets follows the most recent idea elaborated for deriving preset complete tests in our previous work [6], which improves previous proposals, (see [6] for more references) including those used for adaptive testing such as [2, 3, 7]. In all the previous work, the test fragment addressing state reachability includes transfer sequences only to deterministically reachable states, which is extended in this paper by considering definitely reachable states.

The notions of preamble used in this paper and finite transfer tree considered in [11] serve the same purpose of modeling an adaptive process of transferring an NFSM into a desired state. Differently from that work, we use, instead of a tree, a state machine. Modeling preamble as NFSM allows us not only to establish a direct relation with a specification machine, namely, that a preamble is just a certain submachine of the specification FSM, moreover if it exists then any of its reductions possesses the preamble. This allows us to solve its existence and construction problems in a simple intuitive way.

The notion of separator is similar to that of state distinguishing strategies considered in [1] and in [11]. Differently from that work, we not only use an FSM to describe a strategy, but also offer an exact characterization of all state distinguishing strategies in the form of an FSM, called a canonical separator, which is obtained from the self-product of the specification NFSM. A distinguishing strategy becomes then a certain submachine of the canonical separator. The method of [1] allows to test whether a given FSM is *r*-compatible with the specification FSM (but not that the former is a reduction of the latter, as we do), and [3] extends this result to a set of FSMs. The work in [5] also addresses adaptive testing of nondeterministic machines, but its results cannot be used to prove that a given DFSM is a reduction of the specification machine. Compared to [7], we pre-compute all the test fragments, simplifying the test generation process.

6 Conclusion

In this paper, we addressed the problem of adaptive testing of a deterministic implementation machine from its nondeterministic specification. We proposed a method for defining test fragments, combining and executing them in adaptive manner against a given implementation FSM such that the test execution terminates with the verdict pass if and only if it is a reduction of the specification. The novelty of the method lies in the use of definitely reachable states missed by the previous methods and selective execution of test fragments depending on observed traces in adaptive test execution.

Our current work concerns the generalization of the obtained results to nondeterministic implementations. As a future work, it would be interesting to investigate a combination of the proposed method with the specification refinement approach.

Acknowledgement. The authors want to thank a reviewer for useful comments.

References

1. Alur, R., Courcoubetis, C., Yannakakis, M.: Distinguishing Tests for Nondeterministic and Probabilistic Machines. In: 27th ACM Symp. on Theory of Comp., pp. 363–372 (1995)
2. Hierons, R.M.: Testing from a Non-Deterministic Finite State Machine Using Adaptive State Counting. IEEE Transactions on Computers 53(10), 1330–1342 (2004)
3. Gromov, M.L., Evtushenko, N.V., Kolomeets, A.V.: On the Synthesis of Adaptive Tests for Nondeterministic Finite State Machines. Progr. and Comp. Software 34(6), 322–329 (2008)
4. Luo, G.L., Bochmann, G.v., Petrenko, A.: Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized Wp-method. IEEE Transactions on Software Engineering 20(2), 149–161 (1994)
5. Nachmanson, L., Veanes, M., Schulte, W., Tillmann, N., Grieskamp, W.: Optimal Strategies for Testing Nondeterministic Systems. In: ISSTA 2004 Software Eng. Notes, vol. 29, pp. 55–64. ACM, New York (2004)
6. Petrenko, A., Yevtushenko, N.: Conformance Tests as Checking Experiments for Partial Nondeterministic FSM. In: Grieskamp, W., Weise, C. (eds.) FATES 2005. LNCS, vol. 3997, pp. 118–133. Springer, Heidelberg (2006)
7. Petrenko, A., Yevtushenko, N.: Refining Specifications in Adaptive Testing of Nondeterministic Finite State Machines. Vestnik Tomskogo Gos. Universiteta 1(6), 99–114 (2009)
8. Petrenko, A., Yevtushenko, N., Bochmann, G.v.: Testing Deterministic Implementations from their Nondeterministic Specifications. In: Proceedings of the IFIP Ninth International Workshop on Testing of Communicating Systems, pp. 125–140 (1996)
9. Petrenko, A., Yevtushenko, N.: Testing from Partial Deterministic FSM Specifications. IEEE Transactions on Computers 54(9), 1154–1165 (2005)
10. Simao, A., Petrenko, A., Yevtushenko, N.: Generating Reduced Tests for FSMs with Extra States. In: Núñez, M., Baker, P., Merayo, M.G. (eds.) TESTCOM 2009. LNCS, vol. 5826, pp. 129–145. Springer, Heidelberg (2009)
11. Zhang, F., Cheung, T.: Optimal Transfer Trees and Distinguishing Trees for Testing Observable Nondeterministic Finite-State Machines. IEEE Transactions on Software Engineering 29(1), 1–14 (2003)

Compositional Random Testing Using Extended Symbolic Transition Systems

Christian Schwarzl[1], Bernhard K. Aichernig[2], and Franz Wotawa[2]

[1] Virtual Vehicle,
Inffeldgasse 21a, 8010 Graz, Austria
christian.schwarzl@v2c2.at

[2] Institute for Software Technology,
Graz University of Technology, 8010 Graz, Austria
{aichernig,wotawa}@ist.tugraz.at

Abstract. The fast growth in complexity of embedded and software enabled systems requires for automated testing strategies to achieve a high system quality. This raise of complexity is often caused by the distribution of functionality over multiple control units and their connection via a network. We define an extended symbolic transition system (ESTS) and their compositional semantics to reflect these new requirements imposed on the test generation methods. The introduced ESTS incorporates timed behavior by transition execution times and delay transitions. Their timeout can be defined either by a constant value or an attribute valuation. Moreover we introduce a communication scheme used to specify the compositional behavior and define a conformance relation based on alternating simulation. Furthermore we use the conformance relation as the basis for a simple random test generation technique to verify the applicability of the presented approach. This formal framework builds the foundation of our UML test case generator.

Keywords: Symbolic transition system, concurrent reactive behavior, test case generation.

1 Introduction

Since testing is an important task to ensure a certain system quality, the used techniques and approaches in this field strongly advanced in recent years. Nevertheless testing still remains a laborious task and is – due to the high degree of manual interaction – error prone. The steadily increasing complexity of embedded systems requires a high degree of test automation to be able to execute and analyze the large amount of needed test cases.

A further increase in automation can be achieved by the generation of test cases from formal specifications, which has gained a lot of attention in research in recent years and becomes more and more popular in the industry. However, the currently available industrial- and scientific-tools based on unified modeling language (UML) state machines or symbolic transition systems (STSs) [6] are limited to a single model. This situation does not meet the requirements of modern embedded systems, which often consists of communicating components.

For this reason we have implemented a test case generation algorithm, which works on the basis of an extended symbolic transition system (ESTS) composition presented in this work. This prototype supports a systematic and randomized test generation approach, which detailed description is beyond the focus of this paper.

Our contribution in this work is the extension of the STS by delay- and completion-transitions, timing groups, transition priorities and their execution duration. In addition we provide a precise semantics and formally define the model composition. Furthermore we show the applicability of the presented symbolic framework by a randomized test generation approach and the conformance relation to the system under test (SUT).

The remainder of the paper is structured as follows: Section 2 defines the structure of an ESTS and Section 3 precisely describes the compositional behavior. In Section 4 a conformance relation based on alternation simulation is provided and the applicability of the presented approach is demonstrated in Section 5. Section 6 presents an overview of the related work and Section 7 concludes the paper and gives a short outlook.

2 Extended Symbolic Transition System

In this section we define the structure of an ESTS and its semantics with respect to its contained states and transitions. Based on this structure we explain the creation of traces through the ESTS caused by external interactions.

Definition 1 (Extended Symbolic Transition System). *An ESTS is a tuple $\langle \mathcal{S}, \Lambda, \mathcal{A}, \mathcal{P}, \mathcal{T}, \mathcal{G}, s_0, \iota_0 \rangle$, where $\mathcal{S}$ is a set of states, Λ are the signals, $\mathcal{A}$ are the attributes, $\mathcal{P}$ are the signal parameters, $\mathcal{T}$ is the transition relation and $\mathcal{G}$ is a set of timing groups. Moreover s_0 is the initial state and ι_0 is the initial attribute valuation.* [1] $\square$

We define the set of signals $\Lambda = \Lambda_i \cup \Lambda_o$ as the union of input Λ_i and output Λ_o signals. The set $\Lambda_* = \Lambda \cup \{\tau, \gamma, \delta\}$ is the complete set of all defined signals, whereas the constants $\tau, \gamma, \delta \notin \Lambda$ represent the special unobservable-, completion- and delay-signal types, respectively. The attributes $\mathcal{A}$ are the variables of an ESTS and the parameters $\mathcal{P}$ are variables attached to input- or output-signals $\lambda \in \Lambda$. Further it holds that $\mathcal{A} \cap \mathcal{P} = \emptyset$ and we use $V = \mathcal{A} \cup \mathcal{P}$.

The transition relation of an ESTS is defined as $\mathcal{T} \subseteq \mathcal{S} \times \Lambda_* \times \mathfrak{F}(V) \times \mathfrak{T}(V) \times \mathfrak{P} \times \mathcal{S}$, where $\mathfrak{F}(V)$ is a first order logic formula without quantifiers, $\mathfrak{T}(V)$ are attribute value assignment statements given as mathematical terms over the variables V and $\mathfrak{P}$ are the priorities.

[1] In [6] a different naming convention is used, where states are locations, transitions are switches, attributes are location variables and parameters are interaction variables.

We write a transition $t \in \mathcal{T}$ as $s \xrightarrow{\lambda,\varphi,\rho,p_t} s'$, where $s \in \mathcal{S}$ is its source state, $s' \in \mathcal{S}$ is the destination state, $\lambda \in \Lambda_*$ defines its type, $\varphi \in \mathfrak{F}(V)$ is the guard, the action $\rho = \langle \rho_1, \rho_2, .., \rho_n \rangle$ is an ordered list of assignment terms $\rho_j \in \mathfrak{T}(V)$ with list index j and $p_t \in \mathfrak{P}$ is its priority, where $p_t \in \mathbb{N}_0$. A transition $t \in \mathcal{T}$ has a traversal probability $\alpha_t \in \mathbb{R}$ and an execution duration $d_t \in \mathbb{N}_0$ in addition. We omit the presentation of the priority p_t in the remainder for simplicity if the context allows it.

Definition 2 (Timing Group). *A timing group g is a tupel $\langle c, S_\delta, T_\delta, T_r \rangle$, where c is its clock, $S_\delta \subseteq \mathcal{S}$ are the contained states, $T_\delta \subseteq \mathcal{T}$ are the delay transitions and $T_r \subseteq \mathcal{T}$ are the clock reset transitions.* □

A timing group $g \in \mathcal{G}$ specifies a set of states S_δ where each of these states has an outgoing delay transition $t_\delta \in T_\delta$ with the same timeout $n_\delta \in \mathbb{N}_0$. The states in the timing group share a clock c, which is used to trigger the traversal of one of these outgoing delay transitions. The timeout of a delay transition can either be defined by a constant- or by an attribute-value like $n_\delta = 100$ or $n_\delta = x$ if $x \in \mathcal{A}$. Our delay transitions should not be confused with the similar delay transitions in timed automata semantics. In timed automata semantics, delay transitions serve to express the possible waiting times in a state and hence are reflexive transitions increasing time only. Our delay transitions increase time and change the state.

We require that every state $s \in S_\delta$ has an outgoing delay transition $t_\delta \in T_\delta$ to the same destination state $s' \in \mathcal{S}$. This ensures that one of the delay transitions is traversed after the defined amount of time – specified by the timeout of the delay transitions – within the timing group has elapsed. The timing group clock is set to zero if one of the clock reset transitions $t_r \in T_r$ is traversed.

We use $\vartheta = \iota \cup \varsigma$ as variable valuation containing the values of attributes ι and the current signal parameters ς. Accordingly we denote the update of a variable valuation ϑ' according to an action ρ as $\vartheta \mapsto \vartheta'(\rho)$. Given a valuation ϑ and a guard φ we write $\vartheta \models \varphi$ if the valuation satisfies the guard φ, which is a first order logical formula.

Example 1. Figure 1 shows an illustrative example of two communicating ESTS, where we use ? to mark input- and ! for output-signals, the keyword `delay(x)` to indicate delay transitions, the parameter x to denote the delay time and show only the guard of the completion transition γ. Blocking states as given in Definition 4 are shown with two border lines and states belonging to the same timing group are filled gray. Since each of these ESTSs contains only one timing group their presentation is unambiguous. Transition guards are shown within squared brackets and actions follow a slash. □

Definition 3 (Configuration). *A configuration q is a tuple $\langle s, \iota \rangle$, where $s \in \mathcal{S}$ is an explicit state and ι specifies the values of all attributes in $\mathcal{A}$.* □

A configuration fully defines the current state of an ESTS and accordingly we define the initial state $q_0 = \langle s_0, \iota_0 \rangle$, where s_0 is the initial state and ι_0 the initial valuation. In the remainder we use $\mathcal{Q}$ to indicate the set of all possible configurations.

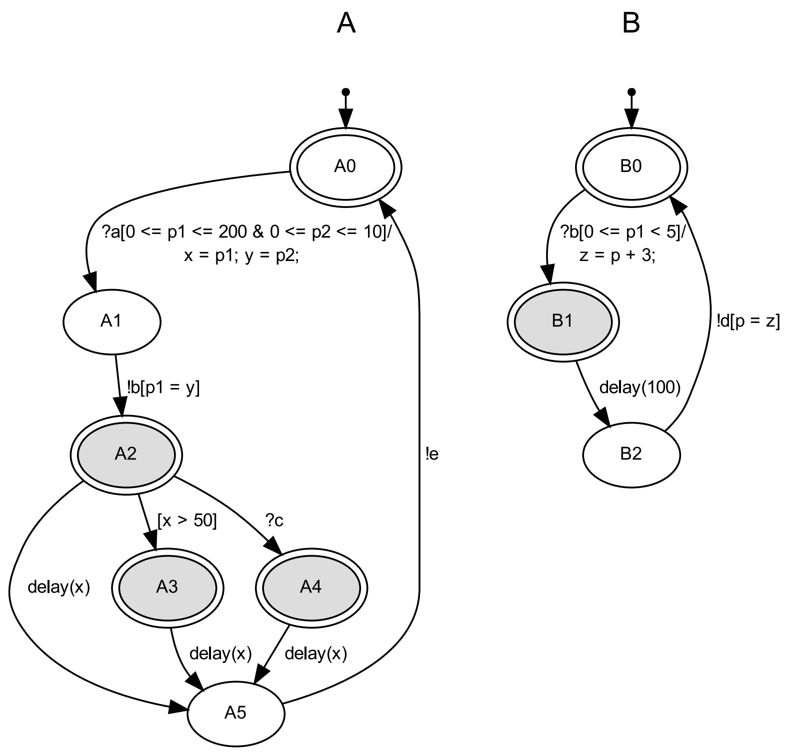


Fig. 1. Two communicating ESTSs A and B

For the following definitions we use $t \in \mathcal{T}$, $\lambda \in \Lambda$, $\lambda_* \in \Lambda_*$ and $g \in \mathcal{G}$. The function $\mathtt{src}(t)$ returns the transition source state, $\mathtt{dest}(t)$ its destination state, $\mathtt{signal}(t)$ the signal of a transition, $\mathtt{arity}(\lambda) \in \mathbb{N}_0$ the number of signal parameters and $\mathtt{dur}(t)$ the execution duration d_t.

