

Lecture Notes in Computer Science

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

2931

Springer

Berlin

Heidelberg

New York

Hong Kong

London

Milan

Paris

Tokyo

Alexandre Petrenko Andreas Ulrich (Eds.)

Formal Approaches to Software Testing

Third International Workshop on
Formal Approaches to Testing of Software, FATES 2003
Montreal, Quebec, Canada, October 6th, 2003
Revised Papers



Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Alexandre Petrenko
Centre de recherche informatique de Montréal (CRIM)
550 Sherbrooke West, Suite 100, Montreal, H3A 1B9, Canada
E-mail: Petrenko@crim.ca

Andreas Ulrich
Siemens AG, Corporate Technology CT SE 1
Otto-Hahn-Ring 6, 81730 Munich, Germany
E-mail: andreas.ulrich@siemens.com

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress.

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie;
detailed bibliographic data is available in the Internet at <<http://dnb.ddb.de>>.

CR Subject Classification (1998): D.2.5, D.2.4, D.2, D.3.1, F.3.1

ISSN 0302-9743

ISBN 3-540-20894-1 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

Springer-Verlag is a part of Springer Science+Business Media
springeronline.com

© Springer-Verlag Berlin Heidelberg 2004
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper SPIN: 10984468 06/3142 5 4 3 2 1 0

Preface

Formal methods provide system designers with the possibility to analyze system models and reason about them with mathematical precision and rigor. The use of formal methods is not restricted to the early development phases of a system, though. The different testing phases can also benefit from them to ease the production and application of effective and efficient tests. Many still regard formal methods and testing as an odd combination. Formal methods traditionally aim at verifying and proving correctness (a typical academic activity), while testing shows only the presence of errors (this is what practitioners do). Nonetheless, there is an increasing interest in the use of formal methods in software testing. It is expected that formal approaches are about to make a major impact on emerging testing technologies and practices. Testing proves to be a good starting point for introducing formal methods in the software development process.

This volume contains the papers presented at the 3rd Workshop on Formal Approaches to Testing of Software, FATES 2003, that was in affiliation with the IEEE/ACM Conference on Automated Software Engineering (ASE 2003). This year, FATES received 43 submissions. Each submission was reviewed by at least three independent reviewers from the program committee with the help of additional reviewers. Based on their evaluations, 18 papers submitted by authors from 13 different countries were selected for presentation at the workshop. The papers present different approaches to using formal methods in software testing. One of the main themes is the generation of an efficient and effective set of test cases from a formal description. Different models and formalisms are used, such as finite state machines, input/output transition systems, timed automata, UML, and Abstract State Machines. An increasing number of test methodologies (re)uses techniques from model checking. The prospects for using formal methods to improve software quality and reduce the cost of software testing are encouraging. But more efforts are needed, both in developing new theories and making existing methods applicable to the current practice of software development projects. Without doubt, coming FATES workshops will continue to contribute to the growing and evolving research activities in this field.

We wish to express our gratitude to the authors for their valuable contributions. We thank the program committee and the additional reviewers for their support in the paper selection process. Last but not least, we thank May Haydar who helped in organizing the proceedings and all persons from the Centre de Recherche Informatique de Montréal and the organizing committee of ASE 2003 who were involved in arranging local matters.

Program Committee

Marie-Claude Gaudel	Université Paris-Sud, France
Jens Grabowski	Georg-August-Universität Göttingen, Germany
Dick Hamlet	Portland State University, USA
Robert M. Hierons	Brunel University, UK
Thierry Jéron	IRISA, France
David Lee	Bell Labs Research, China
Brian Nielsen	Aalborg University, Denmark
Jeff Offutt	George Mason University, USA
Alexandre Petrenko	CRIM, Canada
Jan Tretmans	Universiteit Nijmegen, The Netherlands
Andreas Ulrich	Siemens AG, Germany
Antti Valmari	Tampereen Teknillinen Yliopisto, Finland
Carsten Weise	Ericsson, Germany
Martin Woodward	Liverpool University, UK

Additional Reviewers

Serge Boroday	Keqin Li
Rene de Vries	Andrej Pietschker
Lars Frantzen	Vlad Rusu
Hesham Hallal	Valery Tschaen
May Haydar	Machiel van der Bijl
Ruibing Hao	Dong Wang
Jia Le Huo	Gavin Yang
Claude Jard	Xia Yin
Ahmed Khoumsi	

Table of Contents

Program Testing and Analysis

Black-Box Testing of Grey-Box Behavior	1
<i>Benjamin Tyler and Neelam Soundarajan</i>	
On Checking Whether a Predicate Definitely Holds	15
<i>Alper Sen and Vijay K. Garg</i>	
Using a Software Testing Technique to Improve Theorem Proving	30
<i>Reiner Hähnle and Angela Wallenburg</i>	
Auto-generating Test Sequences Using Model Checkers: A Case Study	42
<i>Mats P.E. Heimdahl, Sanjai Rayadurgam, Willem Visser, George Devaraj, and Jimin Gao</i>	
Mutually Enhancing Test Generation and Specification Inference	60
<i>Tao Xie and David Notkin</i>	
JMLAutoTest: A Novel Automated Testing Framework Based on JML and JUnit	70
<i>Guoqing Xu and Zongyuang Yang</i>	

Test Theory and Test Derivation Algorithms

Compositional Testing with IOCO	86
<i>Machiel van der Bijl, Arend Rensink, and Jan Tretmans</i>	
Defining Observation Objectives for Reactive and Distributed Systems	101
<i>Timo Aaltonen</i>	
Time-Optimal Real-Time Test Case Generation Using UPPAAL	114
<i>Anders Hessel, Kim G. Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou</i>	
Test Cases Generation for Nondeterministic Real-Time Systems	131
<i>Ahmed Khoumsi, Thierry Jéron, and Hervé Marchand</i>	
Property Oriented Test Case Generation	147
<i>Jean-Claude Fernandez, Laurent Mounier, and Cyril Pachon</i>	
Computing Unique Input/Output Sequences Using Genetic Algorithms . . .	164
<i>Qiang Guo, Robert M. Hierons, Mark Harman, and Karnig Derderian</i>	
Automatic Generation of Test Purposes for Testing Distributed Systems . .	178
<i>Olaf Henniger, Miao Lu, and Hasan Ural</i>	

Test Methods and Test Tools

Interaction Testing in an Embedded System Using Hardware Fault Injection and Program Mutation	192
<i>Ahyoung Sung and Byoungju Choi</i>	
Automatic Conformance Testing of Internet Applications	205
<i>Harm M.A. van Beek and Sjouke Mauw</i>	
A Use Case Driven Testing Process: Towards a Formal Approach Based on UML Collaboration Diagrams	223
<i>Mourad Badri, Linda Badri, and Marius Naha</i>	
VISWAS and on Diagnosability with IEEE Std P1522 and UML2.0 Testing Profile	236
<i>Sita Ramakrishnan</i>	
Towards a Tool Environment for Model-Based Testing with AsmL	252
<i>Mike Barnett, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes</i>	
Author Index	267

Black-Box Testing of Grey-Box Behavior

Benjamin Tyler and Neelam Soundarajan

Computer and Information Science
Ohio State University, Columbus, OH 43210, USA
{tyler,neelam}@cis.ohio-state.edu

Abstract. *Object-oriented frameworks* are designed to provide functionality common to a variety of applications. Developers use these frameworks in building their own specialized applications, often without having the source code of the original framework. Unfortunately, the interactions between the framework components and the new application code can lead to behaviors that could not be predicted even if valid *black-box* specifications were provided for the framework components. What is needed are *grey-box* specifications that include information about sequences of method calls made by the original framework code. Our focus is on how to test frameworks against such specifications, which requires the ability to monitor such method calls made by the framework during testing. The problem is that without the source code of the framework, we cannot resort to code instrumentation to track these calls. We develop an approach that allows us to do this, and demonstrate it on a simple case study.

1 Introduction

An important feature of object-oriented (OO) languages is the possibility of enriching or extending the functionality of an OO system [18] by providing, in derived classes, suitable definitions or re-definitions for some of the methods of some of the classes of the given system. *Application frameworks* [9,13,20] provide compelling examples of such enrichment. The framework includes a number of *hooks*, methods that are not (necessarily) defined in the framework but are invoked in specific, and often fairly involved, patterns by the *polymorphic* or *template* methods [11] defined in the framework. An application developer can build a complete customized application by simply providing appropriate (re-)definitions for the hook methods, suited to the needs of the particular application. The calls to the hook methods from the template methods are *dispatched* to the methods defined by the application developer, so that the template methods also exhibit behavior tailored to the particular application. Since the patterns of hook method calls implemented in the template methods are often among the most intricate part of the overall application, a well designed framework can be of great help in building applications, and maximizes the amount of reuse among the applications built on it. Our goal is to investigate approaches to perform specification-based testing of such frameworks.

Testing such systems should clearly include testing these patterns of hook method calls. That is, we are interested in testing what is called the *grey-box*

behavior [2,5,10,22] of OO systems, not just their *black-box* behavior. If we had access to the *source code* of the template methods, we could do this by instrumenting that code by inserting suitable instructions at appropriate points to record information about the hook method calls; for example, just prior to each such call, we could record the identity of the method being called, the values of the arguments, etc. But framework vendors, because of proprietary considerations, often will not provide the source code of their systems. Hence the challenge we face is to find a way to test the grey-box behavior of template methods without being able to make any changes to its code such as adding “monitoring code”, indeed without even having the file containing source code of the system.

In this paper, we develop an approach that allows us to do this. The key idea underlying our approach is to exploit *polymorphism* to *intercept* hook method calls made by the template method being tested. When the hook method call is intercepted, the testing system will record the necessary information about the call, and then allow “normal” execution to resume. In a sense, the testing system that we build for testing a given framework *is itself an application built on the framework being tested*. This “application” can be generated automatically given information about the structure of the various classes that are part of the framework including the names and parameter types of the various methods and their specifications, and the *compiled* code of the framework. We have implemented a prototype *test system generator* that accomplishes this task. We present some details about our prototype later in the paper.

1.1 Black-Box vs. Grey-Box Behavior

How do we specify grey-box behavior? Standard specifications [14,18] in terms of *pre-* and *post-conditions* for each method of each class in the system only specify the black-box behavior of the method in question. Consider a template (or polymorphic, we will use the terms interchangeably) method $t()$. There is no information in the standard specification of $t()$ about the hook method calls that $t()$ makes during execution. We can add such information by introducing a *trace* variable [5,22], call it τ , as an *auxiliary* variable [19] on which we record information about the hook method calls $t()$ makes. When the method starts execution, τ will be the empty sequence since at the start, $t()$ has not made any such calls. As $t()$ executes, information about each hook method call it makes will be recorded on τ . We can then specify the grey-box behavior by including, in the post-condition of $t()$, not just information on the state of the object in question when $t()$ terminates, but also about the value of τ , i.e., about the hook method calls $t()$ made during its execution; we will see examples of this later in the paper. Given such a grey-box specification, the key question we address is, how do we test $t()$, without accessing or modifying its code, to see if its actual grey-box behavior satisfies the specification?

1.2 Comparison to Related Work

A number of authors have addressed problems related to testing of polymorphic interactions [1,21,3,17] in OO systems. In all of this work, the approach is to

try to test the behavior of a polymorphic method `t()` by using objects of all or many different derived classes to check whether `t()` behaves appropriately in each case, given the different hook method definitions to which the calls in `t()` will be dispatched, depending on the particular derived class that the given object is an instance of. Such an approach is not suitable for testing frameworks. We are interested in testing the framework independently of any application that may be built on it, i.e., independently of particular derived classes and particular definitions of the hook methods. The only suitable way to do this is to test it directly to see that the actual sequences of hook method calls it makes during the tests are consistent with its grey-box specification. The other key difference is our focus on testing polymorphic methods without having access to their source code.

Another important question, of course, has to do with *coverage*. Typical coverage criteria that have been proposed [1,21,6] for testing polymorphic code have been concerned with measuring the extent to which, for example, every hook method call that appears in the polymorphic method is dispatched, in some test run, to each definition of the hook method (in the various derived classes). Clearly a criterion of this kind would be inappropriate for our purposes since our goal is to test the polymorphic methods of the framework independently of any derived classes. What we should aim for instead is to select test cases in such a way as to ensure that as many as possible of the sequences of hook method calls allowed by the grey-box specifications actually appear in the test runs. One problem here, as in any specification-based testing approach, is that the specification only specifies what behavior is *allowed*; there is no requirement that the system actually exhibit each behavior allowed by the specification. Hence, measuring our coverage by checking the extent to which the different sequences of hook method calls allowed by the specification show up in the test runs may be too conservative if the framework is not actually capable of exhibiting some of those sequences. Another approach, often used with specification-based testing, is based on partitioning of the input space, i.e., the set of values allowed by the pre-condition of the method. But partition-based testing suffers from some important problems [8,12] that raise concerns about its usefulness. We will return to this question briefly in the final section but we should note that our focus in this paper is developing an approach that, without needing us to access or modifying the source code of a template method, allows us to check whether the method meets its grey-box specification during a test run, rather than coverage criteria.

1.3 Contributions

The main contributions of the paper may be summarized as follows:

- It identifies the importance of testing grey-box behavior of OO systems.
- It develops an approach to testing a system to see if it meets its grey-box specification without accessing or modifying the code of the system under test.
- It illustrates the approach by applying it to a simple case study.

In Sect. 2 we consider how to specify grey-box behavior. In Sect. 3, we develop our approach to testing against such specifications without accessing the code. We use a simple case study as a running example in Sects. 2 and 3. In Sect. 4 we present some details of our prototype system. In Sect. 5, we summarize our approach and consider future work.

2 Grey-Box Specifications

2.1 Limitations of Black-Box Specifications

Consider the `Eater` class, a simple class whose instances represent entities that lead sedentary lives consisting of eating donuts and burgers, depicted in Fig. 1. The methods `Eat_Donuts()` and `Eat_Burgers()` simply update the single member variable `cals_Eaten` which keeps track of how many calories have been consumed; the parameter `n` indicates how many donuts or burgers is to be consumed. `Pig_Out()` is a template method and invokes the hook methods `Eat_Donuts()` and `Eat_Burgers()`.

```
class Eater {
    protected int cals_Eaten = 0;
    public void Eat_Donuts(int n) {
        cals_Eaten = cals_Eaten + 200 * n;
    }
    public void Eat_Burgers(int n) {
        cals_Eaten = cals_Eaten + 400 * n;
    }
    public final void Pig_Out() {
        Eat_Donuts(2); Eat_Burgers(2);
    }
}
```

Fig. 1. Base class `Eater`.

Let us now consider the specification of `Eater`'s methods (Fig. 2). These can be specified as usual in terms of pre- and post-conditions describing the effect of each method on the member variables of the class. Here, we use the prime ($'$) notation in the post-conditions to refer to the value of the variable in question at the time the method was invoked. Thus the specifications of `Eat_Donuts()` and `Eat_Burgers()` state that each of them increments the value of `cals_Eaten` appropriately. Given the behaviors of these methods, it is easy to see that the template method `Pig_Out()` will meet its specification that it increments `cals_Eaten` by 1200.

Now suppose that the implementers of `Eater` provide only the compiled binary file and the black-box specification shown in Fig. 2, but not the source code in Fig. 1, to developers who wish to incorporate `Eater` in their own systems. What can such developers safely say about their own new classes that are extensions of `Eater`? Let us examine this question using the `Eater_Jogger` class, depicted in Fig. 3. `Eater_Jogger`, which is a derived class of `Eater`, keeps track not only of

$$\text{pre.Eat_Donuts}(n) \equiv n > 0 \quad (2.1)$$

$$\text{post.Eat_Donuts}(n) \equiv \text{cals.Eaten} = \text{cals.Eaten}' + 200 * n$$

$$\text{pre.Eat_Burgers}(n) \equiv n > 0 \quad (2.2)$$

$$\text{post.Eat_Burgers}(n) \equiv \text{cals.Eaten} = \text{cals.Eaten}' + 400 * n$$

$$\text{pre.Pig_Out}() \equiv \text{true} \quad (2.3)$$

$$\text{post.Pig_Out}() \equiv \text{cals.Eaten} = \text{cals.Eaten}' + 1200$$

Fig. 2. Eater's black-box specification.

```
class Eater_Jogger extends Eater {
    protected int cals_Burned = 0;
    public void Jog() {
        cals_Burned = cals_Burned + 500; }
    public void Eat_Donuts(int n) {
        cals_Eaten = cals_Eaten + 200 * n;
        cals_Burned = cals_Burned + 5 * n;}
    public void Eat_Burgers(int n) {
        cals_Eaten = cals_Eaten + 400 * n;
        cals_Burned = cals_Burned + 15 * n;}
}
```

Fig. 3. The derived class Eater_Jogger.

cals.Eaten but also the new data member cals.Burned. The new method Jog() simply increments cals.Burned. More important, Eat_Donuts() and Eat_Burgers() have been redefined to update cals.Burned.