In addition we use $\mathtt{out}(s, \lambda_*) = \{t \in \mathcal{T} \mid \mathtt{signal}(t) = \lambda_* \wedge \mathtt{source}(t) = s\}$ returning all outgoing transitions of s having signal λ_*, $\mathtt{delay}(t) = n$ if $\mathtt{signal}(t) = \delta$ or $n = 0$ otherwise, providing the delay time of a transition and $\mathtt{delay}_{min}(t) = min\left(\bigcup \mathtt{delay}(t_\delta) \mid t_\delta \in \mathtt{out}(\mathtt{src}(t), \delta)\right)$, which is is the minimum delay of currently active delay transition.

The function $\mathtt{prio}(t)$ returns the transition priority p_t and $\mathtt{prio}_{max}(s, \lambda) = max\left(\bigcup \{\mathtt{prio}(t) \mid t \in \mathtt{out}(s, \lambda)\}\right)$ calculates the maximum priority of all outgoing transitions from state s having signal λ.

$\mathtt{clk}(t) = \bigcup \{\mathtt{c}(g) \mid \mathtt{src}(t), \mathtt{dest}(t) \in \mathtt{states}(g) \wedge t \notin \mathtt{r}(g)\}$ returns the set of all timing groups to which the given transition belongs.

The function $\mathtt{reset}(t) = \bigcup \{\mathtt{c}(g) \mid t \in \mathtt{r}(g)\}$ defines the clocks which are reset by a traversal of t, where $\mathtt{c}(g)$ returns the clock c, $\mathtt{r}(g)$ the clock reset transitions T_r and $\mathtt{states}(g)$ the contained states S_δ of the given timing group g.

Example 2. Let the ESTS A shown in Figure 1 be initialized with the configuration $q_0 = \langle A0, \{x = 0, y = 0\}\rangle$. After the traversal of the transition $A0 \xrightarrow{?a[0 \leq p1 \leq 200 \& 0 \leq p2 \leq 10]/x=p1;y=p2;} A1$ with the parameters $p1 = 70$ and $p2 = 6$ the ESTS A has the configuration $q' = \langle A1, \{x = 70, y = 6\}\rangle$ as defined by the transition action. □

2.1 Semantics

This section describes the behavior of every allowed transition type, which is defined by the prerequisites of the transition traversal and the performed state and attribute value changes.

We define the semantics of an ESTS in the Rules (1) to (6), where we require $\vartheta \models \varphi$ if not stated differently, we use $Q' \subseteq \mathcal{Q}$ as the set of configurations after a transition traversal and $\mapsto$ denotes an assignment mapping. The shown semantics is similar to the one presented by Frantzen et. al, but we only describe the evaluation of the post state to highlight the extensions.

Rule *Empty* (1) states that the state and the valuation does not change if the empty signal ϵ is executed, where $\top$ indicates that the guard is always satisfied and `id` is the identity function.

$$\frac{s \xrightarrow{\epsilon,\top,\mathtt{id}} s}{Q' \mapsto \{\langle s, \vartheta\rangle\}} \tag{1}$$

In *Input* (2) the semantics of a signal reception $\lambda_i \in \Lambda_i$ is defined, where a signal reception can only be executed if it is sent before an active timeout. Rule *Timed* (3) shows that a delay transition with the shortest delay is executed as soon as the defined amount of time has elapsed. After the traversal the current state and the attribute valuation are updated and the clocks of the timing groups where $t \in T_r$ are set to zero.

$$\frac{s \xrightarrow{\lambda_i,\varphi,\rho,p_t} s' \wedge p_t = \mathtt{prio}_{max}(s, \lambda_i) \wedge \forall c \in \mathtt{clk}(t) \mid c < \mathtt{delay}_{min}(t)}{Q' \mapsto \{\langle s', \vartheta'(\rho)\rangle\}, \forall c \in \mathtt{clk}(t) \mid c \mapsto c + \mathtt{dur}(t), \forall c \in \mathtt{reset}(t) \mid c \mapsto 0} \tag{2}$$

$$\frac{s \xrightarrow{\delta,\varphi,\rho,p_t} s' \wedge p_t = \mathtt{prio}_{max}(s, \delta) \wedge \mathtt{c}(t) \geq \mathtt{delay}(t) \wedge \mathtt{delay}(t) = \mathtt{delay}_{min}(t)}{Q' \mapsto \{\langle s', \vartheta'(\rho)\rangle\}, \forall c \in \mathtt{clk}(t) \mid c \mapsto c + \mathtt{delay}(t) + \mathtt{dur}(t), \forall c \in \mathtt{reset}(t) \mid c \mapsto 0} \tag{3}$$

The rules *Output* (4) and *Completion* (5) have the same behavior in terms of updating configurations and the handling of the clock updates as in (2), but are executed as soon their guard is satisfied. They differ in the IO behavior, because in (4) a signal is sent and (5) only allows for a deterministic configuration update.

$$\frac{s \xrightarrow{\lambda_o,\varphi,\rho,p_t} s' \wedge p_t = \mathtt{prio}_{max}(s, \lambda_o)}{Q' \mapsto \{\langle s', \vartheta'(\rho)\rangle\}, \forall c \in \mathtt{clk}(t) \mid c \mapsto c + \mathtt{dur}(t), \forall c \in \mathtt{reset}(t) \mid c \mapsto 0} \tag{4}$$

$$\frac{s \xrightarrow{\gamma,\varphi,\rho,p_t} s' \wedge p_t = \texttt{prio}_{max}(s,\gamma)}{Q' \mapsto \{\langle s', \vartheta'(\rho)\rangle\}, \forall c \in \texttt{clk}(t) \mid c \mapsto c + \texttt{dur}(t), \forall c \in \texttt{reset}(t) \mid c \mapsto 0} \quad (5)$$

Unobservable (6) defines the semantics of a non-deterministic configuration update. The traversal of an τ transition is not observable and the resulting symbolic states in Q' are its source and destination state.

$$\frac{s \xrightarrow{\tau,\varphi,\rho,p_t} s' \wedge p_t = \texttt{prio}_{max}(s,\tau)}{Q' \mapsto \{\langle s, \vartheta\rangle, \langle s', \vartheta'(\rho)\rangle\}, \forall c \in \texttt{clk}(t) \mid c \mapsto c + \texttt{dur}(t), \forall c \in \texttt{reset}(t) \mid c \mapsto 0} \quad (6)$$

2.2 Simulation

In this section the execution of an ESTS is explained by the creation of execution traces caused by signal receptions or delays. Such execution traces are always embedded between two blocking states described in Definition 4.

Definition 4 (Blocking State). *A blocking state $\tilde{s}$ is a state $s \in \mathcal{S}$ for which it holds that $\exists \lambda \in \lambda_i \cup \delta \mid |\texttt{out}(s,\lambda)| \neq 0 \vee \texttt{out}(s,\lambda_*) = \emptyset$.* □

This means a blocking state is a state having at least one outgoing transition of type λ_i or δ or no outgoing transition $\lambda_* \in \Lambda_*$ at all. Accordingly we denote a blocking configuration as $\tilde{q} = \langle \tilde{s}, \iota\rangle$. Note that a blocking state does not limit the occurrence of outgoing τ, γ or $\lambda_o \in \Lambda_o$ transitions, which makes a mixed state possible. Since we allow outgoing output transitions, the state is not a *quiescent* state as defined in [6] or [11].

Based on Definition 4 we can define an execution trace as shown in Definition 5, which connects two blocking states and must not be interrupted. It is a sequence of transitions and consists of a triggering η_t and completion η_c part, where η_t consists only of transitions with signals $\Lambda_t = \Lambda_i \cup \delta$ and $\Lambda_c = \Lambda_o \cup \gamma \cup \tau$.

Definition 5 (Execution Trace). *An execution trace $\eta = t_1, t_2, \ldots, t_n$ is a sequence of transitions $t_1, \ldots, t_n$, where $\eta_t = t_1$, $\eta_c = t_2, \ldots, t_n$ and $\forall t_i \in \eta \setminus t_1 \mid$ $\texttt{dest}(t_i) = \texttt{source}(t_{i+1})$.* □

The length of an execution trace is denoted as $|\eta|$ and it holds that $|\eta| \geq 1$, where $|\eta_t| = 1$ and $|\eta_c| \geq 0$. Due to the allowed non-deterministic behavior of an ESTS a signal reception or a time lapse can cause multiple execution traces leading to the resulting list $\mathcal{E}(\tilde{q},\lambda)$. Its recursive generation is defined by $\mathcal{E}'(\tilde{q},\lambda) = \bigcup\{e(\eta) \mid \eta \in \mathcal{E}(\tilde{q},\lambda)\}$, where $\lambda \in \Lambda_c$ is the triggering input, $\tilde{q} \in \mathcal{Q}$ is the current configuration and $\mathcal{E}'(\tilde{q},\lambda)$ is initialized with $t \in \texttt{out}(\tilde{q},\lambda) \mid \vartheta \models \varphi$. The function $e(\eta)$ is defined in Equation (7), where $\tilde{q}'$ is the destination state of the last contained transition in η and $\circ$ is the concatenation of traces.

$$e(\eta)' = \begin{cases} \bigcup_{t_c}\{e(\eta \circ t_c) \mid t_c \in \texttt{out}(\tilde{q}', \Lambda_c)\} & \text{if } \texttt{out}(\tilde{q}',\Lambda_t) = \emptyset \wedge \texttt{out}(\tilde{q}',\Lambda_c) \neq \emptyset \\ \eta \cup \bigcup_{t_c}\{e(\eta \circ t_c) \mid t_c \in \texttt{out}(\tilde{q}', \Lambda_c)\} & \text{if } \texttt{out}(\tilde{q}',\Lambda_t) \neq \emptyset \wedge \texttt{out}(\tilde{q}',\Lambda_c) \neq \emptyset \\ \eta & \text{otherwise} \end{cases} \quad (7)$$

The recursive generation of the completion steps in (7) creates an infinite number of traces if a loop of completion transitions exists, which actions do not falsify one of its guards.

Example 3. Let ESTS A shown in Figure 1 again be initialized with $q_0 = \langle A0, \iota_0 \rangle$, where $\iota_0 = \{x = 0, y = 0\}$. Then we can build the list of initial execution traces $\mathcal{E}(\tilde{q_0}, \lambda) = A0 \xrightarrow{?a[0 \leq p1 \leq 200 \& 0 \leq p2 \leq 10]/x=p1;y=p2;} A1$, because we use $\lambda = a$, $\vartheta = \iota_0 \cup \varsigma$ and $\varsigma = \{p1 = 30, p2 = 9\}$, which satisfies the transition guard. If we would use $\varsigma = \{p1 = 30, p2 = 15\}$ instead, then $\mathcal{E}(\tilde{q}, \lambda) = \emptyset$ because $p2 > 10$. The recursive update of this list leads to the final trace list $\mathcal{E}(\tilde{q_0}, \lambda) = A0 \xrightarrow{?a[0 \leq p1 \leq 200 \& 0 \leq p2 \leq 10]/x=p1;y=p2;} A1 \xrightarrow{!b[p1=y]} A2 \xrightarrow{[x>50]} A3$, where the first transition is the triggering- η_t and the last two transitions are the completion-η_c part. The last transition has to be added, because $\vartheta = \{x = 30, y = 9\} \models \varphi$ of the completion transition, which is executed immediately after its guard is satisfied. □

3 Composition

In this section the model composition based on the signal communication between the involved ESTSs is explained. Furthermore we clearly define the observations and interactions, which can be made by the environment.

3.1 Model Communication

In this work we use a deterministic communication scheme using a global queue Q to pass signals between ESTSs. Since we required that an execution trace must not be interrupted, the system behavior can be described by an concatenation of such traces. This concept is similar to the approach presented in the language Creol [3], where only one thread is active at a time.

The reception of a signal λ or the lapse of time in the state $\tilde{q}$ leads to a list of execution traces $\eta \in \mathcal{E}'(\tilde{q}, \lambda)$. The needed execution time or the sent signals by a trace η can cause reactions in other ESTSs or the environment. Therefore a list of system execution traces $\mathcal{E}_M(\tilde{q}, \lambda)$ has to be created, where $M \subseteq \mathcal{M}$ is the set of all involved ESTSs. These traces contain the initial trace η and its concatenated reaction traces of the other ESTSs.

Since not every signal needs to be sent or be observable by the environment – e.g. communication within one component – we split the signals into two categories. The first category contains signals observed or created by the environment $\bar{\Lambda} = \bar{\Lambda}_i \cup \bar{\Lambda}_o \mid \bar{\Lambda}_i \subseteq \Lambda_i \wedge \bar{\Lambda}_o \subseteq \Lambda_o$ to which we refer as *external communication* in the remainder. The second category is the *internal communication*, which contains the signals $\hat{\Lambda} = \hat{\Lambda}_o \cup \hat{\Lambda}_i \mid \hat{\Lambda}_o = \Lambda_o \setminus \bar{\Lambda}_o \wedge \hat{\Lambda}_i = \Lambda_i \setminus \bar{\Lambda}_i$ sent and received by one of the ESTSs. Note that signals part of the external communication *must* be and signals of the internal communication *might* be created or received by the environment.

The trace output signals used for the communication are defined in Definition 6 and are independent of the observability by the environment.

Definition 6 (Observables). *Let $obs(\eta) = \langle\langle \lambda_o, \overline{d}_1\rangle, \ldots, \langle \lambda_o, \overline{d}_N\rangle\rangle$ be the output signals of a trace $\eta \in \mathcal{E}'(\tilde{q}, \lambda)$, where its result is a list of tuples, N is the number of output signals on the trace, $\lambda_o \in \Lambda_o$ are the signals and $\overline{d}$ is the time period in which the signal has to be sent. We define the observable $\overline{obs}(\eta) = obs(\eta) \mid \lambda_o \in \bar{\Lambda}_o$ and unobservable $\widehat{obs}(\eta) = obs(\eta) \mid \lambda_o \in \hat{\Lambda}_o$ output signals of η.* □

The required state updates in the other ESTSs are performed using the signals in $obs(\eta)$, which are passed via the global message queue Q. The execution uses the same algorithm as described above and leads to the execution trace $\eta_m \in \mathcal{E}_m(\tilde{q}, \lambda)$ in the ESTS $m \in M$. We call a trace $\eta_M \in \mathcal{E}_M(\tilde{q}, \lambda)$ containing the initial execution trace η and all according reactions η_m *system execution trace* in the remainder.

Definition 7 (System Execution Trace). *A system execution trace $\eta_M = \eta_{m_1} \circ \cdots \circ \eta_{m_N}$ is the catenation of execution traces η_{m_i} of an ESTS $m \in M$, where $i = 1..N$, $N = |M|$ is the number of entries in M and $M \subseteq \mathcal{M}$ is the set of involved ESTS.* □

$$rt(\eta) = \begin{cases} \eta & \text{if } obs(\eta) = \emptyset \\ \bigcup_{\eta'} \{\eta \circ \eta' \mid \eta' \in rt\left(\eta_m \in \mathcal{E}_m(\tilde{q}_m, \lambda) \mid \lambda \in obs(\eta)\right)\} & \text{otherwise} \end{cases} \quad (8)$$

The creation of system execution traces is defined recursively by the reception trace function $rt(\eta)$ shown in Equation (8), where $\tilde{q}_m$ is the current configuration of $m \in M$. It shows that a system trace is a recursive concatenation of all execution traces of other ESTSs caused by the output signals on the initial path. Given the initial traces $\eta \in \mathcal{E}(\tilde{q}, \lambda)$, we can build a list of all possible execution traces $\mathcal{E}_M(\tilde{q}, \lambda) = \bigcup rt(\eta) \mid \eta \in \mathcal{E}(\tilde{q}, \lambda)$.

This algorithm ensures that all signals stored in the queue are processed before the next execution step can begin. However, each execution trace can also produce output signals, which are also enqueued as described above. This allows for the creation of infinite loops between the involved ESTS, where a signal reception causes an output signal received in another ESTS, which reception causing the initial signal sending responsible for the initial stimuli.

Since the reception of signals has a higher priority than the traversal of delay transitions, their influence was neglected during the finding of the system execution traces. Due to the fact that the traversal of a transition t needs the time $\mathtt{dur}(t)$ to be completed, the execution time is calculated in the function $\mathtt{time}(\eta)$ given in (9).

$$\mathtt{time}(\eta) = \sum_{i=1}^{N} \mathtt{dur}(t_i) + \mathtt{delay}(t_i) - \mathtt{time}(\eta_g(t_i)) \quad (9)$$

In (9) η_g is the connected sub-trace of η consisting of the transitions contained in the timing group g to which the transition t_i belongs. Since an execution trace can also contain transitions not belonging to the ESTS m, which is the owner of the timing group g, these transitions still have to be included. A formal definition of the sub-trace creation is given in (10), where i is the index of t_i in η. It uses (11) to extract the trace from η according to the given indices and (12) to find the start index of the trace based on (13) returning the transitions of the ESTS to which the transition t_i belongs.

$$\eta_g(t_i) = \mathtt{sub}(\eta, i, k) \mid k = \mathtt{idx}_e(t_i, T_m), T_m = \mathtt{gtrace}(\eta, t_i) \tag{10}$$

Using Definition 5 a sub-trace of η is given by (11), which consists of the transitions in η lying in the range $[i, j]$, which is defined by the given indices.

$$\mathtt{sub}(\eta, i, j) = \langle t_i, .., t_j \mid t_i, t_j \in \eta \wedge 1 \leq i, j \leq |\eta| \wedge j \geq i \rangle \tag{11}$$

In (12) the minimum index k of the of the given transition t in T_m is calculated, which references to the first transition of the trace stored in T_m.

$$\mathtt{idx}_e(t, T_m) = k \in \mathbb{N} \mid (\exists t_k \in T_m \mid k = min_{idx}(T_m)) \tag{12}$$

The function $\mathtt{gtrace}(\eta, t_i)$, as defined in (13), returns all transitions and their indices in η satisfying the following criteria, where g is the timing group belonging to t_i. The first term of the constraint $t_j \in \eta$ requires that the transition belongs to η and the second term $\mathtt{src}(t_j), \mathtt{dest}(t_j) \in \mathtt{states}(g)$ that the transitions are contained in the same timing group as t_i. Term three $\mathtt{dest}(t_j) = \mathtt{src}(t_{j+1})$ ensures that the transitions represent a trace without any structural holes. The last term $\exists t_j = t_i$ requires that the given transition t_i is contained in that connected trace to prevent an ambiguous result if t_i is traversed multiple times in trace η.

$$\begin{aligned}\mathtt{gtrace}(\eta, t_i) = \bigcup \{t_j \mid t_j \in \eta \wedge \mathtt{src}(t_j), \mathtt{dest}(t_j) \in \mathtt{states}(g) \\ \wedge \mathtt{dest}(t_j) = \mathtt{src}(t_{j+1}) \wedge \exists t_j = t_i\}\end{aligned} \tag{13}$$

The elapsed time $\mathtt{time}(\eta_M)$ is used to trigger active delay transitions after the processing of the enqueued output signals has finished. This is done by finding the transition with the smallest time overdue $\delta_{due} = \mathtt{delay}(t) - \mathtt{time}(\eta_M)$. If such a transition exists it is executed using the same algorithm as described above. The execution can again cause the sending of new output signals, which are processed before the next delay transition is taken into account. This algorithm again allows for the modeling of an infinite loop, if two traces exist with $\mathtt{time}(\eta_1) \geq \mathtt{delay}(t_2)$ and $\mathtt{time}(\eta_2) \geq \mathtt{delay}(t_1)$ and which lead to their own source state, where η_1 and η_2 are the execution traces to the transitions t_1 and t_2, respectively. The execution traces gained from the processing of the delayed transitions are then concatenated to η being the final result.

In this approach we defined that the treatment of a signal reception has a higher priority than the traversal of an delayed transition. These rules allow that an active delay transition, whereas enough time has elapsed to trigger the traversal, is not traversed in favor to the transition receiving a signal from Q, even if the signal was enqueued after the timeout of a delayed transition.

4 Conformance

In this section the correctness of an implementation under test (IUT) with respect to a specification using alternating simulation [1] is explained. For simplicity, we only discuss the conformance of deterministic ESTS here. In the non-deterministic case the two ESTSs need to be determinized beforehand, similar to [12]. Generally it is required that the IUT can follow all inputs generated from and only produces outputs allowed by the specification. To provide a precise understanding we introduce the function $\texttt{moves}(\tilde{q}, \Lambda)$ shown in Equation (14) first, where $M \subseteq \mathcal{M}$ is a set of ESTSs, $\tilde{q}_M = \bigcup \tilde{q}_m \mid m \in M$ and $\tilde{q}_m$ is the blocking configuration of an ESTS $m \in M$. This function returns the union of all outgoing transitions of all $m \in M$ at state $\tilde{q}_m$, which guard is satisfied and signals are contained in the given set Λ.

$$\texttt{moves}(\tilde{q}_M, \Lambda) = \bigcup \{t \mid \texttt{src}(t) = \tilde{q}_M \wedge \tilde{q}_M \models \varphi_t \wedge \texttt{signal}(t) \in \Lambda\} \tag{14}$$

The meaning of alternating simulation as defined in [12] is formalized in Equation (15) and (16), where $\tilde{q}_1 \in \mathcal{Q}_1$ and $\tilde{q}_2 \in \mathcal{Q}_2$ are the sets of configurations of the IUT and the specification, respectively. Accordingly $\lambda_1 = \texttt{signal}(t_1)$ and $\lambda_2 = \texttt{signal}(t_2)$ are the signals of these transitions and q_1' and q_2' are the destination configurations.

$$\forall t_2 \in \texttt{moves}(\tilde{q}_2, \bar{\Lambda}_i \cup \delta) \mid (\exists t_1 \in \texttt{moves}(\tilde{q}_1, \bar{\Lambda}_i \cup \delta) \mid \lambda_2 = \lambda_1 \wedge \tilde{q}_2' = \tilde{q}_1') \tag{15}$$

Equation (15) states that all input or delay transition traversable in the specification in state $\tilde{q}_2 \in \mathcal{Q}_2$, which has a certain attribute valuation must also be executable on the IUT.

$$\forall t_1 \in \texttt{moves}(\tilde{q}_1, \bar{\Lambda}_o) \mid (\exists t_2 \in \texttt{moves}(\tilde{q}_2, \bar{\Lambda}_o) \mid \lambda_1 = \lambda_2 \wedge \tilde{q}_1' = \tilde{q}_2') \tag{16}$$

The inverse is true for outputs as shown in Equation (16), where it is required that every output produced by the IUT must be allowed by the specification. If both equations hold, then the basic I/O behavior of the implementation is correct with respect to the specification.

Since Equation (15) and (16) do not provide any information on the time behavior, we require in addition that the obtained output of the IUT fulfills the the timing constraints given in the ESTSs. Therefore we require that Equation (17) holds, where $\eta' \in \mathcal{E}_2(\tilde{q}_2, \texttt{signal}(t_2))$ and $\eta \in \mathcal{E}_1(\tilde{q}_1, \texttt{signal}(t_2))$ are the execution traces created for a given input on the IUT and specification respectively.

$$\forall \langle \lambda_o, d \rangle_j \in \overline{obs}(\eta) \mid (\exists \eta' \mid \langle \lambda_o', \bar{d} \rangle_j \in \overline{obs}(\eta') \wedge \lambda_o = \lambda_o' \wedge min(\bar{d}) \leq d \leq max(\bar{d})) \tag{17}$$

Equation (17) requires that for every observable output λ_o at time d part of the trace η produced by the IUT an according transition $t_o' \in \eta'$ exists, which contains the same outputs λ_o' within the time range $\bar{d}$. In (17) $j = 1..|\eta|$ is the index of the output occurrence in η. The time range $\bar{d}$ is given in Equation (18), which is the sum of the execution time elapsed up to the transition at position $k = \mathtt{idx}(t'_o, \eta')$ and includes a transition time jitter ε_k.

$$\bar{d} = \mathtt{time}(\mathtt{sub}(\eta', 1, k)) \pm \varepsilon_k \tag{18}$$

Since we have now defined the required outputs including the occurrence time of the IUT after an input was provided by the specification, we can now check the correctness of the IUT with respect to a given specification. Alternating simulation has the advantage in comparison to *ioco* that the conformance check is computational less intense and provides the same expressive power in the deterministic case [12].

5 Application

We show the applicability of the presented approach on a simple random test case generation example based on the ESTSs shown in Figure 1. In this example the external communication consists of the signals $\bar{\Lambda}_i = \{a, c\}$ and $\bar{\Lambda}_o = \{d\}$ and the internal communication is given by $\hat{\Lambda}_i = \{b\}$ and $\hat{\Lambda}_o = \{b\}$.

In this random approach named *random walk*, we explicitly trigger the traversal of outgoing transitions from the current state $\tilde{q}$. This is done by the generation of feasible data for the transitions $t \in \mathtt{moves}(\tilde{q}, \bar{\Lambda}_i)$ and the lapse of time for delay transitions. The input generation is done separately for each transition t by a constraint solver e.g. provided by GNU Prolog as in our case, which tries to find solutions satisfying the transition guards.

If multiple transitions are possible during this phase, meaning they leave the current state and their guard can be satisfied, we normalize their probabilities α and perform a random selection. Since we also want to generate sequences, which vary in their temporal behavior, the random selection of an explicit *wait* can also be chosen. In such a case the used wait time t_w has to be $0 \leq t_w \leq \delta_{min}$, where δ_{min} is the smallest timeout of all active delay transitions.

In the case an input action has been selected it is sent to and executed on the according ESTS. Wait actions in contrast are executed on the whole system, because the smallest active delay can belong to any of the involved ESTSs. After the input or wait action was performed the system execution traces $\mathcal{E}_M(\tilde{q}, \lambda)$ are generated.

Figure 2 shows three traces, which can be obtained if the inputs are applied as given in Table 1 and Table 2. The inputs in Table 2 lead to the same trace $T3$, but in the second case no additional wait is necessary, because the execution time is longer than the required delay. For these examples we assumed that ever transition has the same execution duration $t_d = 10$.

Both tables also show the observable output generated by each trace, which can be used during the execution of the test case on the SUT. Since it also

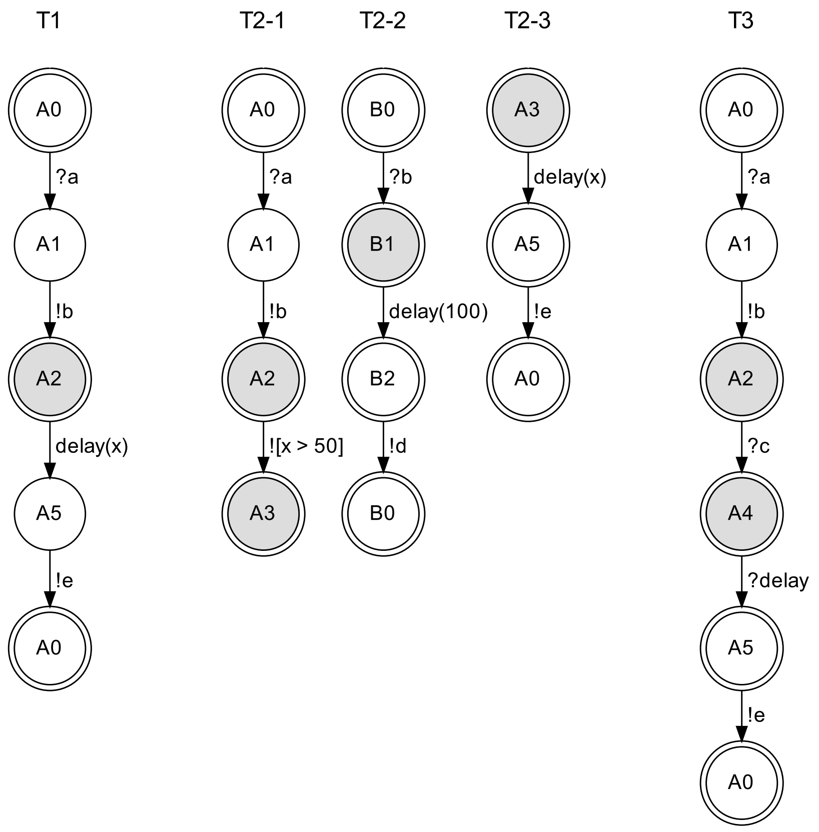


Fig. 2. Three system execution traces $T1, T2$ and $T3$

includes the latest point in time of the real signal reception, it is possible to check given timing constraints.

The random walk can be used for on-the-fly and offline test case generation. During on-the-fly testing the SUT is executed in parallel to the model and the input and outputs can be processed immediately. The advantage of this approach is that the current state is always known, which limits the state space especially in the presence of non-determinism. For offline test case generation all possible traces have to be stored and extended with every step during the random walk. Since non-determinism is allowed, the possible execution traces can be seen as a tree. This requires that the random walk has to continue in one step from every active leaf of this tree, to generate feasible test cases. Depending on the model these trees can become quite big due to the high number of possibilities.

Table 1. Inputs and outputs for traces $T1$ and $T2$

	T1	**T2**
Inputs	$?a(p1 \leq 50, p2 \geq 5)$ $wait(p1)$	$?a(p1 > 100 + 4 * t_d, p2 < 5)$ $wait(100)$ $wait(p1 - (100 + 4 * t_d))$
$\overline{obs}(\eta)$	$\{\langle e, p1 + 4 * t_d\rangle\}$	$\{\langle d(z+3), 100 + 6 * t_d\rangle, \langle e, p1 + 4 * t_d\rangle\}$
$\widehat{obs}(\eta)$	$\{\langle b, 2 * t_d\rangle\}$	$\{\langle b, 2 * t_d\rangle\}$

Table 2. Two possible inputs and outputs for trace $T3$

	T3-1	**T3-2**
Inputs	$?a(t_d < p1 \leq 50, p2 \geq 5)$ $?c$ $wait(p1 - (100 + 2 * t_d))$	$?a(p1 \leq t_d, p2 \geq 5)$ $?c$
$\overline{obs}(\eta)$	$\{\langle e, p1 + 4 * t_d\rangle\}$	$\{\langle e, p1 + 4 * t_d\rangle\}$
$\widehat{obs}(\eta)$	$\{\langle b, 2 * t_d\rangle\}$	$\{\langle b, 2 * t_d\rangle\}$

6 Related Work

Several approaches based on symbolic transition systems have been studied in recent years. STG [4] is a symbolic extension of the test tool TGV and allows the generation of test cases with respect to a given test purpose. The presented framework extends the approach described in [6], by timed behavior, completion transitions and model composition. The approach in [6] is implemented in the STSimulator, which provides a framework for on-the-fly random testing and is used in the Jambition Project [5] to automatically derive test cases for web applications. It uses **sioco** as conformance relation, which is the symbolic variant of **ioco** based on labeled transition systems (LTS).