What can we say about the behavior of Pig_Out() in this derived class? More precisely the question is, if ej is an object of type Eater_Jogger, what effect will the call ej.Pig_Out() have on ej.cals.Eaten and ej.cals.Burned? The calls in Pig_Out() to the hook methods will be dispatched to the methods redefined in Eater_Jogger. If we had access to the body of Pig_Out() (defined in the base class), we can see that it invokes Eat_Donuts(2) and then Eat_Burgers(2), and hence conclude, given the behaviors of these methods as redefined in Eater_Jogger, that in this class, Pig_Out() would increment cals.Eaten by 1200 and cals.Burned by 40. However, we have assumed that we only have access to Eater's black-box specification shown in Fig. 2, but *not* the source code of Pig_Out().

Behavioral subtyping [15] provides part of the answer to this question. In essence, a derived class D is a behavioral subtype of its base class B if every method redefined in D satisfies its B-specification. If this requirement is met then we can be sure that in the derived class, a template method t() will meet its original specification ((2.3) in the case of Pig_Out()). This is because when reasoning about the behavior of t() in the base class, we would have appealed to the base class specifications of the hook methods when considering the calls in t() to these methods. If these methods, as redefined in D, satisfy those specifications, then clearly that reasoning still applies when the calls that t() makes to these methods are dispatched to the redefined versions in D. Our redefined

`Eat_Donuts()` and `Eat_Burgers()` do clearly satisfy their base class specifications (2.1) and (2.2), hence `Pig_Out()` in the derived class will also meet its base class specification (2.3).

But this is only part of the answer. The redefined hook methods not only satisfy their base class specifications but exhibit *richer* behavior in terms of their effect on the new variable `cals_Burned`, which is easily specified (Fig. 4). Indeed, the whole point of redefining the hook methods was to achieve this richer behavior; after all, if all we cared about was the base class behavior, there would have been no need to redefine them at all. Not only is the hook methods' behavior enriched through their redefinition, but the behavior of the template method in the derived class will also be enriched even though its code was not changed. How then, can we reason about this richer behavior of the template method?

$$\text{pre.Eat_Donuts}(n) \equiv n > 0 \quad (4.1)$$

$$\begin{aligned} \text{post.Eat_Donuts}(n) &\equiv \text{cals_Eaten} = \text{cals_Eaten}' + 200 * n \\ &\quad \wedge \text{cals_Burned} = \text{cals_Burned}' + 5 * n \end{aligned}$$

$$\text{pre.Eat_Burgers}(n) \equiv n > 0 \quad (4.2)$$

$$\begin{aligned} \text{post.Eat_Burgers}(n) &\equiv \text{cals_Eaten} = \text{cals_Eaten}' + 400 * n \\ &\quad \wedge \text{cals_Burned} = \text{cals_Burned}' + 15 * n \end{aligned}$$

Fig. 4. Specifications for `Eater_Jogger`'s hook methods.

If we examine the specifications for the redefined hook methods shown in Fig. 4, and (2.3), the black-box specification of `Pig_Out()`, can we arrive at the richer behavior of `Pig_Out()` in `Eater_Jogger`, in particular that it will increment `cals.Burned` by 40? The answer is clearly no, since there is nothing in (2.3) that tells us which, if any, hook methods `Pig_Out()` calls and how many times and with what argument values. Given (2.3), it is possible that it called `Eat_Donuts()` once with 6 as the argument and never called `Eat_Burgers()`; or `Eat_Burgers()` once with 3 as the argument, and `Eat_Donuts()` zero times; it is even possible that `Pig_Out()` didn't call either hook method even once and instead directly incremented `cals.Eaten` by 1200. Even an implementation that called `Eat_Donuts()` ten times with 2 as the argument each time and then decremented `cals.Eaten` by 2800 would work. All of these and more are possible, and depending on which of these `Pig_Out()` actually does, its effect on `cals.Burned` will be different. Note that for all of these cases, the original behavior (2.3) is still satisfied. *That* is ensured by behavioral subtyping. But if we are to arrive at the richer behavior of `Pig_Out()`, we need not just the black-box behavior of the template method in the base class as specified in (2.3), but also its grey-box behavior.

2.2 Reasoning with Grey-Box Specifications

Consider the *grey-box* specification (5.1) in Fig. 5. Here, τ is the *trace* of this template method. τ is the empty sequence, ε , when `Pig_Out()` begins execution. Each time `Pig_Out()` invokes a hook method, we add an element to record this

hook method invocation. This element contains the name of the hook method called, the values of the member variables of the `Eater` class at the time of the call, their values at the time of the return from this call, the values of any additional arguments at the time of the call, their values at the time of the return, and the value of any additional result returned by the call. The grey-box post-condition gives us information about the value of τ when the method finishes, hence about the hook method calls it made during its execution. Thus (5.1) states that $|\tau|$, the length of, i.e. the number of elements in, τ is 2; that the hook method called in the first call, recorded in the first element $\tau[1]$ of the trace, is `Eat_Donuts`; that the argument value passed in this call is 2; the hook method called in the second call is `Eat_Burgers`; and the argument passed in this call is 2.

$$\begin{aligned}
 \text{pre.Pig_Out()} &\equiv \tau = \varepsilon \\
 \text{post.Pig_Out()} &\equiv \text{cals_Eaten} = \text{cals_Eaten}' + 1200 \wedge |\tau| = 2 \\
 &\quad \wedge \tau[1].\text{method} = \text{"Eat_Donuts"} \wedge \tau[1].\text{arg} = 2 \\
 &\quad \wedge \tau[2].\text{method} = \text{"Eat_Burgers"} \wedge \tau[2].\text{arg} = 2
 \end{aligned} \tag{5.1}$$

Fig. 5. Grey-box specification for `Pig_Out` in `Eater`.

It should be noted that (5.1) does not give us additional information about the value that `cals_Eaten` had at the time of either call or return. While this simplifies the specification, it also means that redefinitions of the hook methods that depend on the value of `cals_Eaten` cannot be reasoned about given (5.1). This is a tradeoff that we have to make when writing grey-box specifications; include full information, resulting in a fairly complex specification; or leave out some of the information, foreclosing the possibility of some enrichments (or at least of reasoning about such enrichments, which amounts to the same thing in the absence of access to the source code of the template method).

Given this grey-box specification, what can we conclude about the behavior of `Pig_Out()` in the derived class? Note first that from (4.1) and (4.2), we can deduce that `Eater_Jogger.Eat_Donuts()` and `Eater_Jogger.Eat_Burgers()` satisfy (2.1) and (2.2), i.e., they satisfy the requirement of behavioral subtyping; hence `Pig_Out()` will satisfy (2.3) when invoked on `Eater_Jogger` objects. But we can also conclude given (2.1) and (2.2) and, as specified by (5.1), that `Pig_Out()` will make two hook method calls during its execution, first to `Eat_Donuts()` with argument value 2, and then to `Eat_Burgers()` with argument value 2, that in `Eater_Jogger`, `Pig_Out()` will increment `cals_Burned` by 40, as specified in Fig. 6.

In [22], we have proposed a set of rules that can be used in the usual fashion of axiomatic semantics to show: first, that the body of `Pig_Out()` defined in Fig. 1 satisfies the grey-box specification (5.1); and second, by using the *enrichment rule* to “plug-in” the richer behavior specified in (4.1) and (4.2) for the redefined hook methods into (5.1), that in the derived class, the template method will satisfy the richer specification (6.1). Here our goal is to test the template method to see whether it satisfies its specification, so we now turn to that.

$$\begin{aligned}
\text{pre.Pig_Out()} &\equiv \tau = \varepsilon & (6.1) \\
\text{post.Pig_Out()} &\equiv \text{cals_Eaten} = \text{cals_Eaten}' + 1200 \\
&\quad \wedge \text{cals_Burned} = \text{cals_Burned}' + 40 \wedge |\tau| = 2 \\
&\quad \wedge \tau[1].\text{method} = \text{"Eat_Donuts"} \wedge \tau[1].\text{arg} = 2 \\
&\quad \wedge \tau[2].\text{method} = \text{"Eat_Burgers"} \wedge \tau[2].\text{arg} = 2
\end{aligned}$$

Fig. 6. Grey-box specification for `Pig_Out` for class `Eater_Jogger`. This can be derived from (4.1), (4.2), and (5.1).

2.3 The Challenge: Testing Grey-Box Behavior without Source Code

If we wished to test `Pig_Out()` against its black-box specification, the task can be carried out in a standard, straightforward fashion [18]. All we would need to do is create an object `ee` of type `Eater`, check that it satisfies the pre-condition given in (2.3) (which in this case is vacuous since it is simply `true`), apply `Pig_Out()` on `ee`, and check, when control returns, whether the post-condition specified in (2.3) is satisfied. But testing the grey-box behavior (5.1) is more complex. First, (5.1) refers to τ , and τ is not an actual variable of the class, but an *auxiliary* variable introduced for the purpose of specification. We can take care of this by introducing a trace variable, call it `tau`, as part of our testing setup and initialize it to the empty sequence immediately before invoking `Pig_Out()`. More seriously, `tau` needs to be updated whenever `Pig_Out()` calls one of the hook methods; else, the value of `tau` will remain as ε and will not satisfy the conditions specified in (5.1) even if in fact `Pig_Out()`'s grey-box behavior is in accordance with (5.1). The obvious way to update `tau` would be to examine the code (in Fig. 1) of `Pig_Out()`, identify all the calls that appear in this code body to hook methods, and insert appropriate instructions into the body of `Pig_Out()` at these points to update `tau` appropriately. Thus we would replace the call `Eat_Donuts(2)` by:

`Eat_Donuts(2); tau = tau ^ (Eat_Donuts, 2);`

where “ $\wedge$ ” denotes appending the specified element to `tau`; calls to `Eat_Burgers()` would be handled similarly. Once we insert these instructions, we go through our testing procedure. When `Pig_Out()` finishes, `tau` would indeed have been updated appropriately, and we can check whether the post-condition in (5.1) is satisfied.

As we saw earlier, each element of the trace should record not just the name of the hook method called and the argument value passed, but also the state of the object at the time of the call as well as when the call returns. Thus what we have is incomplete. This does not matter in this example since the grey-box specification (5.1) does not refer to any of this additional information. In general though, it is necessary to include all of this information in each element of `tau`; and it is straightforward (if a bit tedious) to do this by modifying the above instructions appropriately. But this approach does not meet our requirements. As we have noted before, we may not have access to the source code of the template method we want to test. Therefore, we certainly cannot make changes of this kind. The fundamental problem we have to address is, how do we ensure that the trace `tau` is appropriately updated to record the hook-method calls that

`Pig.out()` makes during its execution, without modifying its code, given that these calls are embedded in that code? In other words, how do we do *black-box* testing of `Pig.Out()`'s grey-box behavior?

3 Black-Box Testing of Grey-Box Behavior

The key problem we face in black-box testing of the grey-box behavior of `Pig.Out()` is that we cannot wait until it finishes execution to try to record information about its hook-method calls since, in general, by that point we no longer have that information. What we need to do instead is to *intercept* these calls as `Pig.Out()` makes them. But how can we do that if we are not allowed to modify `Pig.Out()` at the points of these calls? The answer is provided by the same mechanism that template methods are designed to exploit, i.e., *polymorphism*. That is, rather than intercepting the calls by modifying the code of the template method, we will redefine the hook methods so that *they* update the trace appropriately whenever they are invoked.

In Fig. 7 we define our test class, `Test.Eater`. Since in the post-conditions of methods we are allowed to use, by means of primed variables, the values that variables had when the method started execution, when testing against such specifications we need to save these initial values when a method begins executions. Thus in the `test.Pig.Out()` method of `Test.Eater`, we use `old.cals.Eaten` to save the starting value of `cals.Eaten`.

```
class Test.Eater extends Eater {
  protected trace tau;

  public void Eat_Donuts(int n) { // redefined hook
    traceRec tauel;
    tauel = ... info such as name of method called (Eat_Donuts), param. value (n), etc.
    super.Eat_Donuts(n); // call original hook
    tauel = ... add info about current state, etc.
    tau.append(tauel); }

  // Eat_Burgers() is similarly redefined.

  public void test.Pig.Out() {
    if (true) {
      int old_cals_Eaten = this.cals.Eaten; // allowed, since Test.Eater extends Eater
      tau =  $\varepsilon$ ;
      this.Pig.Out();
      assert(grey-box post-condition of Pig.Out() with appropriate substitutions); } }
}
```

Fig. 7. Class `Test.Eater`.

`Test.Eater` is a derived class of `Eater`, and we have redefined both the hook methods to update the trace. `tau` is the trace variable as before and `tauel` will record information about one hook method call which will be appended to `tau`

once the call has finished and returned. Let us see how `Test_Eater.test_Pig_Out()` works using the *sequence call diagram* [4] in Fig. 8. The six vertical lines, each labeled at the top with the name of a method (the three on the left being from `Test_Eater`, the three on the right from `Eater`), represent time-lines for the respective methods. To test that `Eater.Pig_Out()` satisfies its grey-box specification, we create an appropriate instance of the `Test_Eater` class and apply `Pig_Out()` to it. This call is represented by the solid arrow at the top-left of the figure. The method starts by checking the pre-condition, which is represented by the point labeled with a diamond with a single question mark inside it. (The pre-condition is just *true* in this case.) Next it initializes `tau` to $\langle \rangle$ and saves the initial state in the `old` variable; this point is labeled (1t) in the figure. Next, it calls `Pig_Out()` (on the `this` object). Since `Pig_Out()` is not overridden in `Test_Eater`, this is a call to `Eater.Pig_Out()`, which is represented by the solid arrow from `Test_Eater.test_Pig_Out()` to `Eater.Pig_Out()`. (Note that `Pig_Out()` cannot be overridden in any case, since it is a final method.)

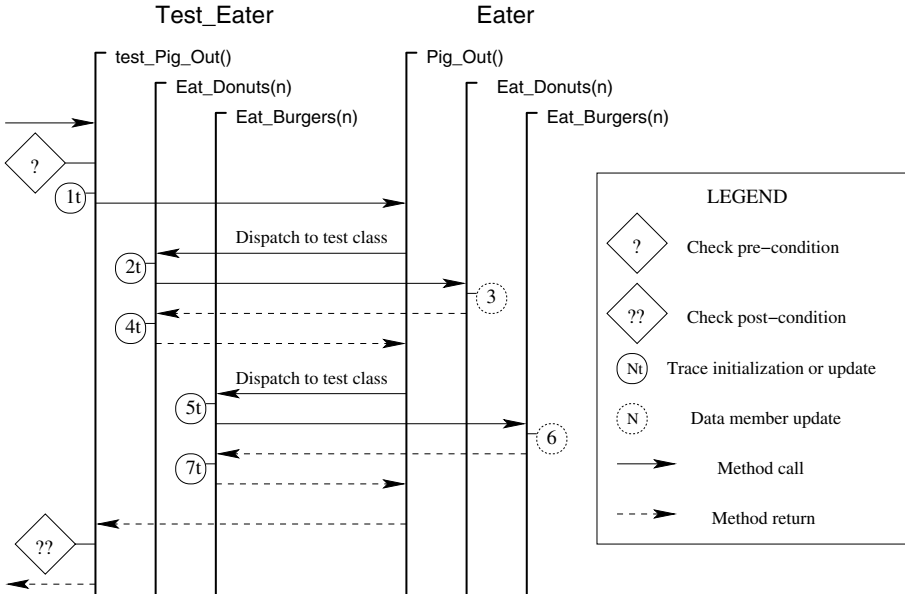


Fig. 8. Sequence Call Diagram for `Test_Eater.Pig_Out()`.

Consider what happens when this method executes. First, it invokes `Eat_Donuts()` which we *have* overridden in `Test_Eater`. This call is dispatched to `Test_Eater.Eat_Donuts()` since the object that `Pig_Out()` is being applied to is of type `Test_Eater`. This dispatch is represented by the solid arrow from the time-line for `Pig_Out()` to that for `Test_Eater.Eat_Donuts()`. Now `Test_Eater.Eat_Donuts()` is simply going to delegate the call to `Eater.Eat_Donuts()` (represented by the arrow from `Test_Eater.Eat_Donuts()` to `Eater.Eat_Donuts()`).