Although the approaches described above were used successfully in various applications, they do not incorporate time as part of the specification. For this reason several extensions were introduced to lift the well understood approaches to timed models. This lead in the case of an LTS to its timed version and the according implementation relations like **tioco** and **rtioco**. A detailed discussion is given in [8], where a survey about the similarities and differences between these approaches and their variants is provided. However, these techniques still rely on an enumerative treatment of data limiting the scalability in data intense applications.

Also model checkers based on timed automata (TA) like UPPAAL [7,2] were used for behavior specification and test case generation. The timing constraints in a timed automata are given as time invariants on states and clock guards on transitions. UPPAAL also allows the interaction of data and time, meaning that attribute values can be used in the timing constraints. Their approach still relies on an explicit modeling of data and therefore faces the same scalability problems as methods based on an LTS. For the generation of test cases UPPAAL requires a deterministic specification, which limits the range of applicable use cases.

A symbolic variant based on timed automata is defined in [14], where the symbolic timed automata (STA) is introduced. It is a combination of an STS with the timing handling of TA and also allows the usage of attribute values as bounds for timing constraints. On the basis of the STA the testing conformance relation **stioco** being an symbolic extension of **tioco** is described. Although an STA allows similar semantics, it does not include a formal description of a composition and neglects unobservable events at the moment.

Spec Explorer [13], which can also use Spec# as specification language, uses alternating simulation to define the conformance between the IUT and the model. It was recently extended to work with UML sequence diagrams used for testing and program slicing. It also supports model composition in a similar way and allows the generation of test sequences based on a model composition. In contrast to the presented work no timed behavior can be modeled, which is which is one of the key features of the presented approach. Since Spec Explorer does no full symbolic state space exploration it allows a wider range of supported data types in contrast to this work, where we are limited to integer and boolean values.

7 Conclusion

We presented in this work an extended symbolic transition system based on the STS defined in [6]. Our approach extends this framework by the incorporation of delay- and completion-transitions for which we also provide a formal semantics. On top of the ESTS we defined a communication scheme, which uniquely defines the compositional behavior. In contrast to [6] we use alternating simulation as testing relation instead of **ioco**, for which we used the distinction between internal- and external-communication. This distinction allows for a clear separation between observable or controllable signals by the environment and those used internally.

We used this symbolic framework for a sample application allowing a random test case generation, which can be performed on-the-fly and offline. The incorporation of delay transitions and transition execution times allows for timing checks of the SUT like the verification of trace files containing time stamps.

The presented ESTS in this work contains similar elements as defined in UML state machines and therefore allows for a straight forward model transformation. For this reason it can be used as a formalization of the UML state machine semantics, which is required for test case generation. This is the first time we presented the formal framework on which basis we have implemented our test case generation prototype from UML state machines. Parts of the tools chain and its application on industrial use cases have been described in [9] and [10].

Future work includes the investigation of other communication schemata and an extension of the transition attributes to allow uncertainties in the timed behavior like $t_d = 100 \pm 5$. This would allow for checks ensuring that a certain signal did not arrive before a given point in time, which is required for modeling real time networks.

Acknowledgment. The authors wish to thank the reviewers, the "COMET K2 Forschungsförderungs-Programm" of the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT), the Austrian Federal Ministry of Economics and Labour (BMWA), Österreichische Forschungsförderungsgesellschaft mbH (FFG), Das Land Steiermark and Steirische Wirtschaftsförderung (SFG) for their financial support.

References

1. Alur, R., Henzinger, T.A., Kupferman, O., Vardi, M.Y.: Alternating refinement relations. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 163–178. Springer, Heidelberg (1998), http://portal.acm.org/citation.cfm?id=646733.759544
2. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
3. de Boer, F., Clarke, D., Johnsen, E.: A complete guide to the future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007), http://dx.doi.org/10.1007/978-3-540-71316-6_22
4. Clarke, D., Jéron, T., Rusu, V., Zinovieva, E.: STG: A symbolic test generation tool. LNCS, pp. 151–173. Springer, Heidelberg (2002)
5. Frantzen, L., Las Nieves Huerta, M., Kiss, Z.G., Wallet, T.: On-the-fly model-based testing of web services with Jambition. In: Bruni, R., Wolf, K. (eds.) WS-FM 2008. LNCS, vol. 5387, pp. 143–157. Springer, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-01364-5_9
6. Frantzen, L., Tretmans, J., Willemse, T.A.C.: Test generation based on symbolic specifications. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 1–15. Springer, Heidelberg (2005)
7. Hessel, A., Larsen, K.G., Mikucionis, M., Nielsen, B., Pettersson, P., Skou, A.: Testing real-time systems using UPPAAL. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST. LNCS, vol. 4949, pp. 77–117. Springer, Heidelberg (2008), http://portal.acm.org/citation.cfm?id=1806209.1806212
8. Schmaltz, J., Tretmans, J.: On conformance testing for timed systems. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 250–264. Springer, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-85778-5_18
9. Schwarzl, C., Peischl, B.: Static- and dynamic consistency analysis of UML state chart models. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 151–165. Springer, Heidelberg (2010)
10. Schwarzl, C., Peischl, B.: Test sequence generation from communicating UML state charts: An industrial application of symbolic transition systems. In: Proceedings of the 2010 10th International Conference on Quality Software, QSIC 2010, pp. 122–131. IEEE Computer Society, Washington, DC (2010), http://dx.doi.org/10.1109/QSIC.2010.22
11. Tretmans, J.: Test generation with inputs, outputs, and quiescence. LNCS, pp. 127–146. Springer, Heidelberg (1996)
12. Veanes, M., Bjørner, N.: Alternating simulation and IOCO. In: Petrenko, A., Simão, A., Maldonado, J.C. (eds.) ICTSS 2010. LNCS, vol. 6435, pp. 47–62. Springer, Heidelberg (2010), http://portal.acm.org/citation.cfm?id=1928028.1928033

13. Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., Nachmanson, L.: Model-based testing of object-oriented reactive systems with Spec Explorer. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST. LNCS, vol. 4949, pp. 39–76. Springer, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-78917-8_2
14. Von Styp, S., Bohnenkamp, H., Schmaltz, J.: A conformance testing relation for symbolic timed automata. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 243–255. Springer, Heidelberg (2010), http://portal.acm.org/citation.cfm?id=1885174.1885193

An Empirical Study on Applying Anomaly Detection Technique to Detecting Software and Communication Failures in Mobile Data Communication Services

Hiroyuki Shinbo and Toru Hasegawa

KDDI R&D Laboratories Inc.,
2-1-15 Ohara, Fujimino-shi, Saitama 356-8502, Japan
{shinbo,hasegawa}@kddilabs.jp

Abstract. A mobile operator offers many mobile data communication services to its users, such as e-mail, Web browsing, company proprietary services. Although quick detection of communication and software failures are important to improve users' satisfaction, such a quick detection is difficult because the services are served by many servers, network nodes and mobile terminals. Thus we developed the anomaly detection technique for the mobile operator's network to detect anomalies caused by communication failures such as server and network halts. Our technique is based on the observation that users reconnect to servers many times when a communication failure occurs. It is useful not only to detect such communication failures, but also those which would be caused by software failures of mobile terminals and servers. This means that a mobile operator would be able to detect software failures missed at the testing period. In this paper, we empirically study how our technique is used to detect software failures of mobile terminals.

Keywords: Mobile data communication, anomaly detection, interoperability testing.

1 Introduction

Mobile terminals such as cellular and smart phones are owned by most people, and mobile data communication services such as e-mail and Web browsing services are becoming inevitable tools for social life. Since out-of-service has a serious impact on it, failures leading to out-of-service should occur as less frequently as possible. However, complete prevention is difficult due to a complicated system structure providing such a service. First, it is prone that software failures, i.e. bugs, are overlooked even by intensive tests because the system consists of various programs which run on various servers and mobile terminals. Especially, due to the competition in the mobile communication market, the number of mobile terminal models is becoming larger and a development period including a testing period is becoming shorter. Second, a service request is not always completed because of insufficient resources of servers and a network. It means that some service requests are thrown away by a congested server and that some messages are lost at a congested link. Of course, such an incomplete service request is also caused by hardware failures of servers and network equipments. Please note that the failures caused by problems on wireless links are out of scope in this paper. Our motivation is to detect failures that affect around a mobile data communication service, and it does not include detections of failures caused by each wireless environment of mobile terminal.

Quick detection of such failures in an operational network is a practical and promising approach. This approach is called anomaly detection [1] assuming that such a failure exhibits some unusual behavior (This unusual behavior is called anomaly.). We have developed an anomaly detection tool to detect how users abruptly change their behaviors [2] regarding a reconnect as the fact that a user's service request is not successfully completed due to some failure. The tool monitors service request messages on the network and calculates how many users reconnect to servers in every sample period, e.g., 180 seconds. Then, it detects an anomaly when a reconnecting terminal ratio (the ratio of terminals reconnecting to a server to all terminals) of current sample period abruptly changes from the previous one.

We applied this tool to a commercial mobile data communication system [2] and the results show that it can detect failures which result in an abrupt increase of reconnecting terminal ratio. Example failures are halts of components which simultaneously handle many sessions such as servers and network equipments. (A session is a communication path between a mobile terminal and an application server. It corresponds to a single service request.) These failures make many users simultaneously reconnect to servers. However, the tool may miss failures if only a small portion of users reconnects. We think that software failures would fall into this category of failures. (In this paper, we call a software failure as a "*bug*", and it does not mean problem locations within program codes in softwares.) For example, some bugs may happen in limited conditions. Other bugs exist in only software programs running on some specific model.

The goal of this paper is to empirically understand how such bugs are detected using our anomaly detection technique. Our insight is that if we focus on only mobile terminals or servers which have a bug, the reconnecting terminal ratio of them abruptly changes. For example, if a new release of software programs is affected by a bug, the reconnecting terminal ratio of mobile terminals which downloaded it would increase. Thus we calculate the ratio with some group of mobile terminals in order to know whether a bug which was overlooked in the testing period is detected or not.

The contributions of the paper are three-fold. First, we actually found a bug of mobile terminals as an anomaly by analyzing the log data of a mobile data communication system. This implies that appropriate grouping of mobile terminals enables to detect a bug of mobile terminals which have a common feature, e.g., the same release and the same application. Second, a threshold used for detecting anomaly is carefully determined. Third, our anomaly detection tool is so scalable that a few PCs (Personal Computer) with the tool can monitor a commercial mobile data communication system.

Although this paper does not deal with software testing techniques, the anomaly detection technique which this paper proposes is complementary to these techniques and plays an important role to detect bugs as soon as possible. This paper is organized as follows. Section 2 provides an overview of a mobile data communication system. Section 3 describes our anomaly detection tool and its algorithm. Section 4 describes how this tool is applied to detect anomalies caused by mobile terminals' bugs. Section 5 discusses the related work. Section 6 presents our conclusions.

2 Overview of Mobile Data Communication System

A mobile operator offers many mobile data communication services such as e-mail, Web browsing and company proprietary applications as shown in Fig.1. We call a mobile data communication service just as a "*service*" and a mobile data communication system just as a "*system*" in the rest of this paper.

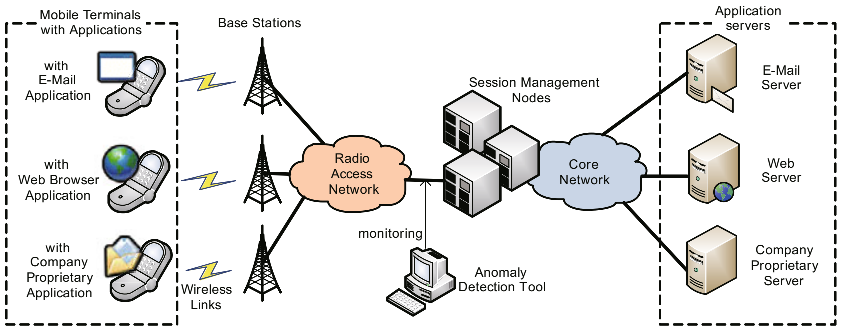


Fig. 1. Overview of a mobile data communication system

A system consists of the following components:

- *Applications* are software programs which provide a service to a user and they are running on a mobile terminal and an application server. A service is provided to a user by combination of applications on both the mobile terminal and server.
- *Mobile Terminals* are accommodated by a mobile operator's *Base Stations* and *Radio Access Network*.
- *Session Management Nodes* provide authorization and charging functions for services. After a mobile terminal is authorized by a session management node by sending a session creation request message, it can send a data request to an application server.
- *Application servers* are operated by either the mobile operator or third party application providers. The servers are accommodated by a *Core Network*.

– *Anomaly Detection Tool* detects anomalies based on a traffic monitoring, and it was developed by us. The detail will be described and discussed in Sect. 3 and later.

Figure 2 shows how a user gets a service in the following two steps.

(A) Session creation step
A *session* is a communication path between a mobile terminal and an application server. Before a user gets a service, a mobile terminal should send a *session creation request* message to a session management node. The mobile operator authorizes the user (or the mobile terminal) by validating this message. After the authorization succeeds, a *session creation complete* message is sent back and a session is created at the session management node. The mobile operator can charge user-data which is sent by the authorized mobile terminal based on the created session. The session creation request message includes the information of identifications of the mobile terminal and the requested service.

(B) Data communication step
After the session is created, *data communication* starts. The application on the mobile terminal sends *data request* messages to an application server and it sends back a data response message. This communication is a request-response style. In the case of a Web browsing service, they correspond to a HTTP Get request message and a HTTP Get reply message containing the data.

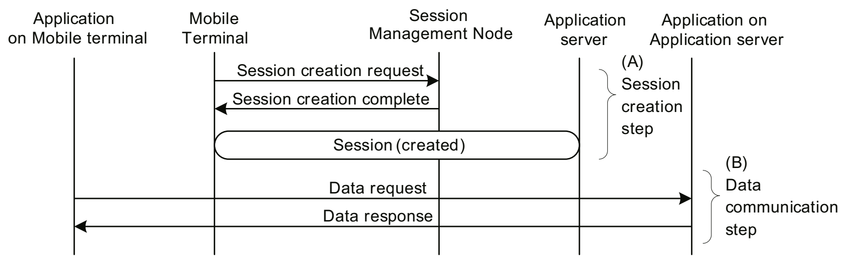

Application on Mobile terminal
Mobile Terminal
Session Management Node
Application server
Application on Application server
Session creation request
Session creation complete
Session (created)
Data request
Data response
(A) Session creation step
(B) Data communication step

Fig. 2. Steps to get a service

3 Methodology

3.1 Principles of Anomaly Detection

An anomaly is an unusual behavior of some component of the system which is likely to be caused by a failure. Our motivation is to use the detected anomaly to identify a failure causing it, thus we summarize failures before defining the anomaly. Table 1 lists possible failures for individual components in the system, such as mobile terminals and session management nodes. Please note that all failures are not listed and that these failures are typical ones. Possible failures are categorized to the steps in Fig.2:

- (A) Session creation step: A failure of this step results in session not being created.
- (B) Data communication step: A failure of this step results in data request not being completed, even if a session is successfully created.

Table 1. Possible failures of the steps (A) and (B)

Step	Component	Possible failures
(A)	Mobile terminals	Hardware failure, Protocol software program bug
(B)		Application software program bug
(A)	Session management nodes	Hardware failure, Node software program bug, Overload
(B)	Application servers	Hardware failure, Application software program bug, Overload
(A)	Radio access network	Network congestion, Network equipment (e.g. Ethernet switches, Routers) failure, Link failure
(B)	Core network	

Usually, an abrupt change of the number of messages (packets) is defined as an anomaly in the Internet [1]. Thus in order to identify which component is in failure, all messages sent by all components such as mobile terminals, session management nodes and application servers should be monitored. It means that many tools (equipments with network interface cards) capturing messages should be set at many links to which these components are connected. Although the method of monitoring all messages is useful to precisely identify the failed component, applying many tools is too expensive to be used commercially.

On the contrary, we use as small number of tools as possible. As described in Sect. 3.2, a few anomaly detection tools are set at the link connected to session management nodes as shown in Fig. 1. The anomaly detection tool captures session creation request messages. Apparently, failures at the session creation step (A) are easily detected because the tool can monitor failed session creation request messages. On the contrary, how anomalies caused by failed data requests are detected is an important issue.

We focus on the observation that a user (a mobile terminal) reconnects to an application server after a mobile data communication service is not successfully completed [3]. It means that since a user should create a session before getting a service from an application server, the user re-sends a session creation request message to a session management node again. Such a re-sent session creation request message is regarded as the fact that a user reconnects to an application server. Thus we define *reconnections* as mobile terminals which re-send session creation request messages for reconnecting to an application server, and an *anomaly* as an abrupt increase (change) of reconnections. (The anomaly is precisely defined in Sect. 3.3.) This enables to detect a failure of a data request to an application server at the data communication step (B) without capturing data request and response messages. As far as we know, only our tool uses such reconnections to detect anomalies in other communication systems.

This definition has two advantages. First, capturing only session creation request messages requires less computing power. It is more scalable than capturing all messages sent by all components such as mobile terminals, session management nodes and application servers. Second, this apparently reflects customers' satisfaction.

3.2 Anomaly Detection Tool

- The anomaly detection tool takes a traffic monitoring approach rather than a probing approach [1].
    - In a probing approach, a probing tool sends test messages to individual components in the system and thus it clearly pinpoints a failure of each component. However, it is time-consuming because a mobile data communication system consists a number of components, e.g., more than hundreds of servers.
    - A traffic monitoring approach enables to quickly detect anomalies because captured packets (messages) are immediately analyzed just after capturing them. In addition, this traffic monitoring avoids imposing load on servers and network nodes in the system.
- The anomaly detection tool is a software program running on a PC (Personal Computer) with a network interface card. The PC with the implemented tool is set at a link of a session management node in a radio access network (described in Fig.1), and it is used to capture only session creation request messages transferred between mobile terminals and session management nodes. The implemented tool running on a PC with Intel CoreDuo 2.0GHz CPU and 2G bytes memory can simultaneously process more than 60,000 session creation request messages per a minute. Thus a few anomaly detection tools are enough to monitor all sessions in a commercial mobile data communication system [2].

3.3 Anomaly Detection Algorithm

This section precisely defines the metric for the anomaly detection. We defined the anomaly as an abrupt change of reconnections, the metric is based on how many mobile terminals re-send session creation request message. Before defining it, we define how many times a mobile terminal sends session creation request messages in a sample period a "*session count*". Since a session creation request message does not explicitly specifies that it is a reconnection, we regard those subsequent to the first request as reconnections. That is, if the session count is 2, the number of reconnections is 1.

An important issue is whether sending of these session creation request messages are really reconnections. Thus we carefully determine how long the sample period is so that no new session creation request messages exist in the sample period. In addition, the sample period should be determined as a small value as possible for quickly anomaly detections.

We collected the 2 month's log of session creation request messages in the system and investigated session intervals of each mobile terminal without failures. A session interval is defined as an interval of sending successive session creation request messages. Then, a cumulative frequency distribution of session intervals is created. We see that 90% of intervals are more than 180 seconds. Then we determine 180 seconds as the sample period. Thus in a 180 seconds sample period, about 90% of session creation request messages might not be reconnections. In this case, about 10% of session creation request messages might be reconnections. We investigated the number of reconnections in a 180 seconds sample period and obtained that about 90% of the number of reconnections was once. (i.e., the session count is 2.)

Another important issue is for what group of mobile terminals and servers the metric is calculated. A mobile operator or a third-party application provider is responsible for each service. Thus the metric is calculated from session creation request messages to the same server which corresponds to a service. Before defining the metric, we calculate a distribution of how many terminals connects to a server the "*session count*" times. We define $v_m[n]$ as the number of mobile terminals which have the session count n at the sample period m. Figure 3 shows how to get a session count for mobile terminals. In Fig.3, at the sample period m, the five mobile terminals *MT-A* to *MT-E* send session creation request messages. 2 (*MT-B* and *MT-E*), 2 (*MT-C* and *MT-D*) and 1 (*MT-A*) terminals send 1, 2, 3 session creation request message(s), respectively. That is, the results of $v_m[n]$ are $v_m[1] = 2$, $v_m[2] = 2$ and $v_m[3] = 1$.

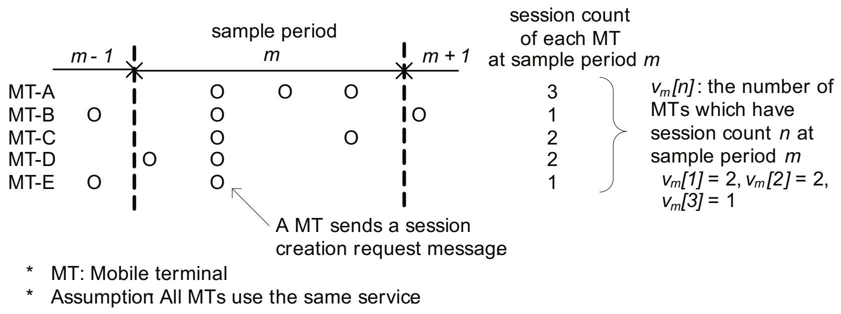

session count of each MT at sample period m
sample period m
m - 1
m + 1
MT-A
MT-B
MT-C
MT-D
MT-E
3
1
2
2
1
v_m[n]: the number of MTs which have session count n at sample period m
v_m[1] = 2, v_m[2] = 2, v_m[3] = 1
A MT sends a session creation request message
* MT: Mobile terminal
* Assumption All MTs use the same service

Fig. 3. How to get a session count

The anomaly detection algorithm focuses on mobile terminals which reconnect once and it means that values of $v_m[2]$ are used. Please note that since about 90% of the number of reconnections was once in a 180 seconds sample period as described at the above, in the rest of paper, we use 2 as a session count n and omit it from variable names. The other session counts or the sum of a several session count values can be chosen.

After this, we will explain how to detect an anomaly based on $v_m[2]$. Precisely, it uses how x_m is changed from that of the previous sample period $m-1$. x_m is the normalized value of $v_m[2]$ with respect to sum of $v_m[n]$ as shown by Equation(1). In Fig.3, x_m is calculated as $v_m[2]/(v_m[1]+v_m[2]+v_m[3]) = 0.4$. We call x_m as the "*reconnecting terminal ratio*" and it is the metric for anomaly detection. A reason why the normalized value used for the metric is that it is not sensitive to a change of the number of total sessions.

Before explaining how to use x_m for an anomaly detection, we defines two values y_m and y'_m. y_m in Equation(2) is the exponential average [4] of x_m, and y'_m in Equation(3) is the square of exponential average of x_m. The α is used for calculating exponential average in Equation(2) and (3). To decide the α, we need to decide the time-window tw and the sample period p. The time-window means that x_m values before tw get lost in oblivion, and we decide one day as tw (one day equals to 86,400 seconds). p is defined as 180 seconds from the beginning of this section, and thus α is calculated as around 0.002 ($= p/tw$).

Equation(4), (5), (6) and (7) are used for the anomaly detection based on x_m. We define the condition of an anomaly detection as that a difference g_m is more than a threshold t_m shown as Equation(7).

- g_m is a difference between the current x_m and the exponential average of y_{m-1}.
- We choose the threshold t_m based on the standard deviation σ_m. Equation(5) shows that σ_m is calculated using the exponential average y_m and y'_m calculated by Equation(2) and (3). The threshold t_m is calculated by k times as the standard deviation σ_m shown as Equation(6). The standard deviation σ_m means a possible range of a difference between x_m and the exponential average y_{m-1} in normal case at the sample period m. Our algorithm detects an anomaly when the difference g_m is more than k times of the possible range.

$$x_m = \frac{v_m[2]}{\sum_{i=1,\infty} v_m[i]} \tag{1}$$

$$\begin{aligned} y_m &= \alpha \times x_m + (1-\alpha) \times y_{m-1} && (2)\\ y'_m &= \alpha \times x_m^2 + (1-\alpha) \times y'_{m-1} && (3)\\ &\text{where} \quad \alpha = p/tw \end{aligned}$$

$$\begin{aligned} g_m &= abs(x_m - y_{m-1}) && (4)\\ \sigma_m &= \sqrt{(y'_m - y_m^2)} && (5)\\ t_m &= k \times \sigma_{m-1} && (6)\\ g_m &> t_m && (7) \end{aligned}$$

Figure 4 shows how anomalies are detected by Equation(7). It shows a time series of 600 samples of g_m and each circle is g_m in Equation(4) at each sample

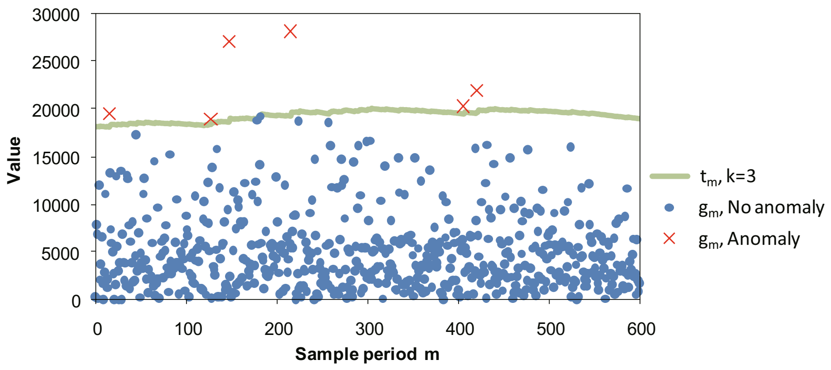


Fig. 4. Anomaly detection example

period. It also shows the curve which plots values of t_m in Equation(6) where k is 3. Samples which are over the curve are regarded as anomalies, and they are marked by cross marks in Fig.4. We will discuss how k is determined in Sect. 4 based on the log data.

3.4 Applying Anomaly Detection Tool to Detecting Bugs

It is relatively easy to detect anomalies caused by failures of session management nodes, servers, a radio access network and a core network because these failures result in many session creation failures. However, we consider that detecting anomalies caused by a bug (software failure) of mobile terminals is not easy because all mobile terminals are not always affected by this bug. For example, when some bug is injected to an application software program which is installed in a new release of mobile terminals, only such mobile terminals have the bug. Another example is that all users do not always use an application software program with a bug. In these cases, only some portion of terminals exhibit unusual behaviors. Monitoring all session creation request messages results in a small change of the reconnecting terminal ratio, i.e., x_m.

Thus it is necessary to group mobile terminals which are affected by the same bug and then to calculate a change of the reconnecting terminal ratios for the groups of mobile terminals. However, it is not clear how mobile terminals are grouped. In Sect. 4, we will study empirically what kinds of the groups are useful by analyzing 6 month log data of session creation request messages.

4 Detection Examples of Software Failures

We analyzed the 6 month log data of session creation request messages in a mobile data communication system. The system which was targeted by our log data analysis handled over 1.5 million mobile terminals and over 100 mobile terminal models. A part of the mobile terminals were connected to the system at the same time, and the mobile data communication services were requested randomly. We successfully detected anomalies caused by failures of session management nodes and a radio access network. The details are described in [2]. In this section, we show how a bug of an application software program by a third-party application provider on a mobile terminal is detected by our algorithm. The bug makes a mobile terminal re-send a session creation request message in some conditions. This section describes how such a bug is detected using our anomaly detection technique.

4.1 How k for Threshold t_m Is Determined

To detect anomalies it is important how k for threshold t_m in Equation(6) is determined. If k is set to a large value, some anomalies may be missed. On the contrary, if k is set to a small value, many fake anomalies which are not caused by failures are erroneously detected. The anomaly detection tool is an operations tool and thus anomalies are reported as "*alarms*" to operators. It is important not to report many fake alarms. We set a goal that the number of fake alarms is less than 0.2 percent. Since one day consists of 480 sample periods, about one fake alarm would be reported in average per day.

To determine such k, we investigated the relationship between g_m and t_m with various values of k. Figure 5 shows one-day result of g_m and t_m with k=1.5, 2, 2.5, 3, 3.5 and 4 from the log data. The curves in Fig.5 correspond to the thresholds t_m at all sample periods with k=1.5, 2, 2.5, 3, 3.5 and 4. Each circle g_m in Equation(6) corresponds to the difference between the reconnecting terminal ratio and its exponential average at each sample period. If a circle is over the curve, it is an anomaly. By counting the number of such circles for the 6 month log data, we choose 3 as k such that anomalies are detected at about one percent of total sample periods.

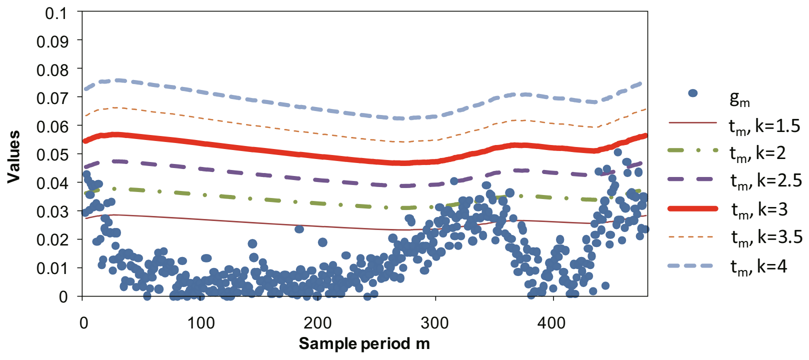


Fig. 5. One-day result of g_m and t_m with various k

4.2 How the Bug Is Detected Based on Reconnections

We apply our algorithm into the 6 month log data. We calculate the reconnecting terminal ratio x_m for all mobile terminals in the system. The anomaly detection detected anomalies as shown in Fig.6(C). It shows the time-series of g_m (calculated from x_m by Equation(4)) and the curve of threshold t_m with k=3 during 40 sample periods. At the sample periods 4 and 24 in Fig.6, values of g_m are over the threshold t_m and these points are detected as anomalies.

However, it was not unknown which component's failure caused these anomalies because there were many candidates of failures causing these anomalies. Although we checked logs of session management nodes, application servers and equipments in the radio access and core networks, no failure was found. Thus we suspected that mobile terminals would be affected by a bug of an application software program on mobile terminals. At this time, since we heard a new release provided by a third-party vendor for mobile terminals, we assumed that this release might have a bug. We validated this assumption in the following steps: First, we found candidates of mobile terminals which might have downloaded this release. Please note that some of these candidates have not downloaded it yet. Second, mobile terminals are divided into two groups: (*Group A*) the group of such candidate mobile terminals, and (*Group B*) the group of other mobile terminals.

Figure 6(A) and (B) show the results for these two groups. Although anomalies are detected in *Group A*, no anomaly is detected in *Group B*. As the result, we consider that this anomaly would be affected by this release and then actually found the bug in this release of application software program.