However, before it delegates the call, it records appropriate information about this call on the trace-record variable `tauel`; this action is labeled by (2t) in the figure. Once `Eater.Eat_Donuts()` finishes (after performing its action consisting of updating `Eater.cals.Eaten`, represented by the point labeled (3)), control returns to `Test_Eater.Eat_Donuts()`, represented by the dotted arrow from `Eater.Eat_Donuts()` to `Test_Eater.Eat_Donuts()`. `Test_Eater.Eat_Donuts()` now records appropriate additional information on `tauel` and appends this record to `tau` (represented by the point labeled (4t)), and finishes. Thus, control returns to `Eater.Pig_Out()`, indicated by the dotted arrow from `Test_Eater.Eat_Donuts()` to `Eater.Pig_Out()`. That method next calls `Eat_Burgers()` and this call is again dispatched to `Test_Eater.Eat_Burgers()`, represented by the solid arrow from `Eater.Pig_Out()` to `Test_Eater.Eat_Burgers()`.

The process of recording initial information, delegating the call to the corresponding method in `Eater`, updating `cals.Eaten`, returning from `Eater.Eat_Burgers()`, and appending the results to `tau`, is repeated; these are represented respectively by the point labeled (5t), the solid arrow from `Test_Eater.Eat_Burgers()` to `Eater.Eat_Burgers()`, the point (6), the dotted arrow from `Eater.Eat_Burgers()` to `Test_Eater.Eat_Burgers()`, and the point (7t). At this point `Test_Eater.Eat_Burgers()` finishes, so it returns to `Eater.Pig_Out()`, which is represented by the dotted arrow. That method is also done so it returns to `Test_Eater.test_Pig_Out()`. The final action, the one that we have been building up towards, is to check if the post-condition specified in the grey-box specification (5.1) (with `tau` substituting for τ and `old_cals.Eaten` for `cals.Eaten'`) is satisfied, labeled by the diamond with the double question mark.

Thus by defining `Test_Eater` as a derived class of `Eater`, and by overriding the hook methods of `Eater`, we are able to exploit polymorphism to intercept the calls that the template method makes to the hook methods. This allows us to record information about these calls (and returns) without having to make any changes to the template method being tested, indeed without having any access to the source code of that method. This allows us to achieve our goal of black-box testing of the grey-box behavior of template methods.

It should be noted that `Test_Eater.Eat_Donuts()` is not the test method for testing `Eater.Eat_Donuts()`. If we wished to test that method, we could include a `test_Eat_Donuts()` method in `Test_Eater` that would simply save the starting value of `cals.Eaten`, call the `Eat_Donuts()` method of the `Eater` class, and then assert that the post-condition of (2.1) is satisfied when control returns from that call.

If there were more than one template method, we could introduce more than one trace variable; but since only one template test method will be executing at a time, and it starts by initializing `tau` to $\langle \rangle$, this is not necessary. Consider now the derived class `Eater_Jogger`. How do we construct `Test_Eater_Jogger`? It should be a derived class of `Eater_Jogger`, not of `Test_Eater`, else the redefinitions of the hook methods in `Eater_Jogger` would not be used by the test methods in `Test_Eater_Jogger`. In general, test classes should be *final*. `Test_C` is only intended to test the methods of `C`. Another class `D`, even if `D` is a derived class of `C`, would have its own test class which would be a derived class of `D`.

4 Prototype Implementation

We have implemented a prototype testing system, which is available at <http://www.cis.ohio-state.edu/~tyler>, that creates testing classes described in Sect. 3. The system inputs the grey-box specifications for template methods of the class *C* under test, and the black-box specifications for the non-template methods. The system then creates the source code for the test class, along with other adjunct classes needed for the testing process, in particular those used in constructing traces when testing the template methods of *C*. The methods to be treated as hooks must be explicitly identified so that they are redefined in the test class. An alternate approach would have been to treat *all* non-final methods as hooks; but our approach allows greater flexibility. Each redefined hook method that the tool produces also checks its pre- and post-condition before and after the dispatched call is made. This helps pinpoint problems if a template method fails to satisfy its post-condition.

Currently, our system does not generate test cases, but creates skeleton calls to the test methods, where the user is required to construct test values by hand. To do the actual testing, the generated classes are compiled, and the test class executed. An example of the system's output is in Fig. 9. The last output shows a case where the grey-box specification was not met. The problem was that the compiled *Eater* class had a bug in the code of *Pig_Out()*: it passed 4 as the parameter to *Eat_Donuts()* and 1 as the parameter to *Eat_Burgers()*; hence, although the black-box specification of *Pig_Out()* was satisfied, its grey-box specification was not.

```

Test number 1: testing Eat_Donuts.
Test number 1 succeeded!

Test number 2: testing Eat_Burgers.
Test number 2 succeeded!

Test number 3: testing Pig_Out.
  Method Eat_Donuts called.
  Method Eat_Burgers called.
Postcondition of Pig_Out not met!
tau = ("Eat_Donuts", 4, 4), ("Eat_Burgers", 1, 1))
Test number 3 failed!
* * * RESULTS * * *
Number of tests run: 3
Number of tests successful: 2

```

Fig. 9. Output from sample run.

5 Discussion and Future Work

Our work was motivated by two observations: First, given that perhaps the most important aspect of template methods is the hook method call patterns

they implement, testing such methods requires us to test against their grey-box specifications. Second, application developers often build their systems using COTS components, including frameworks. If this developer wishes to test such a component, she will have to do so without having access to the source code of the component; Weyuker [23] also notes the importance of testing COTS components without having access to their source code. The approach we have developed addresses both of these considerations.

We conclude with some pointers for future work. We have ignored abstraction so far, instead working directly with the data members of the class under test. Cheon and Leavens [7] describe a testing system that can work with specifications that are given in terms of a conceptual model of the class under test. They do not consider grey-box specifications but we believe their approach can be extended to deal with grey-box behavior and we intend to explore that.

A more serious question is that of generating appropriate test cases to achieve reasonable coverage. As we noted earlier, our prototype system requires the human tester to provide the test cases. One interesting approach for generating test cases is used by TestEra [16] which is a system for specification-based testing Java programs. This system allows us to define, using a first-order relational language, complex properties that the objects must meet. Given a specification written in this notation, the system automatically generates instances that satisfies the pre-condition, so that we can then apply the method under test on the object in question. If the specification can be violated, TestEra generates a test case that shows that. The specifications that TestEra works with are black-box specifications; we plan to investigate whether a similar approach can be used to deal with grey-box specifications.

Another important question relates to the scalability of the methodology when applied to systems whose traces contain complex objects. The current prototype attempts to generate suitable `clone()` (i.e., deep copy) methods when they are not provided by the user in order to save the object state. Sufficiently complex objects coupled with long traces may require a prohibitive amount of memory during testing. One possible way to lessen the problem is to save only values and object references that are mentioned in the specifications, instead of copying whole objects. (This is being done in the latest version of the tool.) Another possibility is to only store information about the particular changes made to objects during execution. These issues must be addressed before our testing systems are useable in practical settings.

References

1. R. Alexander and J. Offutt. Criteria for testing polymorphic relationships. In *Int. Symp. on Softw. Reliability Eng.*, pages 15–23, 2000.
2. M. Barnett, W. Grieskamp, C. Kerer, W. Schulte, C. Szyperski, N. Tilmann, and A. Watson. Serious specifications for composing components. In *6th ICSE Workshop on Component-based Software Engineering*, pages 1–6, 2003.
3. R. Binder. *Testing Object-Oriented Systems*. Addison-Wesley, 1999.

4. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
5. M. Buchi and W. Weck. The greybox approach: when blackbox specifications hide too much. Technical Report TUCS TR No. 297, Turku Centre for Computer Science, 1999. available at <http://www.tucs.abo.fi/>.
6. M.-H. Chen and H. M. Kao. Testing object-oriented programs — an integrated approach. In *Int. Symp. on Softw. Reliability Eng.*, pages 73–83, 1999.
7. Y. Cheon and G. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *Proc. of ECOOP 2002*, pages 231–255. Springer-Verlag LNCS, 2002.
8. J. Duran and S. Ntafos. An evaluation of random testing. *IEEE Trans. on Software Eng.*, 10:438–444, 1984.
9. M.E. Fayad and D.C. Schmidt. Special issue on object oriented application frameworks. *Comm. of the ACM*, 40, October 1997.
10. G. Froehlich, H. Hoover, L. Liu, and P. Sorenson. Hooking into object-oriented application frameworks. In *Proc. of 1997 Int. Conf. on Software Engineering*, pages 141–151. ACM, 1997.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable OO Software*. Addison-Wesley, 1995.
12. D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Trans. on Software Eng.*, 16(12):1402–1411, 1990.
13. R. Johnson and B. Foote. Designing reusable classes. *Journal of OOP*, 1:26–49, 1988.
14. C. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, 1990.
15. B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Trans. on Prog. Lang. and Systems*, 16:1811–1841, 1994.
16. D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proc. of 16th ASE*. IEEE, 2001.
17. R. McDaniel and J. D. McGregor. Testing the polymorphic interactions between classes. Technical Report 94-103, Dept. of Computer Sc., Clemson University, 1994.
18. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.
19. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(1):319–340, 1976.
20. W. Pree. Meta patterns: a means for capturing the essentials of reusable OO design. In *Proceedings of the Eighth ECOOP*, pages 150–162, 1994.
21. A. Rountev, A. Milanova, and B. Ryder. Fragment class analysis for testing of polymorphism in Java software. In *Int. Conf. on Softw. Eng.*, pages 210–220, 2003.
22. N. Soundarajan and S. Fridella. Framework-based applications: From incremental development to incremental reasoning. In W. Frakes, editor, *Proc. of Sixth Int. Conf. on Software Reuse: Advances in Software Reusability*, LNCS 1844, pages 100–116. Springer, 2000.
23. E. Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, 15(5):54–59, 1998.

On Checking Whether a Predicate Definitely Holds^{*}

Alper Sen and Vijay K. Garg

Dept. of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX, 78712, USA

{sen, garg}@ece.utexas.edu
<http://www.ece.utexas.edu/~{sen, garg}>

Abstract. Predicate detection is an important problem in testing and debugging distributed programs. Cooper and Marzullo introduced two modalities *possibly* and *definitely* as a solution to this problem. Given a predicate p , a computation satisfies *possibly*: p if p is true for some global state in the computation. A computation satisfies *definitely*: p if *all* paths from the initial to the final global state go through some global state that satisfies p . In general, *definitely* modality is used to detect *good* conditions such as “a leader is eventually chosen by all processes”, or “a commit point is reached by every process”, whereas *possibly* modality is used to detect bad conditions such as violation of mutual exclusion. There are several efficient algorithms for *possibly* modality in the literature [10,14,1,2,30]. However, this is not the case for *definitely* modality. Cooper and Marzullo’s *definitely*: p algorithm for arbitrary p has a worst-case space and time complexity exponential in the number of processes. This is due to the state explosion problem. In this paper we present efficient algorithms for detecting *definitely*: p . In particular, we give a simple algorithm that uses polynomial space. Then, we present an algorithm that can significantly reduce the global state-space. We determine necessary conditions and sufficient conditions under which detecting *definitely*: p may be efficiently solved. We apply our algorithms to example protocols, achieving a speedup of over 100, compared to partial order reduction based technique of SPIN [13].

1 Introduction

A fundamental problem in distributed computing is *predicate detection* — deciding whether an execution trace of a distributed program satisfies a given predicate. This problem arises in many contexts such as testing and debugging of distributed programs. For example, when debugging a distributed mutual exclusion algorithm, it is useful to monitor the system to detect concurrent accesses to the shared resources.

Cooper and Marzullo introduced two modalities for predicate detection, which are denoted by *possibly* and *definitely*. Given a predicate p , a computation satisfies *possibly*: p if p is true for some global state in the computation. A computation satisfies *definitely*: p if *all* paths from the initial state to the final global state go through some global state that satisfies p . In general, *possibly* modality is used to detect *bad* conditions such as the system reaches a global state where the mutual exclusion predicate

^{*} supported in part by the NSF Grants ECS-9907213, CCR-9988225, Texas Education Board Grant ARP-320, an Engineering Foundation Fellowship, and an IBM grant.

is false. In contrast, *definitely* modality is in general used to detect *good* conditions such as “a leader is eventually chosen by all processes”, or “a commit point is reached by every process”. Cooper and Marzullo’s definitions of these modalities established an important conceptual framework for predicate detection, which has been the basis of considerable research. However, most of the research has focused on *possibly* modality [10,14,1,2,30].

Cooper and Marzullo present an algorithm for detecting *definitely: p* for arbitrary predicate p . The worst-case space and time complexity of their algorithm is exponential in the number of processes. This is due to the *state explosion problem*—in a distributed system of n processes, the number of possible global states (state-space) can be of size $O(m^n)$, where m is the maximum number of events on a process.

This paper presents efficient algorithms for detecting *definitely: p*. We first present a simple algorithm for *definitely: p* that uses $O(nm)$ space in Section 4. Then, we present a polynomial-time state-space reduction algorithm that enables us to work on a distributed computation that is in general much smaller than the original computation. We prove that the original computation satisfies *definitely: p* if and only if the smaller computation satisfies it. It is, in general, coNP-complete to detect a predicate under *definitely* modality [28]. In Sections 5 and 6, we determine necessary conditions and sufficient conditions under which detecting *definitely: p* may be efficiently solved. In order to develop these conditions, we use lattice theoretic properties of distributed computations. We validate the effectiveness of our algorithms with experimental studies in Section 7. For this purpose, we implement our algorithms in the Partial Order Trace Analyzer (POTA) tool [27] and compare performance to partial order reduction based algorithms of model checker SPIN [13]. In one case, our algorithms are significantly faster and space efficient. We have measured over 100-fold gain.

Our work constitutes part of the POTA tool [27,23] for testing distributed program execution traces using temporal logic predicates. Figure 1 displays an overview of POTA architecture. POTA consists of an instrumentation module, a translator module that translates execution traces into Promela [13] (SPIN model checker input language) and an analyzer module. The use of partial order model for execution traces and the use of an effective abstraction technique for temporal logic verification called computation slicing are significant aspects of POTA and constitutes the analyzer module. POTA implements polynomial-time temporal logic predicate detection algorithms. The temporal logic used in POTA is a subset of CTL [3]. With the results of this paper, we extend efficient predicate detection algorithms in POTA with *definitely* operator. Atomic propositions of the logic used in POTA are regular predicates, which widely occur in practice during verification. Some examples of regular predicates are conjunction of local predicates [8,15] such as “all processes are in *red* state”, certain channel predicates [8] such as “at most k messages are in transit from process P_i to P_j ”, and some relational predicates [8].

2 Related Work

Our approach exploits the structure of the predicate itself — by imposing restrictions — to evaluate its value efficiently for a given computation. Polynomial-time algorithms for *possibly: p* have been developed when p belongs to conjunctive [10,14], observer-

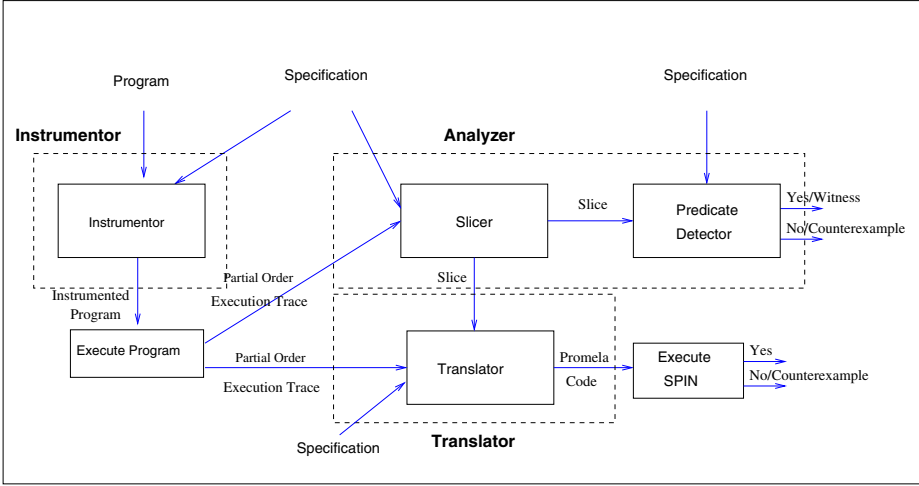


Fig. 1. Overview of POTA Architecture

independent [1], linear [2], and relational predicates [30]. Also in [22] there is an extensive survey on predicate detection techniques.

Tarafdar and Garg [28] proved that it is, in general, NP-complete to detect a predicate under *controllable* modality. A computation satisfies *controllable*: p if every state on some path from the initial global state to the final global state satisfies p . Since the problem of detecting a predicate under *definitely* modality is the dual of the problem of detecting a predicate under *controllable* modality, it is, in general, coNP-complete to detect a predicate under *definitely* modality. Using Tarafdar and Garg's [29] NP-completeness result for controlling a special case of 2-CNF predicates, called *independent mutual exclusion* predicates, we can easily deduce that detecting a special case of 2-DNF predicates, which is the dual of independent mutual exclusion predicates, under *definitely* modality is coNP-complete in general.

Fromentin and Raynal [7] presented a polynomial-time algorithm to solve the predicate detection problem for *proper* modality, which is a special case of *definitely*. A computation satisfies *proper*: p if all paths from the initial state to the final global state go through a *unique* global state that satisfies p .

The *definitely*: p problem has efficient solutions when the predicate is 1-CNF or 1-DNF [8]. However, the complexity problem is open for *definitely*: p for regular p . In this paper, we present efficient conditions to solve the problem for both arbitrary and regular predicates.

The idea of using temporal logic in program testing has been applied in several tools such as the commercial Temporal Rover tool (TR) [6], the MaC tool [17], and the JPaX tool [12]. TR allows the user to specify the temporal formula in programs. These temporal formula are translated into Java code before compilation. The MaC and JPaX tools consider a totally ordered view of an execution trace and therefore can potentially miss bugs that can be deduced from a partial order view of the trace. Hallal et al. in [11] uses a partial order view of an execution trace as in POTA. They translate

execution traces into SDL and use commercial SDL tools for testing translated traces. POTA incorporates several polynomial-time (polynomial in the number of processes) predicate detection algorithms whereas the complexity is exponential-time in [11].

3 Model

We assume a loosely-coupled message-passing asynchronous system without any shared memory or a global clock. A *distributed program* consists of n sequential processes denoted by $P_1, P_2, \dots, P_n$ communicating via asynchronous messages. In this paper, we are concerned with a single *computation (execution)* of a distributed program. We assume that no messages are altered or spuriously introduced. We do not make any assumptions about FIFO nature of channels.

The execution of a process in a computation can be viewed as a sequence of events with events across processes ordered by Lamport's *happened-before* relation, $\rightarrow$ [18]. We use lowercase letters e and f to represent events. The *happened-before* relation between any two events e and f can be formally stated as the smallest relation such that $e \rightarrow f$ if and only if e occurs before f in the same process, or e is a send of a message and f is a receive of that message, or there exists an event g such that e happened-before g and g happened-before f . We represent the set of events as the union of events from each process, $E = \bigcup E_i$, for each $1 \leq i \leq n$. We define a *distributed computation* as the partially ordered set consisting of the set of events together with the happened-before relation and denote it by $\langle E, \rightarrow \rangle$.

We define a *consistent cut* of a computation $\langle E, \rightarrow \rangle$ as a subset $G \subseteq E$ such that $f \in G \wedge e \rightarrow f \Rightarrow e \in G$. We use uppercase letters G, H, J , and K to represent consistent cuts. A consistent cut captures the notion of a reachable global state. We use consistent cut and global state interchangeably. We denote the set of consistent cuts of any distributed computation $\langle E, \rightarrow \rangle$ by $C(E)$. It is well-known that the set of consistent cuts of any distributed computation $\langle E, \rightarrow \rangle$ forms a *distributive lattice*, under the relation $\subseteq$ [19,9]. We denote this lattice by $L = (C(E), \subseteq)$ and also call this as the *state-space* of the distributed computation. For any partially ordered set, we use $\sqcup$ and $\sqcap$ to denote join and meet operators. Note that the join (resp. meet) of two consistent cuts correspond to their union (resp. intersection). We use $\perp$ to denote the initial consistent cut, E to denote the final consistent cut of all processes, and $\top$ to denote a fictitious final cut occurring after E .

We denote the set of maximal (with respect to happened-before relation) elements of a consistent cut G by $\text{frontier}(G)$. Figure 2 shows a computation and its lattice of consistent cuts. A consistent cut in the figure is represented by its frontier. For example, the consistent cut $\{e_3, e_2, e_1, f_2, f_1, \perp\}$ is represented by $\{e_3, f_2\}$. A consistent cut H is *reachable* from a consistent cut G iff it is possible to attain H from G by executing zero or more events. It is easy to see that H is reachable from G iff $G \subseteq H$. We define *successor* of a cut by a relation $\triangleright \subseteq C(E) \times C(E)$ such that $G \triangleright H$ if and only if $H = G \cup \{e\}$ for some $e \in E$ such that $e \notin G$. We say that H is a *successor* of G and G is a *predecessor* of H . A *path* $G_0, G_1, \dots, G_l$ of $(C(E), \subseteq)$ satisfies that for each $0 \leq i < l$, $G_i \triangleright G_{i+1}$.

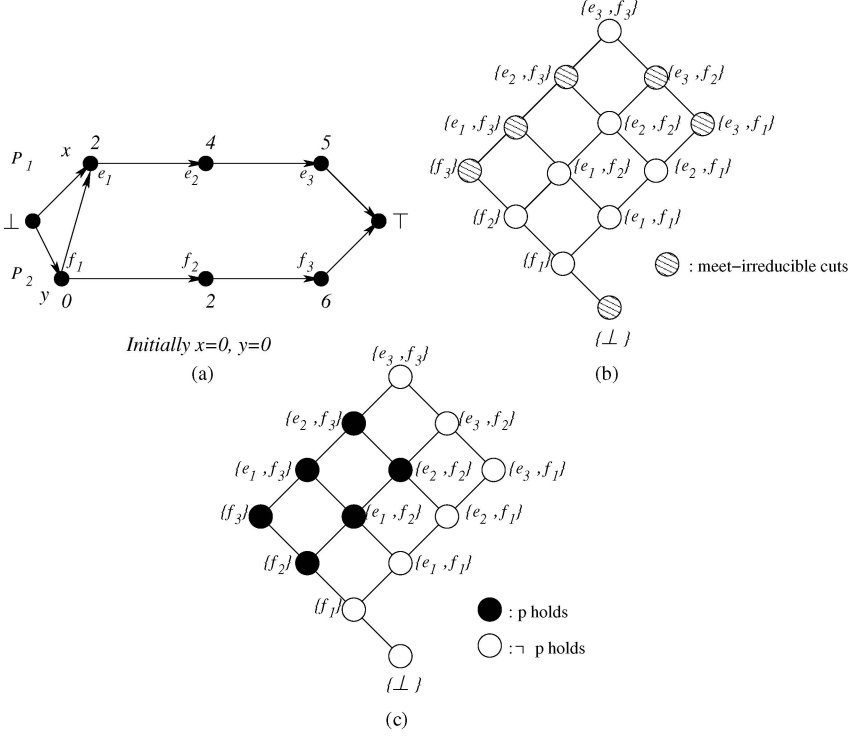


Fig. 2. (a) A computation (b) meet-irreducible cuts (c) corresponding lattice of the computation

A predicate is defined as a boolean-valued function on variables of processes. Given a consistent cut, a predicate is evaluated with respect to the values of variables resulting after executing all events in the cut. If a predicate p evaluates to true for a consistent cut C , we say that “ C satisfies p ”. We leave the predicate undefined for $\top$. A global predicate is *local* if it depends on variables of a single process.

We say that a predicate is *regular* if the set of consistent cuts that satisfy the predicate forms a sublattice of the lattice of consistent cuts. Equivalently, if two consistent cuts satisfy a regular predicate then the cuts given by their set intersection and set union also satisfies the predicate. Let $\inf(p)$ and $\sup(p)$ denote the least and the greatest consistent cut that satisfies a given predicate p , respectively. From the definition of a regular predicate we deduce that both $\inf(p)$ and $\sup(p)$ exist for a regular predicate. There are efficient algorithms for detecting regular predicates under *possibly* and *controllable* modalities [9,25].

4 Polynomial-Space Algorithm

The performance of algorithms for detecting *definitely*: p can be improved by considering a smaller state-space, that is, a smaller computation than the original computation.

In this section, we present a polynomial-time algorithm for reducing the size of the computation. We show that detecting *definitely: p* on the original computation is the same as detecting *definitely: p* on the smaller computation. For this purpose, we define an *interval* of a computation $\langle E, \rightarrow \rangle$ with respect to consistent cuts C and D as the computation $interval(C, D)$, which is a subset of $\langle E, \rightarrow \rangle$ and only the cuts between C and D (including C and D) of $\langle E, \rightarrow \rangle$ belong to $interval(C, D)$.

We state informally a lemma before presenting our state-space reduction algorithm. Given three consistent cuts G, H and J , where H is reachable from J and H is a successor of G , the intersection of G and J is either J or it is a predecessor of J . We present the proofs in the extended version of this paper [26].

Theorem 1 (NSC). *Given a computation $\langle E, \rightarrow \rangle$, *definitely: p* holds in $\langle E, \rightarrow \rangle$ iff *definitely: p* holds in $interval(C, D)$, where C is the meet of predecessors of $inf(p)$, if the predecessors exist, otherwise $inf(p)$, and D is the join of successors of $sup(p)$, if the successors exist, otherwise $sup(p)$.*

Proof. Without loss of generality, assume that both C and D exist and are different from the initial and final consistent cut of $\langle E, \rightarrow \rangle$. We prove the contrapositives.

$\Rightarrow$:

We obtain a path from the initial consistent cut to the final consistent cut in $\langle E, \rightarrow \rangle$ as follows: Pick an arbitrary path from the initial consistent cut of $\langle E, \rightarrow \rangle$ to C . We know that none of the cuts on this path satisfy p since all cuts that satisfy p belong to $interval(C, D)$. Next, using the assumption, continue this arbitrary path with a path in $interval(C, D)$ where none of the cuts on the path satisfy p . Finally, pick an arbitrary path from D to the final consistent cut of $\langle E, \rightarrow \rangle$.

$\Leftarrow$:

Now we prove that if there exists a path from the initial to the final cut in $\langle E, \rightarrow \rangle$ where all cuts on the path satisfy $\neg p$ then there exists a path from the initial to the final consistent cut in $interval(C, D)$ where all cuts on the path satisfy $\neg p$. We prove the claim in two Steps.

Step 1: We first show that if there exists a path, $\mathcal{P}$, from the initial to the final consistent cut in $\langle E, \rightarrow \rangle$ where all cuts on the path satisfy $\neg p$ then there exists a path from the initial to the final cut in $interval(C, E)$ where all cuts on the path satisfy $\neg p$.

Let β be the first cut on the path $\mathcal{P}$ such that $inf(p) \subseteq \beta$. Let β' be the predecessor of β on the path $\mathcal{P}$. From the lemma stated above, the meet of β' and $inf(p)$ is either $inf(p)$ or a predecessor of $inf(p)$, say C' . However, if the meet is $inf(p)$ then $inf(p) \subseteq \beta'$. Since β' is also on the path $\mathcal{P}$ we have that β' is the first cut on the path $\mathcal{P}$ such that $inf(p) \subseteq \beta$. This is a contradiction since β is the first such cut. Therefore the meet of β' and $inf(p)$ is C' .

There exists a path from C' to β' because $C' \subseteq \beta'$. Furthermore every cut on this path satisfies $\neg p$. We prove this as follows. From the definition of $interval(C, D)$, only cuts in $interval(inf(p), sup(p))$ satisfy p . Now consider all cuts F such that $C' \subseteq F \subseteq \beta'$. We have that $inf(p) \not\subseteq \beta'$ and therefore $inf(p) \not\subseteq F$. Therefore, β' and all such F do not satisfy p . Since C is the meet of all predecessors of $inf(p)$ and C' is a predecessor of $inf(p)$, $C \subseteq C'$ and therefore $C \subseteq F$. Also, all cuts from C to C' satisfy $\neg p$ since none of them belong to $interval(inf(p), sup(p))$.

We obtain the required path as follows. Choose an arbitrary path from C to C' , then continue the path from C' to β' and then to β . Continue the path from β to the final cut with the same path from β to the final cut as in path $\mathcal{P}$.

Step 2: Now we show that if there exists a path, $\mathcal{P}$, from the initial to the final consistent cut in $interval(C, E)$ where all cuts on the path satisfy $\neg p$ then there exists a path from the initial to the final consistent cut in $interval(C, D)$ where all cuts on the path satisfy $\neg p$.

The proof is similar to Step 1 with the paths reversed. In this case we choose β as the last cut on the path $\mathcal{P}$ such that $\beta \subseteq sup(p)$ and β' as the successor of β on the path $\mathcal{P}$. Furthermore, we choose D' as a successor of $inf(p)$. We can show in a similar fashion as in Step 1 that there exists a path from β' to D' where all cuts on the path satisfy $\neg p$. Finally, we can construct a path from C to D as the concatenation of the paths from C to β , β to β' , β' to D' , and D' to D . $\square$

We can compute $interval(C, D)$ by computing $inf(p)$ and $sup(p)$ in $O(n|E|)$ time for regular p [9]. Similarly, we can compute the predecessors and successors of a cut in $O(n)$ time. Note that the above theorem is not restricted to predicates with a single least and greatest cut only. For example, if the predicate has several least cuts then first we take the intersection of all those cuts; second, we find the predecessors of the intersection; and finally, we compute the intersection of the predecessors to obtain C .

Although the time complexity of computing $interval(C, D)$ is polynomial, the time and space complexity of detecting $definitely: p$ on this reduced state-space may be exponential since $interval(C, D)$ may contain exponential number of global states. However, it is always better to work on $interval(C, D)$ rather than $\langle E, \rightarrow \rangle$ since $interval(C, D)$ is a subset of $\langle E, \rightarrow \rangle$. In fact, we believe that $interval(C, D)$ is generally much smaller than the original computation $\langle E, \rightarrow \rangle$ and we validate this belief with experimental work. Furthermore, Theorem 1 is orthogonal to the conditions we will present for detecting $definitely: p$, that is, we can always first compute $interval(C, D)$ and then apply those conditions.

Next, we present a polynomial-space algorithm for $definitely: p$. Cooper and Marzullo [4] presented a worst case exponential-space and time algorithm when they introduced $definitely: p$. Their algorithm detects $definitely: p$ using level sets where a *level set* is the set of successors of a consistent cut. The algorithm starts from the initial consistent cut. If p is true in the initial consistent cut we are done. Otherwise, it constructs the next level set including only those consistent cuts in which $\neg p$ is true. Continuing in this manner, if the algorithm can reach the final consistent cut, then $definitely: p$ is false; otherwise, it is true. This algorithm requires space proportional to the size of the largest level set, which is exponential. We obtain a simple space efficient algorithm for detecting $definitely: p$ by generating all paths of cuts for the given computation. This algorithm is based on generating linearizations of a partial order [21]. For each such path, we check whether $\neg p$ holds on every cut on the path. If such a path exists then $definitely: p$ is not satisfied otherwise it is satisfied. The length of every path is at most $|E|$, the total number of events in the system. A frontier of a consistent cut can be represented by an n -dimensional vector. Therefore, for each consistent cut $O(n)$ space is required giving us the space complexity of $O(n|E|)$. The time complexity is bounded by the number of

paths, which may be exponential in the number of processes. We can improve the time complexity using computation slicing technique explained later in this paper.

Figure 3 shows a polynomial-space *definitely: p* algorithm that uses the techniques developed in this section.

Input: A computation $\langle E, \rightarrow \rangle$ and a predicate p
Output: *definitely: p* is satisfied or not

1. compute $\text{inf}(p)$ and $\text{sup}(p)$;
2. let C be the intersection of predecessors of $\text{inf}(p)$;
3. let D be the union of successors of $\text{sup}(p)$;
4. use C and D to obtain $\text{interval}(C, D)$; // reduce the number of global states
5. for each path in $\text{interval}(C, D)$ do // obtain paths using [21]
6. let G be the first cut on the path
7. while G satisfies $\neg p$ do
8. $G :=$ successor of G on the path;
9. endwhile;
10. if $G = D$ then // final cut is reached
11. return false;
12. endif;
13. endfor;
14. return true;

Fig. 3. A polynomial-space algorithm for detecting *definitely: p*

5 Polynomial-Time Necessary Conditions

Now we present a polynomial-time necessary condition to detect *definitely: p* that uses meet-irreducible cuts [5]. We say that a cut is *meet-irreducible* if it has only one successor consistent cut. For example, the predecessors of the final consistent cut of a computation (e.g. predecessors of $\{e_3, f_3\}$ in Figure 2(b)) are all meet-irreducible cuts. The number of meet-irreducible cuts of a distributive lattice is generally exponentially smaller than the number of all cuts in the lattice. In fact, for a finite distributive lattice, the number of meet-irreducible cuts is exactly equal to the size of the longest chain in the lattice [5]. In our case, the length of the longest chain is equal to the number of events $|E|$. Hence, if some computation can be done on meet-irreducible cuts, we get a significant computational advantage.