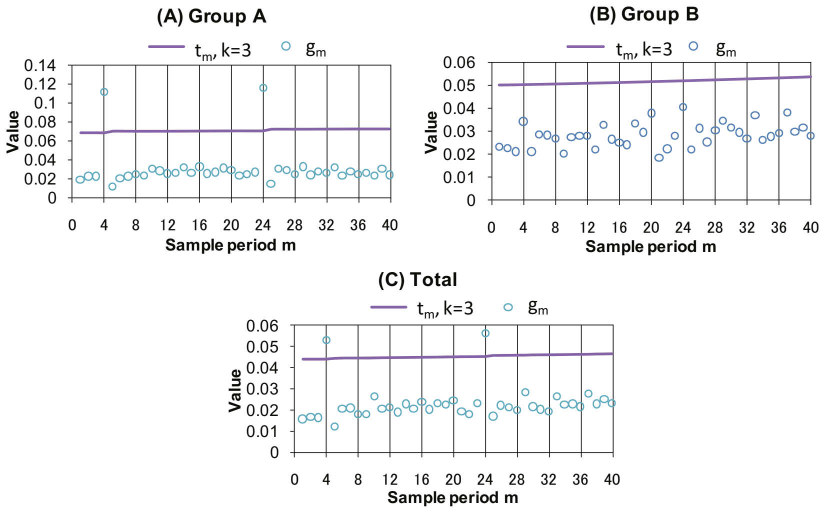


Fig. 6. Time-series on g_m and t_m with k=3

This grouping was useful to identify the bug, but this fact implies that the anomaly could be detected earlier if we monitored only mobile terminals of this group. In this empirical study, some days passed since the new release of the software program had been announced. It means that before starting to collect the log data, the release was announced and many mobile terminals already downloaded it. If we monitored this group of mobile terminals, this anomaly would be detected earlier. We consider that the monitoring for anomaly detection per such groups of mobile terminals is important to quickly detect it.

To validate the above hypothesis, since we do not have the log data before the new release date, we investigated the relationship between g_m and t_m with k=3 for all mobile terminals and the group of them with the bugs (*Group A* and *Total* is the same as Fig. 6) after several days of releasing the patch (the program of fixing the bug) for the software program. Our assumption is that the number of mobile terminals with the bug was decreased since some mobile terminals downloaded the patch to fix the bug and that in this case, the anomaly is detected in *Group A*, but it is not detected in *Total*. If this assumption is correct, we can detect such an anomaly by focusing on a group of mobile terminals with a bug. Figure 7 shows the graphs of the relationship after several days of releasing the patch. Since the circle as g_m over the curve as t_m with k=3, anomalies occur at the sample period of 8 and 24 in *Group A* of Fig. 7(A). On the contrary, in *Total* of Fig. 7(B), there is no anomaly.

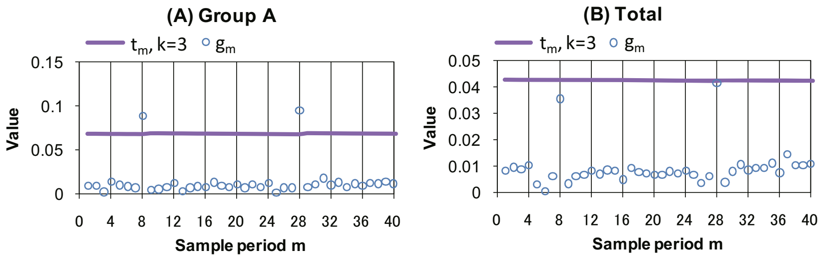


Fig. 7. Relationship between g_m and t_m with k=3 after several days of releasing the patch

5 Related Work

There are two approaches for anomaly detection in communication systems [1]: traffic monitoring and probing approaches. Our anomaly detection tool takes the traffic monitoring approach. In this section, we discuss about the major related work [1].

5.1 Traffic Monitoring

Traffic monitoring is a passive scheme whereby messages (packets) are monitored (observed) to detect anomalies.

Statistics-based monitoring. Statistics-based monitoring means that a management node collects the statistics on packet transmission from nodes such as layer2 switches and routers by communicating with the nodes (e.g., [5]). Such communication overheads between the management node and the other nodes are not negligible. Since the system consists of a number of nodes, i.e., application servers, session management nodes and so on, the overhead would be too heady and time-consuming.

Packet capturing-based monitoring. Packet capturing-based monitoring means that some equipment captures transmitted packets and obtains packet transmission statistics. However, since a huge number of packets (dozens of gigabits per day) are transmitted on a commercial mobile core network, this method has a few disadvantages: Many capturing points are needed to collect transmitted packets. Besides, the method of trajectory sampling [6] can decrease the number of packets that need to be captured. However, such a sampling would be prone to miss anomalies.

On the contrary, our tool only captures session creation request messages (packets) which contain user's requested services. This means that the number of captured packets less than the number of all transmitted packets. All session creation request messages can be obtained at a few (maybe one) capture points with a few PCs.

5.2 Probing

Probing is an active scheme whereby probes which check whether equipments are not in failure are sent to the equipments.

Internal probing. Probe programs are installed into equipments such as communication nodes or application servers, and they check whether the equipments are not in failure [7]. However, such probe programs are difficult to install if the service nodes are not in networks administrated by operators. In our system, they cannot be installed on third-party application providers' servers.

External probing. A probe tool checks equipments by actually sending test messages to servers [8,9]. This external probing is not scalable for a large-scale network. Many hours would be needed to check all components such as application servers, session management nodes and mobile terminals.

6 Conclusion

In this paper, we applied an anomaly detection technique to detect anomalies caused by mobile terminal software failures (bugs). The anomaly detection technique focuses on user's behavior to reconnect a service and it detects anomalies based on how many mobile terminals re-send session creation request messages.

Although this is a light-weight mechanism, it enables to quickly (within a few minutes) detect anomalies caused by not only communication failures, but also bugs. This empirical study shows that a bug of mobile terminal was actually detected. We consider that anomaly detection techniques are useful to detect bugs which were overlooked during a testing period. As far as we know, this paper is one of the first papers which actually detected a bug in a commercial environment. Although this paper does not deal with software testing techniques, an anomaly detection technique is complementary to these techniques and plays an important role to detect bugs as soon as possible.

In the future, we try a remaining issue about how to find such a group of mobile terminals which are affected by the same bug. We also consider how to apply our anomaly detection algorithm to other systems. Since our algorithm can be applied to session-request based system, for example, we may apply it easily to the IP Multimedia Subsystem (IMS) [10].

References

1. Marina, T., Chuanyi, J.: Anomaly Detection in IP Networks. IEEE Transactions on Signal Processing 51(8), 2191–2204 (2003)
2. Hiroyuki, S., Satoshi, K., Hideyuki, K., Teruyuki, H., Hidetoshi, Y.: A scheme for detecting communication abnormality in mobile core networks based on user behavior. In: The 12th International Symposium on Wireless Personal Multimedia Communications, WPMC 2009 (2009)
3. Sumaru, N.: Experimental Study on User Behavior Analysis by Keylogs of Cellular Phone. In: IEICE 5th Workshop on Brain Communication (2008) (in Japanese)
4. NIST/SEMATECH: e-Handbook of Statistical Methods, Sect.6.4.3.1 (2006), http://www.itl.nist.gov/div898/handbook/
5. Case, J., Fedor, M., Schoffstall, M., Davin, J.: Simple Network Management Protocol (SNMP). RFC1157 (1990)
6. Duffield, N.G., Gerber, A., Grossglauser, M.: Trajectory Engine: A Backend for Trajectory Sampling. In: IEEE Network Operations and Management Symposium 2002 (2002)
7. Matthias, W., Peter, U., Xavier, D.: An SNMP based failure detection service. In: The Proceedings of 25th IEEE Symposium on Reliable Distributed Systems, SRDS 2006 (2006)
8. Irina, R., Mark, B., Natalia, O., Sheng, M., Genadv, G.: Real-time Problem Determination in Distributed Systems using Active Probing. In: IEEE/IFIP Network Operations and Management Symposium (NOMS 2004), vol. 1, pp. 133–146 (2004)
9. Wolfgang, B.: Nagios, Second Edition, System and Network Monitoring. No Starch Press (2008)
10. Gonzalo, C., Miguel-Angel, G.: The 3G IP Multimedia Subsystem: Merging the Internet and the Cellular Worlds. Wiley, Chichester (2004)

Assessing Test Adequacy for Black-Box Systems without Specifications

Neil Walkinshaw

Department of Computer Science,
The University of Leicester
nw91@le.ac.uk

Abstract. Testing a black-box system without recourse to a specification is difficult, because there is no basis for estimating how many tests will be required, or to assess how complete a given test set is. Several researchers have noted that there is a duality between these testing problems and the problem of inductive inference (learning a model of a hidden system from a given set of examples). It is impossible to tell how many examples will be required to infer an accurate model, and there is no basis for telling how complete a given set of examples is. These issues have been addressed in the domain of inductive inference by developing statistical techniques, where the accuracy of an inferred model is subject to a tolerable degree of error. This paper explores the application of these techniques to assess test sets of black-box systems. It shows how they can be used to reason in a statistically justified manner about the number of tests required to fully exercise a system without a specification, and how to provide a valid adequacy measure for black-box test sets in an applied context.

1 Introduction

When do we know that a test set is adequate? How do we know that it is sufficiently rigorous for its execution to highlight the presence of any faults? If it is not adequate, how many more tests will we need to generate to achieve a requisite level of adequacy? These questions are fundamental to software testing.

Although numerous approaches are routinely used to assess test adequacy (e.g. code or model coverage), these have significant drawbacks. Code-based coverage has been shown to be an unconvincing fault predictor (c.f. work by Nagappan *et al.* [16]). Model-based coverage on the other hand makes the restrictive assumption that there exists a complete and up-to-date model of the system in question.

Over the past 30 years, a different approach to test adequacy has emerged that attempts to circumvent the weaknesses of traditional techniques. This approach exploits an intuitive relationship between the seemingly unrelated fields of inductive inference and software testing. The idea is to treat the two approaches as two sides of the same coin; both are dealing with a system that is unknown; testing elicits behaviour, and inductive inference reasons about its behaviour by inferring models. From the perspective of test adequacy, there is a direct link between the accuracy of an inferred model and the adequacy of the test set that was used to infer it [32,31]. If a model can be shown to be accurate, the underlying test set evidently exercises the system in a sufficiently extensive manner.

There has been a recent resurgence in techniques that expolit this relationship [3,5,6,8,12,17,19,22,23,26,28,29,30] by inferring models from test sets, and in some cases using these models to elicit further test cases. However, these techniques tend to suffer from two problems: (1) there is no means of predicting how many tests would be required to arrive at an adequate test set and (2) given a partial test set, there is no basis for gauging how close it is to being adequate.

Problems that are analogous to these have been the subject of much research in the context of inductive inference [4,9,24]. These techniques, which are largely based on probabilistic reasoning, are especially interesting from a testing perspective because they offer potential solutions to these testing problems. This was the subject of a reasonably concentrated amount of research in the eighties and nineties [6,20,21,30,32,31], but has not been revisited in the light of the aforementioned surge in popularity of learning-based testing techniques.

This paper investigates the application of these techniques in a realistic testing context. The key contributions are as follows:

1. An implementation of Valiant's PAC framework [24] in a testing context. This enables the probabilistic specification of what would be considered to be an adequate test set in terms of the accuracy of the model that is inferred from it.
2. The application of PAC-based probabilistic techniques [9,4] to estimate lower bounds on the number of tests required for a test set of a black-box SUT to be adequate.
   – An applied demonstration of how to apply these approaches to SUTs that may be modelled by Finite State Machines.
3. A practical demonstration of the use of the PAC framework in an applied setting to quantify the adequacy of test sets with respect to a small black-box simulator of an SSH client. The entire infrastructure used for experimentation have been made openly available.

Section 2 will present the background to combining inductive inference with testing. Section 3 will show how the PAC framework can be reinterpreted in a testing context. Section 4 will show how this can be used to estimate the required size of an adequate test set. Section 5 shows how the PAC framework can be used in a practical context to estimate the adequacy of existing test sets. Section 6 will discuss related work, and section 7 will present the conclusions and discuss future work.

2 Background

2.1 The Setting

This paper considers a setting where the SUT is a black-box, but where there is no usable specification to generate tests from. In this context, a test case is simply an input to the SUT without an expected output. An *adequate* test set [30,31,32] will exercise every essential element of functionality in the system, and in doing so trigger any obvious faults such as a crash or an uncaught exception. This setting is realistic. The source code of a system, even if it is available, is only effective to a limited extent when as a basis for test set generation[16]. Although there are several sophisticated model-based testing techniques [13], developers rarely produce and maintain models that are sufficiently accurate and up-to-date to serve as a suitable basis for test generation.

The task of generating an adequate test set in this setting is seemingly impossible. Without a specification or source code there is no means by which to assess how complete the test set is. There is also no coverage-like metric to serve as a basis for homing-in on an adequate test set.

2.2 Testing with Inductive Inference

Over the past thirty years one approach has emerged that can (at least in principle) assess test sets in the above setting. Instead of generating a test set in a single step and subsequently executing it, the idea is to generate test sets by experimentation. The outputs produced by an initial test set are observed and are used to infer a hypothetical model of system behaviour. Depending on the approach, this may then be used to drive the generation of further test sets, or to assess the adequacy of the original test set by somehow comparing the model with the SUT.

Inductive inference is a means of reasoning about a black-box SUT in terms of its observable behaviour. If a test set is comprehensive enough to enable the inference of an accurate model, then it can be deemed to be adequate [30,31]. The relationship between inductive inference and software testing was first explored by Weyuker in 1983 [30]. Since then a large number of techniques have been developed that adopt different types of model inference. Initially, Weyuker's work and subsequent work by Bergadano *et al.* [3,30] focussed on synthesised programs. Since then however, similar approaches have been based upon Artificial Neural Nets [12,22], invariants [8], decision trees [5] and deterministic finite state automata [2,19,23,26,28,29].

2.3 Practical Problems in Establishing Test Adequacy

The use of inductive inference provides a plausible method for assessing the adequacy of test sets in a meaningful way (i.e. with respect to the behaviour they elicit). However, from a practical point of view, there remain two important barriers to its widespread use:

1. **Predicting expense:** There is no reliable basis for estimating how expensive the testing process will be, i.e. how many tests will be required to produce an adequate test set. This is a fundamental testing problem and is not restricted to testing techniques that incorporate inductive inference. In the context of testing techniques that use inductive inference, it is akin to stating that it is not known how many examples will be required to infer an accurate model.
2. **Quantifying adequacy:** Current testing approaches that revolve around inductive inference implicitly assume that an inferred model must be accurate before the test set can be considered adequate. Their feedback is binary: adequate or inadequate. This is impractical for two reasons. Firstly, there is no feedback to provide any insights about how close the test set is to being adequate, or determining whether one test set is better than an other. Secondly, most inductive inference algorithms are prone to making mistakes and can at best infer a model that is approximate even if the test set itself is adequate. However, there is no way to account for this by allowing for a given degree of error.

3 Inductive Inference and Testing in a Probably Approximately Correct Setting

In the context of machine learning, the area that seeks to address such problems is generally referred to as *Computational Learning Theory* (also *Statistical Learning Theory*). Given the widely acknowledged link between inductive inference and testing, it seems intuitive that some of the Computational Learning Theory principles that have been successfully applied in inductive inference should be readily applicable in a testing context. This section sets the foundations for this by recoding a framework by Valiant [24], which has become widely known as the *Probably Approximately Correct* (PAC) framework, into the testing setting.