Theorem 2 (NC). *Given a computation $\langle E, \rightarrow \rangle$ and a regular predicate p , if $\neg p$ holds at the initial consistent cut and at the successor of every meet-irreducible cut then *definitely: p* does not hold in $\langle E, \rightarrow \rangle$.*

Proof. We show that there exists a path from the initial to the final consistent cut in the computation $\langle E, \rightarrow \rangle$ where all cuts on the path satisfy $\neg p$. Given an arbitrary consistent cut C that satisfies $\neg p$ and different from the final consistent cut, we first show that there exists a successor of C that satisfies $\neg p$. There are two cases.

Case 1: C has a single successor. In this case C is a meet-irreducible cut and from the assumption $\neg p$ holds at the successor of C .

Case 2: C has at least two successors. Observe that if more than one successor of C satisfies p then from the regularity of p , the intersection of those successor cuts, which is C , satisfies p . This leads to a contradiction. Therefore, there exists at least one successor of C where $\neg p$ holds.

We construct the path as follows: From the assumption, $\neg p$ holds at the initial cut. From above we have that for every consistent cut that satisfies $\neg p$ we can find a successor consistent cut that satisfies $\neg p$. Finally, we reach the final consistent cut which is the successor of a cut that satisfies $\neg p$. $\square$

The converse of Theorem 2 is false. Figure 2(c) displays the lattice of consistent cuts of the computation in Figure 2(a). From the lattice we observe that this computation satisfies the right side of Theorem 2. However, the left side of the theorem does not hold because the successor of the meet-irreducible cut $\{f_3\}$ satisfies p . A similar condition can be given for join-irreducible cuts. A *join-irreducible* cut of a distributive lattice is such that it has only one predecessor consistent cut. Meet and join-irreducible cuts are duals of each other.

Theorem 3. *Given a computation $\langle E, \rightarrow \rangle$ and a regular predicate p , if $\neg p$ holds at the final consistent cut and at the predecessor of every join-irreducible cut then definitely: p does not hold in $\langle E, \rightarrow \rangle$.*

We can check Theorem 2 (resp. Theorem 3) by finding the meet-irreducible (resp. join-irreducible) cuts of the computation in $O(n^2|E|)$ time for regular p [24].

Next we present another polynomial-time condition for detecting *definitely: p* based on the notion of intervals introduced earlier. We say that a predicate is an *interval predicate* if there exists a unique initial cut, C , and a unique final cut, D , that satisfies the predicate and the predicate holds in *all* cuts between C and D . An interval predicate with the initial and final cuts C and D defines an *interval*(C, D) in a computation $\langle E, \rightarrow \rangle$. Observe that *interval*(C, D) may partition the lattice of consistent cuts of a computation as in Figure 4. The patterned region in the figure denotes the cuts that belong to *interval*(C, D), i.e., the set of cuts that satisfy the interval predicate. A cut F belongs to partition I if $C \not\subseteq F \subseteq D$, partition II if $C \not\subseteq F \not\subseteq D$, partition III if $C \subseteq F \subseteq D$, and partition IV if $C \subseteq F \not\subseteq D$. Given that *interval*(C, D) exists, that is, partition III exists, other partitions may not exist. For example, if C is the initial consistent cut of $\langle E, \rightarrow \rangle$ and D is the final consistent cut of $\langle E, \rightarrow \rangle$ then only partition III exists.

Theorem 4. *Given a computation $\langle E, \rightarrow \rangle$ and an interval predicate p with *interval*(C, D), there exists a consistent cut F that belongs to partition II in $\langle E, \rightarrow \rangle$ iff definitely: p does not hold in $\langle E, \rightarrow \rangle$.*

Proof. $\Rightarrow$:

We know that F is reachable from the initial cut. For the purpose of contradiction, assume that there exists a cut H on a path from $\perp$ to F such that H satisfies p . For F to be reachable from H , we must have that $H \subseteq F$. However since H satisfies p , $C \subseteq H$ and since F is in partition II, $C \not\subseteq F$, therefore we have a contradiction. Similarly, we can show that there does not exist a cut H' on a path from F to E such that H' satisfies p . C cannot be $\perp$ and D cannot be E because we assume that partition II exists. Therefore, partitions I and IV also exist. Now we obtain a path where all cuts satisfy $\neg p$ by starting

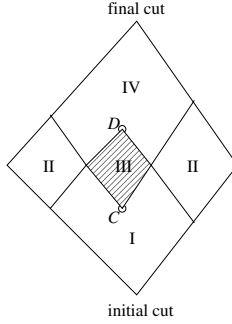


Fig. 4. $interval(C, D)$ partitions the lattice of consistent cuts

from $\perp$ and following an arbitrary path in partition I such that the path reaches F in partition II. Then we follow an arbitrary path from F to the final consistent cut.

$\Leftarrow$:

We prove by contradiction. Suppose that partition II does not exist and there exists a path in $\langle E, \rightarrow \rangle$ from initial to the final consistent cut where all cuts on the path satisfy $\neg p$. Since there exists such a path, we have that partitions I and IV exist. Otherwise, $C = \perp$ and $D = E$ and we do not have a path from $\perp$ to E where $\neg p$ holds on the path. Since partition II does not exist and a path of cuts satisfying $\neg p$ exists, there is a path from partition I to partition IV without passing through partition III (since p is an interval predicate). We will show that this is impossible.

Consider two cuts, F and H , on a path from $\perp$ to E where $\neg p$ holds on the path, such that F belongs to partition I and H belongs to partition IV and H is a successor of F . From the definition of partitions, we have that $C \not\subseteq F \subseteq D$ and $C \subseteq H \not\subseteq D$. Furthermore, from the definition of successor of a cut, we know that $H = F \cup \{e\}$, where e is an event in $\langle E, \rightarrow \rangle$ and $e \notin F$. To obtain H from F , there are two cases: On one hand, we should add $e \notin D$ to F (therefore $e \notin C$) so that $H \not\subseteq D$. On the other hand, we should add $e \in C$ to F (therefore $e \in D$) so that $C \subseteq H$. However, $e \in D$ and $e \notin D$ leads to a contradiction. $\square$

We present a weaker result for regular predicates. The necessary conditions of Theorem 2 and 3 are not comparable with the condition of Theorem 5 below. Furthermore, observe that the converse of the next condition is false.

Theorem 5. *Given a computation $\langle E, \rightarrow \rangle$, and a regular predicate p with $interval(C, D)$, where $C = inf(p)$ and $D = sup(p)$, if there exists a consistent cut F that belongs to partition II in $\langle E, \rightarrow \rangle$ then definitely: p does not hold in $\langle E, \rightarrow \rangle$.*

We can use a technique called slicing, which we explain next, to detect whether there exists a consistent cut F in partition II. The overall complexity of checking the existence of F using slicing is $O(n^2|E|^2)$ [20].

6 Polynomial-Time Sufficient Condition

We have advocated the use of a technique called *computation slicing* for predicate detection in [9,20,25]. The notion of computation slice is based on Birkhoff's Representation

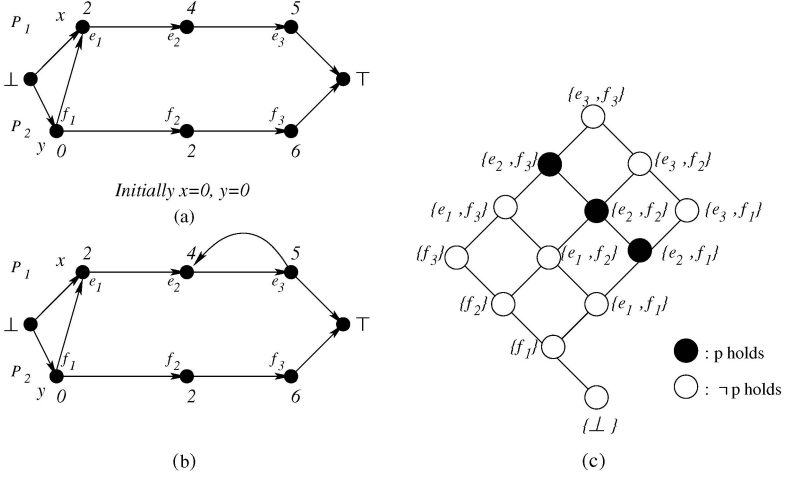


Fig. 5. (a) A computation (b) slice wrt $x \neq 4$ (c) lattice of consistent cuts of the computation

Theorem for Finite Distributive Lattices [5]. The readers who are not familiar with earlier papers on slicing [9,20,25] are strongly urged to read the extended version of this paper in [26]. We also use a directed graph model of a computation to handle both computations and computation slices in a uniform and convenient manner. In this model, a distributed computation $\langle E, \rightarrow \rangle$ is a directed graph with vertices as the set of events and edges as $\rightarrow$. A subset of vertices forms a consistent cut if the subset contains a vertex only if it also contains all its incoming neighbours. Observe that a consistent cut either contains all vertices in a strongly connected component or none of them. Roughly speaking, a computation slice (or simply a slice) is a concise representation of all those consistent cuts of the computation that satisfy the predicate. More precisely,

Definition 1 (slice [20]). A slice of a computation with respect to a predicate is a directed graph with the least number of consistent cuts that contains all consistent cuts of the given computation for which the predicate evaluates to true.

We denote the slice of a computation $\langle E, \rightarrow \rangle$ with respect to a predicate p by $\text{slice}(\langle E, \rightarrow \rangle, p)$. It was shown in [20] that the slice exists and is uniquely defined for all predicates. Intuitively, the consistent cuts that belong to the slice are obtained by computing the union and intersection closure of the cuts in the computation that satisfy the predicate. In other words, if two cuts G and H satisfy p , then the slice contains cuts $G \sqcup H$ and $G \sqcap H$ too.

Given a computation as in Figure 5(a), and a regular predicate p , such as $(x = 4)$, where x is a local variable defined on process P_1 , now we consider the slice of the computation with respect to $\neg p$ as displayed in Figure 5(b). The consistent cuts that belong to the slice are denoted by white filled circles in Figure 5(c). Note that in this example the cuts that belong to the slice are already closed under union and intersection. We make the following two observations on the computation and its slice. First, consider the cuts in the computation. On every path from the initial to the final

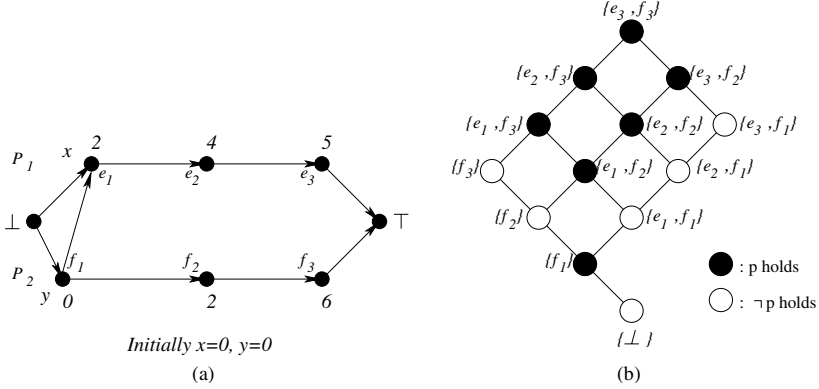


Fig. 6. (a) A computation (b) corresponding lattice

consistent cut there is a consistent cut that contains event e_2 but not e_3 . These cuts are $\{e_2, f_1\}, \{e_2, f_2\}, \{e_2, f_3\}$. Furthermore, all of these cuts satisfy p . Second, consider the cuts in the slice, when the slice contains a non-trivial strongly connected component, such as $\{e_2, e_3\}$ in Figure 5 (b), then none of the cuts of the original computation that contain a single element from this component belongs to the slice. For example, cuts that contain only e_2 but not e_3 do not belong to the slice.

From the above two observations, if the slice for $\neg p$ contains a non-trivial strongly connected component then in the computation, on every path from the initial to the final consistent cut, there exists a consistent cut that satisfies p which does not belong to the slice. Therefore, *definitely: p* holds. We can use these observations to state a sufficient condition for detecting *definitely: p*.

Theorem 6 (SC). *Given a computation $\langle E, \rightarrow \rangle$ and a regular predicate p , if $\text{slice}(\langle E, \rightarrow \rangle, \neg p)$ contains a non-trivial strongly connected component then *definitely: p* holds in $\langle E, \rightarrow \rangle$.*

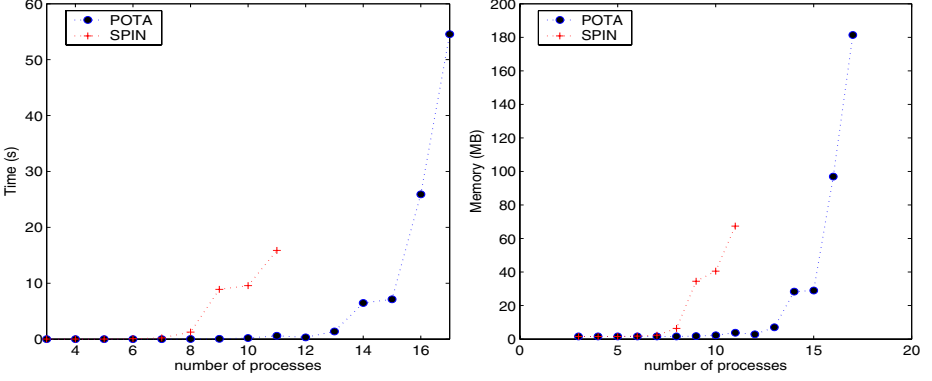
We can check this condition by finding the slice in $O(n^2|E|^2)$ time [20] and then checking the strongly connected components of the slice in $O(n|E|)$ time [20].

The converse of Theorem 6 is false. Figure 6(a) displays a computation that satisfies *definitely: p*. When we compute the union and intersection closure of the cuts that satisfy the predicate (the closure of white filled circles), we obtain the set of consistent cuts that belongs to the computation, that is, $\text{slice}(\langle E, \rightarrow \rangle, \neg p)$ has the same set of cuts as $\langle E, \rightarrow \rangle$. Therefore, the slice does not contain a non-trivial strongly connected component not in $\langle E, \rightarrow \rangle$.

Another advantage of slicing is that we can use the slice with respect to $\neg p$ instead of the computation to obtain a smaller number of linearizations for the first polynomial-space algorithm explained in Section 4.

7 Experimental Results

We implemented the conditions in this paper in POTA and applied it to a leader election protocol and the General Inter-Orb Protocol (GIOP).



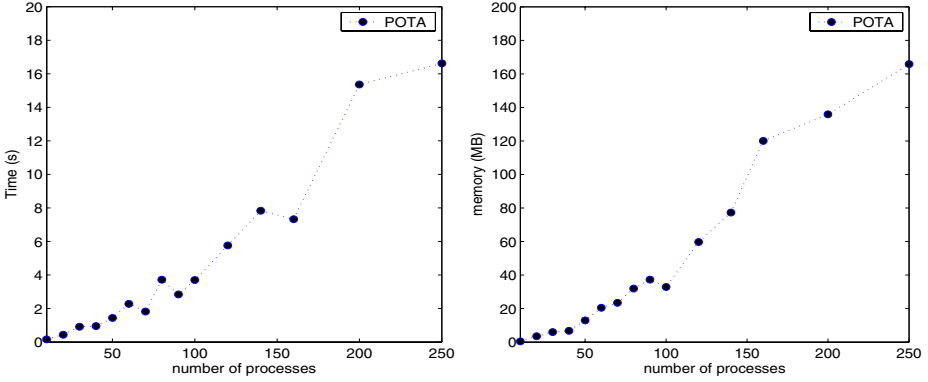


Fig. 8. GIOP verification results: SPIN runs out of memory for > 10 processes

8 Conclusion

We presented space and time efficient algorithms for testing programs with respect to *definitely: p* predicates. Earlier, we developed polynomial time detection algorithms in POTA for predicates from a subset of the temporal logic CTL that did not include definitely modality. We can enlarge this subset by the results of this paper.

References

1. B. Charron-Bost, C. Delporte-Gallet, and H. Fauconnier. Local and Temporal Predicates in Distributed Systems. *ACM Transactions on Programming Languages and Systems*, 17(1):157–179, Jan 1995.
2. C. Chase and V. K. Garg. Detection of Global Predicates: Techniques and their Limitations. *Distributed Computing*, 11(4):191–201, 1998.
3. E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, Yorktown Heights, New York, May 1981.
4. R. Cooper and K. Marzullo. Consistent Detection of Global Predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163–173, Santa Cruz, California, 1991.
5. B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 1990.
6. D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Verification*, volume 1885 of *LNCIS*, pages 323–330, 2000.
7. E. Fromentin and M. Raynal. Inevitable global states: A concept to detect unstable properties of distributed computations in an observer independent way. In *Proceedings of the 6th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 242–248, Los Alamitos, CA, 1994.
8. V. K. Garg. *Elements of Distributed Computing*. John Wiley & Sons, 2002.
9. V. K. Garg and N. Mittal. On Slicing a Distributed Computation. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 322–329, Phoenix, Arizona, April 2001.
10. V. K. Garg and B. Waldecker. Detection of Weak Unstable Predicates in Distributed Programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, March 1994.

11. H. Hallal, A. Petrenko, A. Ulrich, and S. Boroday. Using SDL Tools to Test Properties of Distributed Systems. In *Proc. FATES'01, Workshop on Formal Approaches to Testing of Software*, August 2001.
12. K. Havelund and G. Rosu. Monitoring Java Programs with Java PathExplorer. In *Runtime Verification 2001*, volume 55 of *ENTCS*, 2001.
13. G.J. Holzmann. The Model Checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.
14. M. Hurfin, M. Mizuno, M. Raynal, and M. Singhal. Efficient Distributed Detection of Conjunctions of Local Predicates in Asynchronous Computations. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 588–594, New Orleans, October 1996.
15. M. Hurfin, M. Mizuno, M. Raynal, and M. Singhal. Efficient Detection of Conjunctions of Local Predicates. *IEEE Transactions on Software Engineering*, 24(8):664–677, 1998.
16. M. Kamel and S. Leue. Formalization and Validation of the General Inter-ORB Protocol (GIOP) Using Promela and SPIN. *Software Tools for Technology Transfer*, 2(4):394–409, April 2000.
17. M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: a Run-time Assurance Tool for Java Programs. In *Runtime Verification 2001*, volume 55 of *ENTCS*, 2001.
18. L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)*, 21(7):558–565, July 1978.
19. F. Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms: Proceedings of the Workshop on Distributed Algorithms (WDAG)*, pages 215–226. Elsevier Science Publishers B. V. (North-Holland), 1989.
20. N. Mittal and V. K. Garg. Computation Slicing: Techniques and Theory. In *Proceedings of the Symposium on Distributed Computing (DISC)*, pages 78–92, Lisbon, Portugal, October 2001.
21. G. Preusse and F. Ruskey. Generating Linear Extensions Fast. *SIAM J. Comput.*, 23, 373–386, 1994.
22. R. Schwartz and F. Mattern. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing*, 7(3):149–174, 1994.
23. A. Sen and V. K. Garg. Partial Order Trace Analyzer (POTA). <http://maple.ece.utexas.edu/~sen/POTA.html>.
24. A. Sen and V. K. Garg. Detecting Temporal Logic Predicates on the Happened-Before Model. In *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS)*, Fort Lauderdale, Florida, 2002.
25. A. Sen and V. K. Garg. Automatic Generation of Computation Slices for Detecting Temporal Logic Predicates. In *Proc. of the 7th International Conference on Principles of Distributed Systems (OPODIS)*, 2003.
26. A. Sen and V. K. Garg. On Checking Whether a Predicate Definitely Holds. Technical Report TR-PDS-2003-003, PDSL, ECE Dept. Univ. of Texas at Austin, 2003.
27. A. Sen and V. K. Garg. Partial Order Trace Analyzer (POTA) for Distributed Programs. In *Runtime Verification 2003*, volume 89 of *ENTCS*, 2003.
28. A. Tarafdar and V. K. Garg. Predicate Control for Active Debugging of Distributed Programs. In *Proceedings of the 9th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 763–769, Orlando, 1998.
29. A. Tarafdar and V. K. Garg. Software Fault Tolerance of Concurrent Programs Using Controlled Re-execution. In *Proceedings of the 13th Symposium on Distributed Computing (DISC)*, pages 210–224, Bratislava, Slovak Republic, September 1999.
30. A. I. Tomlinson and V. K. Garg. Monitoring Functions on Global States of Distributed Programs. *Journal of Parallel and Distributed Computing*, 41(2):173–189, March 1997.

Using a Software Testing Technique to Improve Theorem Proving

Reiner Hähnle and Angela Wallenburg

Chalmers University of Technology and Göteborg University
Department of Computing Science
SE-412 96 Göteborg, Sweden
`{angelaw,reiner}@cs.chalmers.se`

Abstract. Most efforts to combine formal methods and software testing go in the direction of exploiting formal methods to solve testing problems, most commonly test case generation. Here we take the reverse viewpoint and show how the technique of partition testing can be used to improve a formal proof technique (induction for correctness of loops). We first compute a partition of the domain of the induction variable, based on the branch predicates in the program code of the loop we wish to prove. Based on this partition we derive a partitioned induction rule, which is (hopefully) easier to use than the standard induction rule. In particular, with an induction rule that is tailored to the program to be verified, less user interaction can be expected to be required in the proof. We demonstrate with a number of examples the practical efficiency of our method.

1 Introduction

Testing and formal verification at first glance seem to be at opposing ends in the spectrum of techniques for software quality assurance. Testing is a core technique used by practitioners every day, while formal verification is difficult to master, and employed mostly by specialists in academia. Most practitioners agree that formal verification is too cumbersome and difficult to be useful in practice. On the other hand, testing cannot be used on its own to prove the absence of errors, because exhaustive testing is usually impossible. In practice, one stops testing once the number of found errors drops below a certain threshold (or simply when the testing budget is used up). Formal verification, although costly, can ensure that a program meets its (formal) specification for any input. Given this state of affairs, it might seem surprising that testing and verification can fruitfully interact – nevertheless, this is what we want to show in the present paper.

There is one fairly established connection between formal methods and testing (documented, for example, in several papers collected in this proceedings): test case generation from formal specifications. The presence of a formal specification can also solve the oracle problem. One obstacle of this approach is that availability of a formal specification is the exception rather than the rule. On the other hand, if the cost for providing a formal specification has been invested already, one can use it as a basis not only for testing, but even for formal source code verification. The contribution of this paper is to show that techniques from testing can considerably simplify the verification effort. Hence, the availability of a formal specification is doubly useful: on the one hand, with by now

established techniques one can generate test cases automatically. In addition, as we show below, by employing techniques from testing, even formal verification may come into reach. We see our work as a first step towards a framework, where both testing and verification can be usefully combined.

Partition testing is a software testing technique used to systematically reduce test volume. A program's possibly infinite input space is divided into a finite number of disjoint subdomains. Testing is done by picking one or more elements from each subdomain to form a test set that is somehow representative for the program behaviour. Ideally, all elements in a subdomain behave in the same way with respect to the specification, that is, they are all processed correctly or they are all processed incorrectly. Subdomains with this property are called *revealing* [1] or *homogeneous* [2].

There is a line of work in software testing theory [3,4,1,5,2], where it is shown that testing *can* be used to show the *absence* of errors *provided that* certain properties in the test case selection are fulfilled. In the context of partition testing, the sought-after property is that subdomains are revealing. Unfortunately, establishing this property in practice means usually to give a formal correctness proof (for each subdomain). Hence, given the difficulties of general theorem proving, this was often discarded as impractical. Our results may be considered as a step towards obtaining such correctness proofs practically, because it suggests that proving correctness for each subdomain separately requires less user interaction than giving a proof simultaneously for the entire domain (as usually done in theorem proving).

In a nutshell, here is what we do: the implementation basis for our work is a software verification system for the programming language JAVA CARD called KeY [6]. The verification paradigm of KeY is to execute programs with symbolic values, which then are checked (symbolically) against the formal specification. More exactly, KeY is based on a first order dynamic logic with arithmetic [7,8]. It uses a sound and relatively complete calculus which contains rules *mimicking* symbolic execution. This idea was first presented in [9] and formalized in [10,11].

The main obstacle in automating software verification to an acceptable degree is the handling of programs with loops or recursive methods. These constructs require induction on one of the inductive data structures occurring in the program (for example, numbers or lists). The difficulty is to find a suitable induction hypothesis. This can be a formidable challenge even for formal methods experts. The complexity of the induction, of course, depends on the complexity of the loop or method body and post condition at hand. In simple cases, the induction can be performed automatically. Therefore, it would be extremely beneficial to simplify the required induction hypotheses. The key insight that we work out in the present paper is that the technique of partition testing is in fact a fairly general and *automatic* divide-and-conquer concept that can be used to simplify inductions in formal verification proofs.

Roughly speaking, to verify a loop, we use a white-box partition analysis based on the branch predicates of its body and condition, to compute a partition of the domain of the induction variable. This partition is then used to derive (mechanically) an induction rule which takes the partition into account: let us call the *standard induction rule* for natural numbers the rule that allows to conclude that a statement $\phi(n)$ holds for all $n \in \mathbb{N}$ provided that it holds for the single base case (" $\phi(0)$ ") and for the single step case ("for

any i , if $\phi(i)$ then $\phi(i+1)$ ”). This is replaced by an induction rule that has m base cases and r step cases, each of which matches a subdomain of the partition and, hopefully, needs much less user interaction.

Other work that is related to ours can be found elsewhere. For instance, there was an early effort [12] to use test data to aid in proving program correctness. In contrast to this approach, we do not actually run any tests, but our approach relies on a test case generation technique (partition analysis). Also, there is a recent runtime analysis technique to generate invariants inductively from test cases, presented in [13]. For higher order functional programming languages, [14] describes how to formally derive induction schema for recursively defined functions. However, our work has the advantage to be applicable to a real object-oriented programming language, JAVA CARD.

The remainder of the paper is organized as follows. We start with a motivating example in Sect. 2. In Sect. 3 we describe the method. Then we show it at work. First, we revisit the introductory example (Sect. 4.1), followed by a more sophisticated problem (Sect. 4.2). We close by pointing out current limitations (and, hence, future work).

2 Motivating Example

In this section we describe a simple example of a loop that is not possible to prove (without complex user interaction) using a standard induction rule, but is easy with our approach. The description here is brief. Our method is explained in detailed in the following section. Here is the JAVA CARD code of the loop:

```

int final c = ... ;
int i;
...

while (i > 0) {
  if (i >= c) {
    i = i - c;
  } else {
    i--;
  }
}

```

For this while-loop to terminate in a state where $i = 0$ we need in the precondition that $i \geq 0$ and $c \geq 1$. c is constant. In dynamic logic (briefly DL – the essentials of our logical framework are described in Sect. 4.1) the proof obligation is $\forall i \cdot \phi(i)$, where $\phi(i)$ is:

```

i ≥ 0 ∧ c ≥ 1 →
  ⟨ while (i > 0) {
    if (i >= c) {
      i = i - c;
    } else {
      i--;
    }
  } ⟩ i = 0

```

The formula contains a total correctness assertion: the program within the brackets $\langle \rangle$ (here the code of the while-loop) terminates and in the final state the postcondition following the brackets must hold (here $i = 0$).

The simplest possible choice for the induction hypothesis when proving correctness of the loop is to take $\phi(n)$. It is completely schematic and requires no interaction with the user. This hypothesis, however, is too weak when using the standard induction rule. Roughly speaking, in a proof attempt of the standard step case, $\forall n \in \mathbb{N} \cdot \phi(n) \rightarrow \phi(n+1)$, the following happens: the while-loop is unwound for $n+1$ and the proof branches at the if-statement. One case (the one with “ $i--;$ ”) is possible to prove, because “ $(n+1)--;$ ” is equal to n after symbolic execution. The proof obligation for this case simplifies to $\forall n \in \mathbb{N} \cdot \phi(n) \wedge n < c \rightarrow \phi(n)$, which is valid. In the other case symbolic execution gives $n+1-c$ so that the resulting proof obligation $\forall n \in \mathbb{N} \cdot \phi(n) \wedge n \geq c \rightarrow \phi(n+1-c)$ is in general unprovable. With standard induction, a more powerful induction hypothesis must be found – a difficult task for a user with no training in formal methods!

In our approach we instead create *mechanically* a new, partitioned induction rule. For our example loop the partitioned induction rule has *two* base cases and one step case:

$$\phi(0) \tag{1}$$

$$\phi(1) \wedge \dots \wedge \phi(c-1) \tag{2}$$

$$\forall n \in \mathbb{N} \cdot \phi(n) \rightarrow \phi(n+c) \tag{3}$$

These are constructed from a branch coverage partition of the induction variable i . For instance (2) above corresponds to the subdomain with all values of i causing the “`else`” branch inside the loop to be executed. The creation of the partitioned induction rule for this particular example is described in more detail in Sect. 4.1. Note that this partitioned induction rule is powerful enough to make the proof go through automatically with the unchanged induction hypothesis $\phi(n)$ that is just what we desire in our effort to minimise the user interaction.

3 Computing Partitioned Induction Rules

The idea is to create for each loop that we want to prove a new, tailor-made induction rule based on partitions. A partition is used, through the new induction rule, to divide the proof into smaller and hopefully simpler (in terms of user interaction) parts. Here is an overview of the method:

1. Compute a partition based on the branch predicates in the program code. We employ techniques readily available in the software testing community. Details are out of the scope of this paper, but the approach we use here is similar to the construction of the implementation partition in [5].
2. Refine this partition, thereby making use of the implicit case distinction contained in operators (such as `mod` or `÷`) that occur in branch predicates. The goal of the partition refinement is to arrive at subdomains of a syntactic form that is suitable for generation of the new induction rule.
3. Based on the refined partition, create a new (program-specific) induction rule with one base case for each finite subdomain of the partition and one step case for each infinite subdomain.

4. Prove correctness of the loop as usual, but use the new induction rule. This requires typically less user interaction than with the standard induction rule.

Now to the details. Specifically, assume that we have a program loop with input domain, or in our case, a domain of the variable that we want to perform induction over: $D \subseteq \mathbb{N}$. From a partition analysis as described in step 1 above we obtain a finite number of disjoint subdomains, say, $D = D_1 \cup \dots \cup D_m$. Let d_i be the characteristic predicate for each $i \in 1, \dots, m$ with $x \in D_i$ iff $d_i(x)$ holds. Hence, $x \in D$ iff $d_1(x) \vee \dots \vee d_m(x)$. The d_i are called *branch predicates*.

The branch predicates originate from the branching conditions in the program code and might contain operators defined by case distinction, for instance, $\div$, mod , and $\geq$. These implicit case distinctions drive further partitioning.

For each such operator, if necessary, we create a partition such that each case distinction in the definition of the operator gives rise to a new subdomain. In the future, we plan to create a library of partitions for all operators that occur in `JAVA CARD` expressions and the standard `JAVA CARD` API so that refining partitions can be looked up mechanically. In general, we strive to refine the original partition to obtain new subdomains of a particular *syntactic form*:

1. $\{\}$ (the empty set)
2. A finite set $\{x_1, \dots, x_k\}$. Such a set is important to distinguish because it can quite simply be used as a base case in the new induction rule.
3. An infinite set of the form $\{\lambda x.f(x) \mid x \in C\}$, where $C \subseteq \mathbb{N}$. It is important that $f(x)$ always increases its argument, because it is eventually to be used as an induction step in our new induction rule. We use $\lambda x.f(x)$ because we aim for an *expression* (and not the value of a function) to describe a set of values that we want to perform induction over.

Here is an example of a partition refinement based on an operator definition by case distinction: in the example from Sect. 2 one of the branch conditions contains the operator $\geq$, which has an implicit case distinction. We use the following definition of $\geq$:

$$x \geq z = \begin{cases} \text{true} & \text{if } \exists y \in \mathbb{N} \cdot (x = z + y) \\ \text{false} & \text{if } \exists y \in \mathbb{N} \cdot (x = z - 1 - y) \end{cases}$$

Each case gives directly a simple expression of the desired form: $\lambda n.(c+n)$, respectively, $\lambda n.(c-1-n)$ would be used to refine a subdomain defined by the predicate $i \geq c$.

More precisely, we refine the subdomain of the original partition ($\{i \in \mathbb{N} \mid i \geq c\}$) into two new subdomains by replacing i with $(c+n)$ and $(c-1-n)$ respectively. We then get $\{c-1-n \mid n \in \mathbb{N} \wedge c-1-n \geq c\} = \{\}$ and $\{c+n \mid n \in \mathbb{N} \wedge c+n \geq c\} = \{c+n \mid n \in \mathbb{N}\}$, which are both of the form required above. The latter can be used to derive an induction step case.

Assume now that we have a refined partition of the syntactic form detailed above, where operators with implicit case distinctions are eliminated.