3.1 The PAC Framework

The PAC framework [24] describes a basic learning setting, where the key factors that determine the success of a learning outcome are characterised in probabilistic terms. As a consequence, if it can be shown that a specific type of learner fits this setting, important characteristics such as its accuracy and expense with respect to different sample sizes can be reasoned about probabilistically. The specific elements of the framework are illustrated here with respect to the example problem of learning a deterministic finite state machine from sample sequences (to save space, we presume the conventional definition and notation [27]). Much of the notation used here to describe the key PAC concepts stems from Mitchell's introduction to PAC [14].

The PAC setting assumes that there is some *instance space* X. As an example, if we are inferring a finite state machine with an alphabet Σ, X could be the set of all words in Σ^*. A *concept class* C is a set of concepts over X, so in our case case it the set of all deterministic finite state machines that can accept and reject words in X. A *concept* $c \subset X$ corresponds to a specific target within C to be inferred (in our case it is the finite state machine that accepts a specific subset of words in Σ^*). Given some element x (in our case a word), $c(x) = 0$ or 1, depending on whether it belongs to the target concept. It is assumed that there is some selection procedure $EX(c, \mathcal{D})$ that randomly selects elements in X following some static distribution $\mathcal{D}$ (we do not need to know this distribution, but it must not change).

The basic learning scenario is that some learner is given a set of examples as selected by $EX(c, D)$. After a while it will produce a hypothesis h. The error rate of h subject to distribution $\mathcal{D}$ ($error_{\mathcal{D}}(h)$) can be established with respect to a further 'test' sample from $EX(c, \mathcal{D})$. This represents the probability that h will misclassify one of the test samples, i.e. $error_{\mathcal{D}}(h) \equiv Pr_{x\in\mathcal{D}}[c(x) \neq h(x)]$.

In most practical circumstances, a learner that has to guess a model given only a finite set of samples is susceptible to making a mistake. Furthermore, given that the samples are selected randomly, its performance might not always be consistent; certain input samples could happen to suffice for it to arrive at an accurate model, whereas others could miss out the crucial information required for it to do so. To account for this, the PAC framework enables us to explicitly specify a limit on (a) the extent to which an inferred model is allowed to be erroneous to still be considered approximately accurate, and (b) the probability with which it will infer an approximate model. The error parameter ϵ that puts an upper limit on the probability that an inferred model may mis-classify a given input. The δ parameter denotes an upper bound on the probability of a failure to infer a model (within the error bounds).

3.2 A PAC-Compatible Testing Framework

Figure 3.2 shows how the inductive inference and testing processes can fit into the PAC framework [31,26]. The arcs are numbered to indicate the flow of events. The test generator produces tests according to some fixed distribution $\mathcal{D}$ that are executed on the SUT c. With respect to the conventional PAC framework they combine to perform the function of $EX(c, \mathcal{D})$.

The process starts with the generation of a test set A by the test generator (this is what we are assessing for adequacy). These are executed on the SUT, the executions are recorded and supplied to the inference tool. This infers a hypothetical test oracle. Now, the test generator supplies a further test set B, and the user supplies some acceptable error bounds ϵ and δ. The observations of test set B are then compared against the expected observations from the model to compute $error_{\mathcal{D}}(h)$. If this is smaller than ϵ, the model inferred by test set A can be deemed to be *approximately accurate* (i.e. the test set can be deemed to be *approximately adequate*).

The δ parameter is of use if we want to make broader statements about the effectiveness of the combination of learner and test generator. By running multiple experiments, we can count the proportion of times that the test set is approximately adequate for the given SUT. If, over a number of experiments, this

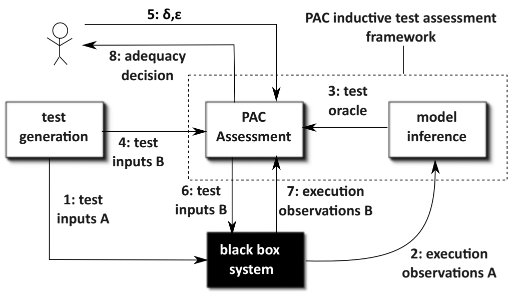


Fig. 1. Inductive testing with the PAC framework

proportion is greater than or equal to $1 - \delta$, it becomes possible to state that, in general, the test generator produces test sets that are *probably approximately adequate* (to paraphrase the term 'probably approximately correct', that would apply to the models inferred by the inference technique in a traditional PAC setting).

4 Estimating Test Set Size

Given that the SUT is a black-box, and that we can only reason about it by experimenting with it, it is seemingly impossible to ascertain a-priori how many test sets will be required to constitute an adequate test set. Surprisingly work that builds on the PAC framework *does* enable us to obtain a bound on the number of tests (if we make certain assumptions about the SUT, which are discussed later). By assuming that the set of test cases is selected randomly from some fixed distribution, it becomes possible to make a probabilistic argument the number of test cases required to arrive at a point where any model that is consistent with the test sets must be sufficiently accurate.

The approach relies on the ability to characterise the complexity of the learning task. In the context of PAC-learning Haussler [9] describes two approaches, each of which is based on a different characterisation of complexity. One of them assumes that it is possible to place an absolute bound the number of possible hypotheses that could be produced by a learner (known as the Version Space), whilst the other assumes that it is possible to place a bound on the internal complexity of the hypothesis space (known as the VC Dimension). These two approaches will be presented in this section, followed by a demonstration of how each of them can be applied to reason about the size of an adequate test set for a black-box SUT that could be modelled by a deterministic finite state machine.

4.1 Bounding Test Set Size with Version Spaces

The question of how many tests belong to an adequate test set is akin to the question of how many tests would be required to ensure that an accurate (within the limits of ϵ and δ) model can be inferred. To establish this, Haussler's Version-Space based approach [9] estimates a lower bound on the number of tests that would be required to ensure that all consistent hypotheses that could possibly be inferred from the test set fall within the acceptable error bounds.

To reason about the possible range of hypotheses, Haussler uses Mitchell's notion of *version spaces* [15]. In the testing context, a test set D consists of inputs x and their expected outputs $c(x)$. Version spaces are defined as follows (using the definition from Mitchell's book [14]):

$$VS_{H,D} = \{h \in H | (\forall \langle x, c(x) \rangle \in D)(h(x) = c(x))\}$$

Haussler defines the the version space as *ϵ-exhausted* if all of the hypotheses that can be constructed from D have an error below ϵ, with respect to any distribution $\mathcal{D}$. More formally [14]:

$$(\forall h \in VS_{H,D}) error_{\mathcal{D}}(h) < \epsilon$$

The number of elements in D that is required to ϵ-exhaust $VS_{H,D}$ is exponential or infinite in the worst case. However, given that we are using the PAC setting, this is not the case, we know that the elements in D are selected independently and at random, the number number of tests m that are required to ϵ-exhaust $VS_{H,D}$ is considerably improved [9]. If $VS_{H,D}$ is finite, it becomes possible to establish a lower bound on m. Assuming that set D is constructed by $m \geq 1$ independent, random test cases, he shows that the probability that $VS_{H,D}$ is *not* ϵ-exhausted is less than $|H|e^{-\epsilon m}$ (see Haussler's paper for the proof [9]). Within the PAC framework, this probability should be less than or equal to δ. This can be factored in to the above probability, and rearranged to impose a lower bound on m, the number of test cases that constitute D:

$$m \geq \frac{(ln|VS_{H,D}| + ln(1/\delta)}{\epsilon} \tag{1}$$

The number of required tests m grows linearly in $1/\epsilon$, it grows logarithmically with $1/\delta$, and it grows logarithmically in the size of $VS_{H,D}$ [9].

4.2 Bounding Test Set Size with the Vapnik-Chervonenkis Dimension

Depending on the SUT, it may be impossible to easily impose an upper limit on the size of $VS_{H,D}$ (e.g. the SUT could be a function that computes a real number). For this case, Haussler proposed an alternative approach to bound the test set size that does not rely on the size of $VS_{H,D}$, but uses a measure of complexity of H known as the Vapnik-Chervonenkis or VC dimension [25] (though this is not necessarily finite either).

To define the notion of a VC dimension, it is necessary to first introduce the notions of *dichotomies*, and *shattering* (see Haussler's paper [9] for details). If I is some subset of the instance space X, then an hypothesis $h \in H$ induces a *dichotomy* on I by dividing I into those examples that are classified as belonging to h, and those that are not. $\Pi_H(m)$ denotes the maximum number of dichotomies that can be induced by H on any set of m instances.

If H induces all possible 2^I dichotomies of I, then H *shatters* I. The VC dimension of H $VC(H)$ is the cardinality of the largest subset of X that is shattered by H. Equivalently, it is the largest m such that $\Pi_H(m) = 2^m$ [9].

Using a proof that is analogous to the one used in the version space approach, Haussler shows that the probability that $VS_{H,D}$ is not ϵ-exhausted is less than $2\Pi_H(2m)2^{-\epsilon m/2}$. From this, and by incorporating further results from Blumer *et al.* [4], it can be rearranged to yield the lower bound on m:

$$m \geq \frac{4log_2(2/\delta) + 8VC(H)log_2(13/\epsilon)}{\epsilon} \tag{2}$$

Mitchell [14] notes that this measure will often produce tighter bounds than the equivalent estimation using the Version Space approach. The bound m grows logarithmically in $1/\delta$, but grows log linear in $1/\epsilon$. As will be shown in section 4.3, the choice between the version space and the VC approach depends on the ability of the tester to characterise the complexity hypothesis space for the SUT.

4.3 Bounding Test Sets for SUTs That Are Finite State Machines

This subsection demonstrates the two above techniques, showing how they can estimate the number of tests that are required to test a black-box SUT. For the purposes of illustration, it is assumed that H is the range of deterministic finite state machines over some known alphabet. It is important to note that the use of state machines is merely for the purpose of illustration – Haussler's approach can be applied to a broad range of other representations.

As mentioned previously, the choice between the Version Space approach and the VC dimension approach depends upon the ability to characterise the complexity of H in an appropriate way. In practice the size of $VS_{H,D}$ is infinite for any DFA inference technique, because there are an infinite number of possible DFAs that are consistent with a given test set. The VC dimension for DFAs is also infinite; for any subset of words in X it is possible to produce an exact hypothesis to shatter them, and the largest subset of X is infinite [10].

As a consequence, to ascertain a limit on the test set, it becomes necessary to make some assumptions about the DFA, or the context in which it will be tested. The remainder of this section shows how such assumptions can be used to make the two techniques possible. Specifically, the Version Space approach can be used by imposing a bound on the length of the test cases (implying a bound on the depth of the DFA). Alternatively the VC dimension approach can be used by imposing an upper limit on the number of states in the DFA.

Using the Version Space approach by bounding test case length. For DFA inference, the relationship between the set of samples D and the version space $VS_{H,D}$ was described by Dupont [7]. He showed how the version space can be interpreted as a lattice where the most specific element is a prefix tree automaton (PTA) (a tree-shaped minimal DFA that exactly represents the sample D [7,28,29]), and the most general element is the universal DFA that accepts every element in the DFA alphabet Σ. The size of this version space, which is what we are interested here, is infinite if the depth of the PTA is unrestricted.

However, if the length of the test cases is limited to a chosen length n and the size of the alphabet is denoted σ, the maximum size of a PTA can be computed as: $\sum_{i=0}^{n} \sigma^i$.

In Dupont's lattice version space, any hypothesis $h \in H$ corresponds to a particular partition of the set of states in the PTA (corresponding to the merging of states into their respective equivalence classes). Thus, the size of $VS_{H,D}$ is bounded by the number of possible set partitions of a set the size max – the Bell number of max.

The extremely rapid growth of this number limits the use of the version space approach in this finite state machine setting, and the example shown here is restricted to a very simple SUT. We consider a setting where the length of a test set is restricted to 7, and the size of the alphabet is 3. In this case, the maximum size of the PTA is 3,280 states[1]. The upper bound on the number of DFAs that can be generated from such a PTA as computed by the Bell number of 3280 is approximately $1.5 * 10^{7722}$.

Now the task for the tester is to decide a realistic error margin for the assessment of the test adequacy. To simply state that the test set should always be sufficiently comprehensive to produce an exact model is unrealistic. In our example the tester might consider it sufficient if the model inferred from the test set has an error ≤ 0.1, and that this should happen with a probability of 90%. In other words, $\epsilon = 0.1$ and $\delta = 0.1$ (calculated by $1 - 0.9$). This now allows us to apply Haussler's version-space estimation (see equation 1):

$$\begin{aligned} m &\geq \frac{(ln|VS_{H,D}| + ln(1/\delta)}{\epsilon} \\ \approx m &\geq \frac{(ln(1.5 * 10^{7722}) + ln(1/0.1)}{0.1} \\ \approx m &\geq \frac{17,778.67977 + 2.303}{0.1} \\ \approx m &\geq 177,809.8277 \end{aligned} \tag{3}$$

Taking these values at face value, the task of constructing an adequate test set of this size for such a relatively simple scenario is unrealistic. It is however important to bear in mind the proportions of the problem space. From a possible $1.5 * 10^{7722}$ hypotheses, it is possible to assert that a consistent learner will produce an accurate hypothesis from a 177,810 tests – i.e. to statistically justify this test set will be adequate.

[1] A small Erlang module with all of the routines used to compute the results in this paper is available
http://www.cs.le.ac.uk/people/nwalkinshaw/Files/ictss_code.zip

Of course, considering the relative simplicity of the system in question, this is very large number of tests (despite the vast size of the hypothesis space). There are two things that one has to bear in mind when interpreting this number. Firstly, it is a conservative worst-case estimate. It does not take any failed / impossible tests into account (which would eliminate vast numbers of false hypotheses from an inference standpoint), and does not place any expectations on the learner to do anything with the input data other than be consistent (i.e. not to produce an hypothesis that contradicts the input data). In practice, negative sequences merge out a vast number of invalid merges, and inference techniques often use heuristics [11] to efficiently home-in on the correct merges. Ultimately a justifiable upper bound, even if it is too large, is better than no bound at all, because it provides at least a rough guide for the complexity of the SUT, and the associated testing effort.

Using the VC dimension approach by bounding the number of states in the SUT. In certain cases, it might not be possible to bound $VS_{H,D}$. In the previous setting, any larger alphabets or test lengths would become too large to compute in a practical way, and it might simply be impossible to guess an upper bound on the maximum length of a test case anyway. The VC dimension alternative is useful because it does not rely on a finite version space, but instead provides an internal measure of the complexity of a potentially infinite hypothesis space.

Unfortunately, depending on the representation, it is not always possible to calculate a finite VC dimension. For arbitrary DFAs the VC dimension is infinite and can only be made finite by making assumptions about its maximum number of states – if this is n states, the VC-dimension is bounded by $n \log_2 n$ [10]. Thus, in this case, the choice between version spaces and VC-dimension approaches is determined by the nature of any additional knowledge of the DFA.

Estimating a suitable number of states n relies on the intuition of the tester, from their prior knowledge of the SUT. For this example, let us guess that the SUT contains at most 300 states. The VC dimension is thus bounded by $300 * \log_2(300) = 2,468.65$.

This enables us to substitute for equation 2. As in the initial case for the version space example, let us assume that $\epsilon = 0.1$ and $\delta = 0.1$:

$$\begin{aligned} m &\geq \frac{4log_2(2/\delta) + 8VC(H)log_2(13/\epsilon)}{\epsilon} \\ \approx m &\geq \frac{4log_2(2/0.1) + 8 * 2,468.65 * log_2(13/0.1)}{0.1} \\ \approx m &\geq 1,387,032 \end{aligned} \tag{4}$$

As in the previous version-space approach, this number is a conservative worst-case estimation. It fails to take any heuristic capabilities of the inference technique into account. As previously, depending on the circumstances it might be possible to take this added efficiency into account by increasing the value of ϵ. If, using the same rationale, ϵ is increased to 0.4, the result is a much reduced bound of $m \geq 248,012$.