We create a new induction rule with the following set of proof obligations:

1. For each non-empty finite subdomain $\{x_1, \dots, x_k\}$, we create a base case consisting of the proof obligation

$$\phi(x_1) \wedge \dots \wedge \phi(x_k)$$

2. For each infinite subdomain D_i a new step case needs to be proven:

$$\forall n \in C_i \cdot \phi(n) \rightarrow \phi((\lambda x. f_i(x))n)$$

For the new induction rule to be sound it is important that some criteria are fulfilled:

1. For each step case of the form $\forall n \in C_i \cdot \phi(n) \rightarrow \phi((\lambda x. f_i(x))n)$ the following holds:

$$\forall n \in C_i \cdot f_i(n) > n$$

This is to ensure that it really is a step, and it is achieved by constructing $f_i(x)$ such that it increases its argument.

2. Each element of the domain D of the induction variable is covered in at least one of the step or base cases. Let B be the union of all finite subdomains giving rise to a base case and let $f_1, \dots, f_r$ be the functions that define the step cases. Then we require

$$\forall x \in D \cdot (\exists k \in \{1, \dots, r\} \cdot \exists y \in C_k \cdot x = f_k(y)) \vee x \in B$$

This property is guaranteed by construction, because the partition property is invariant in the process; we only refine partitions or do not change them at all.

The first property entails that the minimal element of D cannot be in the subdomain defined by any step case. The second property says that all elements of D is in either a step case or a base case. As a consequence, there must be at least one base case of the induction.

4 Examples

4.1 Simple Example Revisited

Now we return to the motivating example from Sect. 2 and show how we actually computed the partitioned induction rule for it.

In KeY, the logical infrastructure is JAVA CARD DL [8], an extension of dynamic logic (DL) [7] to handle side effects, aliasing, exceptions and other complications of a real object-oriented programming language such as JAVA CARD. In DL, a formula $\varphi \rightarrow \langle p \rangle \psi$ is valid if for every state s satisfying precondition φ a run of the program p starting in s terminates, and in the terminating state the post-condition ψ holds. The proof obligation from Sect. 2, that was written using pure DL, is slightly more complicated in JAVA CARD DL. Our proof obligation is $\forall i_l \cdot \phi(i_l)$. Let $\phi(i_l)$ be the following formula:

$$\begin{aligned} i_l \geq 0 \wedge c_l \geq 1 \rightarrow \\ \{i := i_l\} \{c := c_l\} \{ \textbf{while } (i > 0) \{ \\ \quad \textbf{if } (i \geq c) \{ \\ \quad \quad i = i - c; \\ \quad \} \textbf{else } \{ \\ \quad \quad i--; \\ \quad \} \\ \} \} i = 0 \end{aligned}$$

The curly brackets in front of the formula are called (state) *updates*. Updates are the JAVA CARD DL solution to deal with aliasing and assignment in the calculus. They are basically primitive assignments of the form $\{loc := val\}$ where val must be a logical (side effect free) term and loc a program variable.

To prove correctness of the loop we need to perform induction on the variable i . In JAVA CARD DL one cannot quantify over program variables, so the induction variable is a corresponding logical variable i_l . The domain of the induction variable is $\mathbb{N}$. From the branching conditions in the program we obtain the first partition of i 's domain:

$$\begin{aligned} D_1 &= \{x \in \mathbb{N} \mid d_1(x)\} = \{x \in \mathbb{N} \mid x \leq 0\} = \\ &= \{0\} \\ D_2 &= \{x \in \mathbb{N} \mid d_2(x)\} = \{x \in \mathbb{N} \mid x > 0 \wedge x < c\} = \\ &= \{1, \dots, c-1\} \\ D_3 &= \{x \in \mathbb{N} \mid d_3(x)\} = \{x \in \mathbb{N} \mid x > 0 \wedge x \geq c\} = \\ &= \{x \in \mathbb{N} \mid x \geq c\} \end{aligned}$$

The subdomains $D_1 = \{0\}$ and $D_2 = \{1, \dots, c-1\}$ are finite and thus already in one of our desired formats.

Then, to refine/rewrite the original subdomain D_3 , remember from Sect. 3 that for the operator $x \geq z$, we may use the expressions $\lambda y.z + y$ and $\lambda y.z - 1 - y$ to refine a subdomain. This gives a refinement of $D_3 = \{x \in \mathbb{N} \mid x \geq c\}$ into two new subdomains $D_3 = D_{31} \cup D_{32}$, where

$$\begin{aligned} D_{31} &= (\text{replace } x \text{ in } D_3 \text{ with } c + y) \\ &= \{c + y \mid y \in \mathbb{N} \wedge c + y \geq c\} \\ &= \{c + y \mid y \in \mathbb{N}\} \\ D_{32} &= (\text{replace } x \text{ in } D_3 \text{ with } c - 1 - y) \\ &= \{c - 1 - y \mid y \in \mathbb{N} \wedge c - 1 - y \geq c\} \\ &= \{\} \end{aligned}$$

So the new subdomains D_1 , D_2 and D_{31} are of the form we need to construct the new induction rule. To prove $\forall n \in \mathbb{N} \cdot \phi(n)$, it is then enough to prove

$$\phi(0) \tag{1}$$

$$\phi(1) \wedge \dots \wedge \phi(c-1) \tag{2}$$

$$\forall n \in \mathbb{N} \cdot \phi(n) \rightarrow \phi(c+n) \tag{3}$$

where (1) is a base case and covers D_1 , (2) is also a base case and covers D_2 , and (3) is a step case that covers all elements in the subdomain D_{31} .

The proving process in KeY is partially automated, though it is an interactive theorem prover. When using the partitioned induction rule above, the following kinds of user interaction are required to complete the proof:

Instantiation means a single quantifier elimination by supplying a suitable instance term. In the KeY system, the user can simply drag-and-drop the desired term.

Induction rule application: when applying the partitioned induction rule, one can state the induction hypothesis by drag-and-dropping the existing proof obligation, and then pick the induction variable.

Unwinding of the loop needs to be initiated, but is done automatically.

Decision procedure is an automatic procedure that tries to decide the validity of arithmetic expressions over the integers. The decision procedure is sound but not complete. The user decides when (if) to run it.

Compared to the user interaction needed when using standard induction, this is less complicated. Using standard induction, if one uses the unmodified induction hypothesis and the same induction variable as above, one is left with an open proof goal and no rules to apply: one has to figure out a strong enough induction hypothesis.

4.2 Russian Multiplication Example

Let us see how the method works for proving the correctness of a more complicated algorithm – russian multiplication. The loop has more complicated control flow than in the previous example.

```

int  russianMultiplication  (int a, int b) {
    int z = 0;
    while (a != 0) {
        if (a mod 2 != 0) {
            z = z + b;
        }
        a = a/2;
        b = b*2;
    }
    return z;
}

```

For this loop we have the precondition $a_0 \geq 0$ and the post-condition $z = z_0 + a_0 * b_0$, where a_0, b_0, z_0 are the values of a, b and z before the loop. In JAVA CARD DL the proof obligation for the total correctness of this loop is $\forall a_0 \cdot \phi(a_0)$, where $\phi(a_0)$ is

$$\begin{aligned}
 &\forall b_0 \cdot \forall z_0 \cdot a_0 \geq 0 \rightarrow \\
 &\{a := a_0\} \{b := b_0\} \{z := z_0\} \{ \text{while } (a \neq 0) \{ \\
 &\quad \text{if } (a \bmod 2 \neq 0) \{ \\
 &\quad \quad z = z + b; \\
 &\quad \} \\
 &\quad a = a / 2; \\
 &\quad b = b * 2; \\
 &\} \} z = z_0 + a_0 * b_0
 \end{aligned}$$

where a_0, b_0 and z_0 are new logical variables.

This cannot be proven using standard induction unless the induction hypothesis is strengthened in a non-trivial way. In an attempt to prove the standard step case using $\phi(a)$ as the induction hypothesis in $\forall a \in \mathbb{N} \cdot \phi(a) \rightarrow \phi(a+1)$, after unwinding and symbolically executing the loop we end up with $\forall a \in \mathbb{N} \cdot \phi(a) \rightarrow \phi((a+1)/2)$ which is unprovable without induction.

Now let us compute a partitioned induction rule for this loop instead. The induction variable is a (the corresponding logical variable is a_0). Its domain is $\mathbb{N}$ and the first partitioning, using branch predicates, gives the following subdomains:

$$\begin{aligned} D_1 &= \{x \in \mathbb{N} \mid d_1(x)\} = \{x \in \mathbb{N} \mid x = 0\} \\ &= \{0\} \\ D_2 &= \{x \in \mathbb{N} \mid d_2(x)\} \\ &= \{x \in \mathbb{N} \mid x \neq 0 \wedge x \bmod 2 \neq 0\} \\ D_3 &= \{x \in \mathbb{N} \mid d_3(x)\} \\ &= \{x \in \mathbb{N} \mid x \neq 0 \wedge x \bmod 2 = 0\} \end{aligned}$$

In words, we have the singleton set containing zero and the sets with the odd and (non-zero) even numbers respectively.

Consider the branch predicate $d_3(x) \leftrightarrow x \neq 0 \wedge x \bmod 2 = 0$, which defines subdomain D_3 . The definition of $d_3(x)$ contains an operator with implicit case distinction: $\bmod$. We look up the definition of $\bmod 2$:

$$x \bmod 2 = \begin{cases} 0 & \text{if } \exists y \in \mathbb{N} \cdot (x = 2 * y) \\ 1 & \text{if } \exists y \in \mathbb{N} \cdot (x = 2 * y + 1) \end{cases}$$

Hence, we use the expressions $\lambda y. 2 * y$ and $\lambda y. 2 * y + 1$ to refine the original partition. Using the case distinction in the definition of $x \bmod 2$, gives us the refinement of the original subdomain $D_3 = \{x \in \mathbb{N} \mid x \neq 0 \wedge x \bmod 2 = 0\}$ into two new subdomains $D_3 = D_{31} \cup D_{32}$:

$$\begin{aligned} D_{31} &= (\text{replace } x \text{ in } D_3 \text{ with } 2 * y) \\ &= \{2 * y \mid y \in \mathbb{N} \wedge (2 * y) \neq 0 \wedge (2 * y) \bmod 2 = 0\} \\ &= \{2 * y \mid y \in \mathbb{N} \wedge y \neq 0 \wedge 0 = 0\} \\ &= \{2 * y \mid y \in \mathbb{N}_1\} \\ D_{32} &= (\text{replace } x \text{ in } D_3 \text{ with } 2 * y + 1) \\ &= \{2 * y + 1 \mid y \in \mathbb{N} \wedge 2 * y + 1 \neq 0 \wedge (2 * y + 1) \bmod 2 = 0\} \\ &= \{2 * y + 1 \mid y \in \mathbb{N} \wedge 1 = 0\} \\ &= \{\} \end{aligned}$$

Similarly, for the branch predicate $d_2(x)$ of the original partition, we get

$$\begin{aligned} D_{21} &= \{2 * y + 1 \mid y \in \mathbb{N}\} \\ D_{22} &= \{\} \end{aligned}$$

After refinement, we have non-empty subdomains of the form:

$$\begin{aligned}
D_1 &= \{0\} \\
D_{21} &= \{2 * y + 1 \mid y \in \mathbb{N}\} \\
D_{31} &= \{2 * y \mid y \in \mathbb{N}_1\}
\end{aligned}$$

Thus, the new subdomains have the syntactic form we need to construct an induction rule. With this rule, to prove $\forall n \in \mathbb{N} \cdot \phi(n)$, it is enough to prove that

$$\phi(0) \tag{1}$$

$$\forall n \in \mathbb{N}_1 \cdot \phi(n) \rightarrow \phi(2 * n) \tag{2}$$

$$\forall n \in \mathbb{N} \cdot \phi(n) \rightarrow \phi(2 * n + 1) \tag{3}$$

where (1) is the base case, covering D_1 , (2) the first step case, covering all elements in the subdomain D_{31} and (3) the second step case, covering D_{21} .

To prove the russian multiplication algorithm in KeY with the partitioned induction rule the required user interactions are basically the same as in the previous example. In particular, the induction now goes through completely unmodified.

5 Limitations and Future Work

In this paper we demonstrated that the technique of partition testing can be turned into a divide-and-conquer concept to simplify inductions in formal verification proofs. We have defined a syntactic framework that allows us to derive tailor-made induction rules based on partitions in a practically efficient manner. Resulting induction rules are sound and complete by construction. The actual verification in KeY using the partitioned induction rules can often be performed automatically. Several examples were carried out not just by hand, but as concrete experiments in an interactive theorem prover. The experimental findings confirmed our conjecture. We think that our work is a first step towards a framework, where both testing and formal verification can be usefully combined.

In the current setting, our method has a number of limitations but its reach could be extended considerably. For a start, we considered induction not over arbitrary inductive data structures, but only the natural numbers. Future work is to extend our approach to also include induction over lists, trees, etc.

Our focus has been entirely on the verification of loops, and not on arbitrary programs. Since loops are usually the major source of complexity in verification, in testing as well as in theorem proving, it is here that we expect the largest gain. Still, we also wish to investigate the idea of partitioning proofs for loop-free programs, since it has been seen [15] that in the case of very large proof obligations, it is beneficial to split the proof into parts which can be handled separately.

Clearly, not all induction proofs can be simplified with our approach. The crucial point is that our method requires that the branch predicates somehow capture what is being computed in the corresponding branch. This is often the case, but not always. If the branch predicates are completely unrelated to the induction variable, we simply get no information from the branch predicates on how to partition the domain of the induction variable. For instance in array-sorting algorithms, it is common that the induction goes over the indexes of the array, but the branch predicates typically have a comparison

between the elements to be sorted and these might be randomly ordered. In future work we plan to remedy this by also using the weakest preconditions of updates to induction variables when we refine the partition. In KeY there is already a strongest postcondition generator.

Finally, the process of transforming general branch predicates into predicates of the form that our method requires (using λ -expressions) is non-trivial; in particular for the process to be mechanised. In our examples, quite simple branch conditions occurred. It is future work to investigate what exactly can be done mechanically. It includes dealing with predicates containing arbitrary linear operators and method calls, but quadratic operators and operators like $\sin$, we expect to be beyond reach. However, our method is conservative in the sense that if it does not find a useful refinement of a partition for a certain subdomain, the subdomain stays the same. In that case the proof will not be simplified, but it will not be more complicated either.

Acknowledgements

We thank the anonymous reviewers and the editors for their many helpful comments. This work was also supported by a STINT (the Swedish Foundation for International Cooperation in Research and Higher Education) grant.

References

1. Weyuker, E.J., Ostrand, T.J.: Theories of Program Testing and the Application of Revealing Subdomains. *IEEE Transactions on Software Engineering* **6** (1980) 236–246
2. Hamlet, D., Taylor, R.: Partition Testing Does Not Inspire Confidence. *IEEE Transactions on Software Engineering* **16** (1990) 1402–1411
3. Goodenough, J.B., Gerhart, S.L.: Toward a Theory of Test Data Selection. *IEEE Transactions on Software Engineering* **1** (1975) 156–173
4. Howden, W.E.: Reliability of the Path Analysis Testing Strategy. *IEEE Transactions on Software Engineering* **2** (1976) 208–215
5. Richardson, D., Clarke, L.: Partition Analysis: A Method Combining Testing and Verification. *IEEE Transactions on Software Engineering* **11** (1985) 1477–1490
6. Ahrendt, W., Baar, T., Beckert, B., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Schmitt, P.H.: The KeY system: Integrating object-oriented design and formal methods. In Kutsche, R.D., Weber, H., eds.: *Fundamental Approaches to Software Engineering*. Volume 2306 of LNCS., Springer-Verlag (2002) 327–330
7. Harel, D., Kozen, D., Tiuryn, J.: *Dynamic Logic*. MIT Press (2000)
8. Beckert, B.: A dynamic logic for the formal verification of Java Card programs. In Attali, I., Jensen, T., eds.: *Java on Smart Cards: Programming and Security*. Revised Papers, Java Card 2000, Cannes, France. Volume 2041 of LNCS., Springer-Verlag (2001) 6–24
9. Burstall, R.M.: Program proving as hand simulation with a little induction. In Rosenfeld, J.L., ed.: *Information Processing 74, Proceedings of IFIP Congress 74*. (1974) 201–210
10. Hähnle, R., Heisel, M., Reif, W., Stephan, W.: An interactive verification system based on dynamic logic. In Siekmann, J., ed.: *Proc. 8th Conference on Automated Deduction CADE*, Oxford. Volume 230 of LNCS., Springer-Verlag (1986) 306–315
11. Heisel, M., Reif, W., Stephan, W.: Program verification by symbolic execution and induction. In: *Proc. 11th German Workshop on Artificial Intelligence*. Volume 152 of *Informatik Fachberichte.*, Springer-Verlag (1987) 201–210

12. Geller, M.M.: Test data as an aid in proving program correctness. *Communications of the ACM* **21** (1978) 368–375
13. Nimmer, J.W., Ernst, M.D.: Automatic generation of program specifications. In: *Proceedings of the international symposium on Software testing and analysis*, ACM Press (2002) 229–239
14. Slind, K.: Derivation and use of induction schemes in higher-order logic. In Gunter, E.L., Felty, A., eds.: *Proc. 10th International Theorem Proving in Higher Order Logics Conference*. Volume 1275 of LNCS., Springer-Verlag (1997) 275–290
15. Claessen, K., Hähnle, R., Mårtensson, J.: Verification of hardware systems with first-order logic. In Sutcliffe, G., Pelletier, J., Suttner, C., eds.: *Proc. Problems and Problem Sets Workshop*, affiliated to CADE-18, Copenhagen, DIKU, University of Copenhagen, Denmark (2002) Technical Report.