It is important to bear in mind that it does not make sense to compare the two approaches as presented here with respect to their estimated test sizes, because this would be comparing estimations for (potentially) completely different systems. Nonetheless, with respect to DFAs, if the number of states can be bounded it is better to use the VC-dimension approach, because it is easier to compute an estimate for more complex systems and tends to compute a much lower bound than the version-space approach (this latter fact applies to all representations, not just DFAs [9,14]).

5 Using the PAC Setting to Empirically Assess Test Sets

The ability to predict the sizes of test sets is only one side of the benefit of using the PAC framework for testing, and has been already explored to some extent in previous literature [31,21,20]. From an empirical aspect, the framework is equally valuable, because it presents us with a basis for making statistically justified measurements of test set adequacy for black-box SUTs, by assessing the performance of the inferred models. This section presents a practical example of this. It not only shows the value of being able to assess test sets, but also highlights an important practical consideration that can lead to problems of accuracy when computing the lower bounds for test size computed by the techniques presented in the previous section.

Current techniques that combine testing with model inference make a binary decision; a test set is adequate if it leads to an exactly accurate model (as assessed within the limits of some model-based testing technique), and inadequate otherwise. This is problematic because there is no basis for homing in on an adequate test set. This section illustrates how the PAC testing framework (as shown in Figure 3.2) can be applied in a practical context to provide feedback about test set adequacy.

5.1 The SUT, and the Choice of Test Generation and Model Inference Techniques

The SUT in question simulates the behaviour of an SSH client, in terms of the FSM specification described by Poll *et al.* [18]. It accepts sequences of instructions as specified, but will throw an "unexpected input" exception if given a sequence of inputs that is not part of the specification. The system is written in Erlang (implemented using the `gen_fsm` behavioural pattern – available with the source code provided with this paper).

Let us assume that we have a small sample of 10 test scenarios that execute some of the expected behaviour of the system. Because these only exercise a tiny (albeit functionally significant) fraction of program behaviour, it is necessary to substantially bulk up the test set if we want it to be adequate. Given that we are presuming no further domain knowledge about the system, the rest of the tests will have to be generated randomly. For this we use a random generator that produces a set of unique random sequences from the given alphabet up to a certain length (for this example we choose 13 to be the maximum test case length, and choose the length of each test case randomly).

> The problem with any resulting test set, no matter how large, is that we do not know how *adequate* it is.

This is where the PAC framework can offer a solution. By generating two non-intersecting test sets, using one to infer a model, and the other to assess its accuracy, it is possible to obtain an insight into how adequate the first test set is.

To infer a model from the tests we choose Price's EDSM blue-fringe state merging algorithm [11] – until recently the most accurate algorithm for inferring state machines from arbitrary examples. We use the openly available Ruby implementation by Bernard Lambeau that was developed as a baseline for the StaMInA inference competition [27].

5.2 Application of the PAC Framework

We start by generating test sets A and B (see Figure 3.2). The PAC framework assumes that these are drawn randomly from the same distribution, but must not overlap. To ensure that this is the case a large set of unique random test cases is generated, and the contents of A and B are selected at random from this superset. Set A will be used to train the model, and will be the test set that we assess, and test set B will be the set with which we assess the accuracy of the model (and so the adequacy of A).

Due to time constraints, we terminate the generation algorithm after an hour. In that time sets A and B have been populated with 42,410 tests each. Now the tests and their respective outcomes from set A are used to infer a model using the StaMInA tool. The model is then used to predict the outcomes for test set B. The error rate can then tell us how adequate test set A. If we use the conventional definition of $error_{\mathcal{D}}(h)$ to compute the error, we end up with an adequacy assessment of 99.99%.

Upon closer inspection, splitting the test cases up into true or false positives and negatives shows that this figure has to be interpreted with care. Out of the 42,410 test cases, 42,397 tests are true negatives, five tests are true positives, five tests are false positives and three are false negatives. Ultimately, the fact that there is such a high overlap between sets A and B says more about distribution from wich they were sampled than it does about the SUT. PAC-learning assumes that the distributions A and B reflect the routine behaviour of the system in question, in which case this measure of overlap is appropriate.

In a testing context, this measure is not particularly helpful. A randomly generated test case will generate arbitrary distributions of test cases that do not evenly represent the input domain of the SUT, and may lead to heavily skewed error rates. To account for this, we use the Balanced Classification Rate (BCR) [27][2], which balances the ability of the inferred model to reject false negatives

[2] This measure is commonly used in machine learning, and should not be attributed to the author, but is described in this paper with respect to DFA inference.

against its ability to reject false positives: $BCR_{\mathcal{D}}(h) = \frac{1}{2}(TP/(TP + FN)) + (TN/(TN+FP))$. If we apply this to calculate the adequacy of test set A above, we obtain a more balanced test adequacy assessment of 0.8124.

5.3 Discussion

This section has demonstrated how to assess the adequacy of a test set for a black-box system. However, in drawing the distinction between the different measures for calculating classification error ($error_{\mathcal{D}}(h)$ and $bcr_{\mathcal{D}}(h)$), it highlights an important caveat for interpreting the test-size estimations produced by the techniques in section 4. These predictions only apply when the test set can be reliably assessed using the $error_{\mathcal{D}}(h)$ measure, i.e. when the distribution of tests is roughly balanced between valid and invalid cases.

If this is not the case, the estimated lower bound on the number of required test cases will probably be a significant underestimation. Given that the SSH implementation has 19 distinct states [18], this can be illustrated with the VC-dimension approach. For 19 states we obtain a VC-dimension of 80.71. We might guess a conservative ϵ value of 0.1, and choose a δ value of 0.1. Substituting into equation 2, this gives us an estimated lower bound of 45,515 test cases.

This happens to be relatively close to the number of tests we generated in section 5.2. Had we used the conventional measure for $error_{\mathcal{D}}(h)$, this would certainly be accurate. However, given that out of 42,410 random test-cases only 8 of these produce valid outputs from the SUT, it is clear that a much larger number of random test cases would be required to fully exercise the SUT in terms of its valid behaviour as well (and so reach a high level of adequacy with respect to the BCR). Future work (see section 7) will elaborate the techniques from section 4 to develop more accurate lower bounds that take into account the the balance between valid and invalid test cases.

6 Related Work

As discussed in the Background section, there has been much work on relating the fields of machine learning and software testing, and several relevant references are included in this paper. Due to limited space, this section shall focus on the more specific topic of the use of probabilistic techniques to reason about test sets for black-box systems without specifications.

The idea of testing "to a confidence level" by joining the fields of Inductive Inference and Testing was first raised by Cherniavsky and Smith in 1987 [6] (although their work was primarily concerned with learning exact models). The subject was subsequently explored by Zhu *et al.* [32,31]. They used the PAC framework as a theoretical basis to reinterpret and justify a set of fundamental test adequacy axioms they had proposed in earlier work. They also suggest using an alternative to the VC-dimension and version-space estimation approaches to predict the necessary size of a test set (Haussler's Pseudo-dimension) though given that the work is theoretical in nature, there is no suggestion of how this might be used in a practical context (e.g. how to compute the pseudo-dimension for a given type of black box SUT).

The combination of PAC learning and testing was the subject of a substantial amount of work by Romanik *et al.*. By adopting the PAC framework, they proposed the notion of *approximate testing* [20]. They show how familiar machine learning concepts such as the VC-dimension can be used to reason about the general (approximate) testability of particular classes of program, with a particular interest in reasoning about certain classes that are *un-testable.* In subsequent work, Romanik [21] considers the relationship between the internal complexity of a program (i.e. its source code branching) and its testability, and proposes an extension of the VC-dimension (the VCP-dimension) to measure this.

Although Zhu and Romanik made pioneering theoretical contributions to the research on combining the two fields, it was perhaps the relative primitiveness of machine learning techniques at the time that prevented the practical application. It is only recently that inference techniques have developed the capabilities to infer (approximately) accurate models of software systems. To the best knowledge of the author, this paper the first work that attempts to experiment with the combination of PAC-learning and testing in a practical sense for the sake of assessing test sets.

Many of the recent testing techniques to involve machine learning (c.f. work by Raffelt *et al.* and Shahbaz *et al.* [19,23] are based on Angluin's L^* algorithm [1]. In her paper, she discusses how her algorithm could be adapted to suit a PAC setting. To the best of the author's knowledge, this has not yet been implemented in a testing context, but suggests that it would in principle be straightforward to adapt these existing testing techniques to apply the PAC-based principles that have been discussed in this paper.

7 Conclusions and Future Work

The challenge of producing a comprehensive test set for a black-box system without recourse to a specification is seemingly impossible. There is no obvious basis for determining whether a test set is adequate, and for identifying a candidate set of test cases from a potentially infinite set of potentials.

Against this backdrop, machine learning is a particularly interesting discipline, because it provides a wealth of techniques to reason in a systematic way about hidden systems by way of experimentation. Valiant's PAC framework provides a useful formal basis for this, and has formed the basis for a limited amount of theoretical work on software testing [31,21,20]. Specifically, the PAC framework can be used to reason about the accuracy of the inferred model which, in turn, provides feedback about the adequacy of the test set that was used to infer it. Furthermore, the use of the PAC framework enables the estimation of how many test sets might be required to produce an adequate test set.

In the light of the recent emergence of numerous testing techniques that are founded on specific machine learning techniques [2,5,8,12,17,19,22,23,26,28,29], this paper has sought to investigate the practical application use of the PAC framework. The paper shows how the PAC framework can be applied in practice. it shows how test set sizes can be predicted, and how the framework can be used to obtain a statistically valid assessment of test adequacy in practice.

The practical example in this paper has highlighted one problem of applying the PAC framework in a testing context. It is assumed that there is a rough balance between valid and invalid test cases, and the conventional measure of error can be misleading when this is not the case (which is typical in a testing context). Future work will attempt to adapt Haussler's predictions [9], to produce more accurate predictions for typical random testing situations, where the test set is not balanced.

Acknowledgements. Much of the background material that relates inductive inference to software testing was influenced by discussions with Gordon Fraser at Saarland University in Saarbrücken.

References

1. Angluin, D.: learning regular sets from queries and counterexamples. Information and Computation 75, 87–106 (1987)
2. Berg, T., Grinchtein, O., Jonsson, B., Leucker, M., Raffelt, H., Steffen, B.: On the correspondence between conformance testing and regular inference. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 175–189. Springer, Heidelberg (2005)
3. Bergadano, F., Gunetti, D.: Testing by means of inductive program learning. ACM Transactions on Software Engineering and Methodology 5(2), 119–145 (1996)
4. Blumer, A., Ehrenfeucht, A., Haussler, D., Warmuth, M.: Learnability and the vapnik-chervonenkis dimension. Journal of the ACM 36, 929–965 (1989)
5. Briand, L., Labiche, Y., Bawar, Z., Spido, N.: Using machine learning to refine category-partition test specifications and test suites. Information and Software Technology 51, 1551–1564 (2009)
6. Cherniavsky, J., Smith, C.: A recursion theoretic approach to program testing. IEEE Transactions on Software Engineering 13 (1987)
7. Dupont, P., Miclet, L., Vidal, E.: What is the search space of the regular inference? (1994)
8. Ghani, K., Clark, J.: Strengthening inferred specifications using search based testing. In: International Conference on Software Testing Workshops (ICSTW). IEEE, Los Alamitos (2008)
9. Haussler, D.: Quantifying inductive bias: Ai learning algorithms and valiant's learning framework. Artificial Intelligence 36, 177–221 (1988)
10. de la Higuera, C.: Grammatical Inference: Learning Automata and Grammars. Cambridge University Press, Cambridge (2010)
11. Lang, K., Pearlmutter, B., Price, R.: Results of the Abbadingo One DFA Learning Competition and a New Evidence-Driven State Merging Algorithm. In: Honavar, V.G., Slutzki, G. (eds.) ICGI 1998. LNCS (LNAI), vol. 1433, pp. 1–12. Springer, Heidelberg (1998)
12. Last, M.: Data mining for software testing. In: The Data Mining and Knowledge Discovery Handbook, pp. 1239–1248. Springer, Heidelberg (2005)
13. Lee, D., Yannakakis, M.: Principles and Methods of Testing Finite State Machines - A Survey. Proceedings of the IEEE 84, 1090–1126 (1996)

14. Mitchell, T.: Machine Learning. McGraw-Hill, New York (1997)
15. Mitchell, T.: Generalization as search. Artificial Intelligence 18(2), 203–226 (1982)
16. Nagappan, N., Murphy, B., Basili, V.: The influence of organizational structure on software quality: an empirical case study. In: International Conference on Software Engineering (ICSE), pp. 521–530. ACM, New York (2008)
17. Perkins, J., Ernst, M.: Efficient incremental algorithms for dynamic detection of likely invariants. SIGSOFT Software Engineering Notes 29, 23–32 (2004)
18. Poll, E., Schubert, A.: Verifying an implementation of ssh. In: Workshop on Issues of Theory of Security (WITS), pp. 164–177 (2007)
19. Raffelt, H., Steffen, B.: Learnlib: A library for automata learning and experimentation. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 377–380. Springer, Heidelberg (2006)
20. Romanik, K.: Approximate testing and its relationship to learning. Theoretical Computer Science 188(1-2), 175–194 (1997)
21. Romanik, K., Vitter, J.: Using Vapnik-Chervonenkis dimension to analyze the testing complexity of program segments. Information and Computation 128(2), 87–108 (1996)
22. Shahamiri, S., Kadira, W., Ibrahima, S., Hashim, S.: An automated framework for software test oracle. Information and Software Technology 53 (2011)
23. Shahbaz, M., Groz, R.: Inferring mealy machines. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 207–222. Springer, Heidelberg (2009)
24. Valiant, L.: A theory of the learnable. Communications of the ACM 27(11), 1134–1142 (1984)
25. Vapnik, V., Chervonenkis, A.: On the uniform convergence of relative frequencies of events to their probabilities. Theory of Probability and its Applications 16(2), 264–280 (1971)
26. Walkinshaw, N.: The practical assessment of test sets with inductive inference techniques. In: Bottaci, L., Fraser, G. (eds.) TAIC PART 2010. LNCS, vol. 6303, pp. 165–172. Springer, Heidelberg (2010)
27. Walkinshaw, N., Bogdanov, K., Damas, C., Lambeau, B., Dupont, P.: A framework for the competitive evaluation of model inference techniques. In: Proceedings of the First International Workshop on Model Inference In Testing (MIIT), pp. 1–9. ACM, New York (2010)
28. Walkinshaw, N., Bogdanov, K., Derrick, J., Paris, J.: Increasing functional coverage by inductive testing: A case study. In: Petrenko, A., Simão, A., Maldonado, J.C. (eds.) ICTSS 2010. LNCS, vol. 6435, pp. 126–141. Springer, Heidelberg (2010)
29. Walkinshaw, N., Derrick, J., Guo, Q.: Iterative refinement of reverse-engineered models by model-based testing. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 305–320. Springer, Heidelberg (2009)
30. Weyuker, E.: Assessing test data adequacy through program inference. ACM Transactions on Programming Languages and Systems 5(4), 641–655 (1983)
31. Zhu, H.: A formal interpretation of software testing as inductive inference. Software Testing, Verification and Reliability 6(1), 3–31 (1996)
32. Zhu, H., Hall, P., May, J.: Inductive inference and software testing. Software Testing, Verification, and Reliability 2(2), 69–81 (1992)

Author Index