Auto-generating Test Sequences Using Model Checkers: A Case Study^{*}

Mats P.E. Heimdahl¹, Sanjai Rayadurgam¹, Willem Visser²,
George Devaraj¹, and Jimin Gao¹

¹ Department of Computer Science and Engineering
University of Minnesota, Minneapolis, MN 55455
{heimdahl,rsanjai,devaraj,jgao}@cs.umn.edu

² NASA Ames Research Center
Moffett Field, California 94035
{wvisser@nasa.arc.gov}

Abstract. Use of model-checking approaches for test generation from requirement models have been proposed by several researchers. These approaches leverage the witness (or counter-example) generation capability of model-checkers for constructing test cases. Test criteria are expressed as temporal properties. Witness traces generated for these properties are instantiated to create complete test sequences, satisfying the criteria. State-space explosion can, however, adversely impact model-checking and hence such test generation. Thus, there is a need to validate these approaches against realistic industrial sized system models to learn how well these approaches scale. To this end, we conducted a case study using six models of progressively increasing complexity of the mode-logic in a flight-guidance system, written in the RSML^{-e} language. We developed a framework for specification-based test generation using the NuSMV model-checker and code based test case generation using Java Pathfinder, and collected time and resource usage data for generating test cases using symbolic, bounded, and explicit state model-checking algorithms. This paper briefly discusses the approach, presents the results from the study and analyzes its implications.

1 Introduction

Software development for high assurance systems, such as the software controlling aeronautics applications and medical devices, is a costly and time consuming process. In such projects, the validation and verification phase (V&V) consume approximately 50%–70% of the software development resources. Thus, automatic generation of test cases from requirement specifications has found considerable interest in the research community. Such automation could result in dramatic time and cost savings, especially for verifying safety-critical systems.

^{*} This work has been partially supported by NASA grant NAG-1-224 and NASA contract NCC-01-001. We also want to thank the McKnight Foundation for their generous support over the years.

Model checking techniques have been proposed as one method of achieving this automation [3,9,2,10,20,16]. These proposed test case generation approaches leverage the witness (or counter-example) generation capability of model-checkers for constructing test cases. Test criteria are expressed as temporal properties. Witness traces generated for these properties are instantiated to create complete test sequences, satisfying the criteria. Nevertheless, one of the issues that often stymies model-checking is the state-space explosion problem. As the size of the state-space to be explored increases, model-checking might become too time-consuming or infeasible. But in the context of test generation based on structural properties, one is interested in falsifying properties so that counter-examples can be instantiated to test sequences. We have hypothesized that finding violations of the properties characterizing a test case is easy and that the counter-examples can be constructed easily even for large models.

While these ideas are appealing there is a need to validate the approach using realistic models of critical systems. To this end, we conducted a case study using six models of progressively increasing complexity of the mode-logic in a flight-guidance system, written in the RSML^{-e} language [22,23]. We developed a framework for specification-based test generation using the NuSMV [19] model-checker and code based test case generation using Java Pathfinder [25] and collected time and resource usage data for generating test cases using symbolic, bounded, and explicit state model-checking algorithms. The purpose of this study was to determine if a model-checking based approach to test generation could scale to software system models of industrial size and complexity.

To summarize our findings, our case study points out limitations of symbolic as well as explicit state model checkers when used for test case generation. A bounded model checker, however, performed very well in our application domain and shows great promise.

The rest of the paper is organized as follows. Section 2 provides a short overview of related efforts in the area of test-generation using model checking techniques and briefly describes our overall approach. We describe how we conducted our case study in Section 3, and present the FGS case example in Section 4. Sections 5 and 6 briefly discuss RSML^{-e} and the test coverage criteria used for this study. Section 7 analyzes the results obtained from our experiments with the RSML^{-e} specification language. In Section 8 we cover our experiences with an explicit state model checker applied to Java code. Finally, Section 9 discusses the implications of the results and points to future studies and experiments that are further required to validate model-checking based test generation approaches.

2 Finding Tests with a Model Checker

Model checkers build a finite state transition system and exhaustively explore the reachable state space searching for violations of the properties under investigation [7]. Should a property violation be detected, the model checker will produce a counter-example illustrating how this violation can take place. In

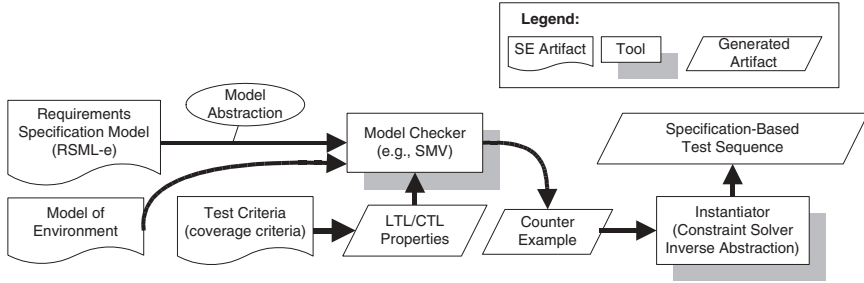


Fig. 1. Test sequence generation overview and architecture

short, a counter-example is a sequence of inputs that will take the finite state model from its initial state to a state where the violation occurs.

A model checker can be used to find test cases by formulating a test criterion as a verification condition for the model checker. For example, we may want to test a transition (guarded with condition C) between states A and B in the formal model. We can formulate a condition describing a test case testing this transition – the sequence of inputs must take the model to state A ; in state A , C must be true, and the next state must be B . This is a property expressible in the logics used in common model checkers, for example, the logic LTL. We can now challenge the model checker to find a way of getting to such a state by negating the property (saying that we assert that there is no such input sequence) and start verification. The model checker will now search for a counterexample demonstrating that this property is, in fact, satisfiable; such a counterexample constitutes a test case that will exercise the transition of interest. By repeating this process for each transition in the formal model, we use the model checker to automatically derive test sequences that will give us transition coverage of the model. The proposed test generation process is outlined in Figure 1. Naturally, the same thinking can be applied to the generation of test cases from source code, for example, from Java as we will illustrate later in the paper.

Several research groups are actively pursuing model checking techniques as a means for test case generation.

Gargantini and Heitmeyer [10] describe a method for generating test sequences from requirements specified in the SCR notation. To derive a test sequence, a *trap property* is defined which violates some known property of the specification. In their work, they define trap properties that exercise each case in the event and condition tables available in SCR – this provides a notion of branch coverage of an SCR specification.

Ammann and Black [2,1] combine mutation analysis with model-checking based test case generation. They define a specification based coverage metric for test suites using the ratio of the number of mutants killed by the test suite to the total number of mutants. Their test generation approach uses a model-checker to generate mutation adequate test suites. The mutants are produced by systematically applying mutation operators to both the properties specifications

and the operational specification, producing respectively, both positive test cases which a correct implementation should pass, and negative test cases which a correct implementation should fail.

Rayadurgam, *et al.* in [20] provide a formalism suitable for structural test-case generation using model checkers and in [21] illustrate how this approach can be applied to a formal specification language. They also presented a framework for specification centered testing in [13].

Lee, *et al.* [16] formulate a theoretical framework for using temporal logic to specify data flow test coverage criteria. They also discuss various techniques for reducing the size of the test set generated by the model checker [15]. The underlying argument in all these works, as in our own earlier work, is that when test criteria can be appropriately formulated as temporal logic formulas, one could use model-checking to produce witnesses for those formulas, which could then be seen as test sequences satisfying the coverage criteria.

However, to our knowledge, not much experimental data is available about the efficiency of model-checking based test-generation for realistic systems. Our goal is to conduct a series of studies using realistic systems, apply the techniques and examine how well these techniques perform and to what extent they scale up.

3 Case Study Overview

In our case study, we were interested in answering four questions:

1. If we naively generate one test case for each structure we want to cover, how many test cases will be generated for various coverage criteria?
2. Does test case generation using symbolic and bounded model checking scale to realistic systems?
3. Where do the test case generation capabilities of symbolic and bounded model checking break down?
4. Can a code model checker, such as JPF, be used to find test cases based on realistic code?

To answer these questions, we devised a rigorous case study evaluating the test case generation capabilities of model checkers. We have developed a test case generation engine integrated in our NIMBUS toolset for the development of RSML^{-e} specifications [22] (described in Section 5). This test case generator allows us to generate test cases to various structural coverage criteria using the NuSMV model checker. The coverage criteria we have used in this study are discussed in Section 6.

To stress the capabilities of NuSMV, we wanted to work with models with realistic structure as well as realistic size. In a related project, Rockwell Collins Inc., in collaboration with the University of Minnesota, have developed a collection of progressively more complex RSML^{-e} models of the mode logic of a flight guidance system (FGS). The models range from a very simple “toy-version” of the FGS (FGS00) to a close to production version of the logic (FGS05). The case example is discussed in some detail in Section 4.

Table 1. Data on the size of the RSML^{-e} and SMV FGS models

	FGS00		FGS01		FGS02		FGS03		FGS04		FGS05	
	RSML	SMV	RSML	SMV	RSML	SMV	RSML	SMV	RSML	SMV	RSML	SMV
#lines of code	287	423	455	639	774	1045	1034	1186	1781	2052	2564	2902
#vars/BDD's	19	109	27	167	43	281	57	353	90	615	142	849

We performed the case study by conducting the following steps:

1. Use NIMBUS to automatically generate the LTL properties (trap properties) characterizing the test cases needed to satisfy various coverage criteria for FGS00 through FGS05.
2. Use the symbolic as well as bounded model checkers provided in NuSMV to generate counterexamples for the suites of trap properties.
3. Automatically process the counterexamples to provide test cases suitable for use in a test automation environment.

During the case study, we collected information on (1) how many test cases were generated, (2) run time and memory usage of the model checkers, and (3) the average length of the test cases generated.

To complete the case study, we investigated the feasibility of using a code model checker to complete the test suites derived from the formal specification. This capability would be used should the specification-based tests not provide adequate coverage of the implementation. To this end, we derived Java code from the formal specifications, executed a test suite generated from the specification, identified branches that were not covered, and derived tests for these branches using Java Pathfinder Java model checker [25].

In the remainder of this paper we provide a detailed description of the artifacts and activities involved in the case study.

4 Flight Guidance System

A Flight Guidance System (FGS) is a component of the overall Flight Control System (FCS) in a commercial aircraft. It compares the measured state of an aircraft (position, speed, and altitude) to the desired state and generate pitch and roll guidance commands to minimize the difference between the measured and desired state¹. The FGS can be broken down to mode logic, which determines which lateral and vertical modes of operation are active and armed at any given time, and the flight control laws that accept information about the aircraft's current and desired state and compute the pitch and roll guidance commands. In this case study we have used the mode logic.

Figure 2 illustrates a graphical view of a FGS in the NIMBUS environment. The primary modes of interest in the FGS are the horizontal and vertical modes. The horizontal modes control the behavior of the aircraft about the longitudinal, or roll, axis, while the vertical modes control the behavior of the aircraft about

¹ We thank Dr. Steve Miller and Dr. Alan Tribble of Rockwell Collins Inc. for the information on flight control systems and for letting us use the RSML^{-e} models they have developed using NIMBUS.

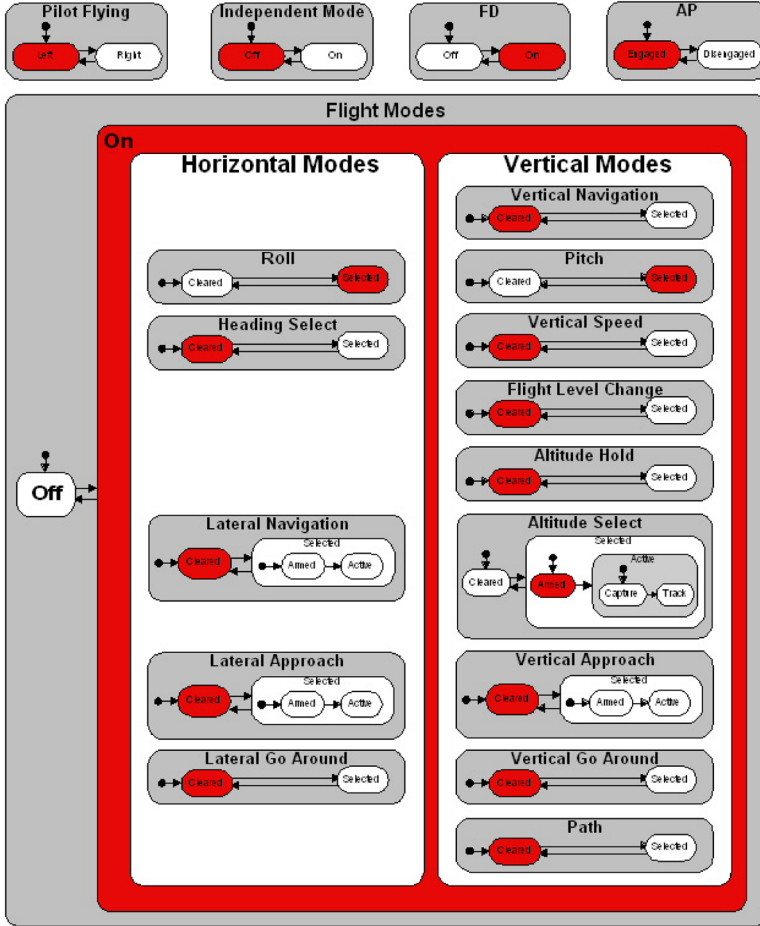


Fig. 2. Flight Guidance System

the vertical, or pitch, axis. In addition, there are a number of auxiliary modes, such as half-bank mode, that control other aspects of the aircraft's behavior.

The FGS is ideally suited for test case generation using model checkers since it is discrete – the mode logic consists entirely of enumerated and Boolean variables. As mentioned earlier, we used six models that are of progressively increasing complexity. An indication of the model size can be found in Table 1. The measure for the RSML^{-e} models refer to the number of state variables in the RSML^{-e} model and measure for the SMV models refers to the number of BDD variables needed to encode the model.

5 NIMBUS and RSML^{-e}

Figure 3 shows an overview of the NIMBUS tools framework we have used as a basis for our test case generation engine. The user builds a behavioral model of

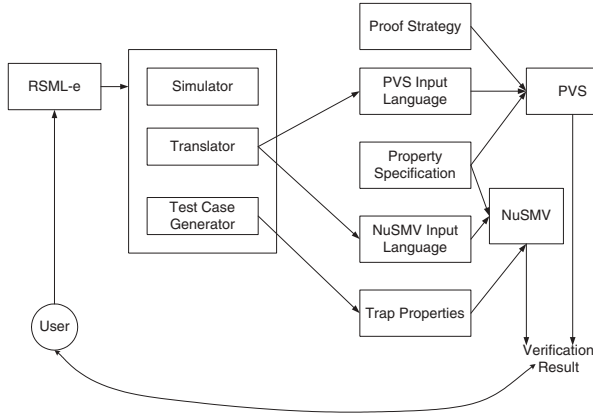


Fig. 3. Verification Framework

the system in the fully formal and executable specification language RSML^{-e} (see below). After evaluating the functionality and behavioral correctness of the specification using the NIMBUS simulator, users can translate the specifications to the PVS or NuSMV input languages for verification (or test case generation as is the case in this report). The set of LTL trap properties required to use NuSMV to generate test sequences are obtained by traversing the abstract syntax tree in NIMBUS and then outputting sets of properties whose counterexamples will provide the correct coverage (the coverage criteria an associated properties are discussed in the next section).

To generate test cases in NIMBUS, the user would invoke the following steps:

Model creation and trap property generation: The formal model in NuSMV can be generated automatically from the RSML^{-e} specification from the NIMBUS command line. The test criterion is specified as a command line argument when building the NuSMV model. The result of this command is an SMV model of the system and a collection of trap properties whose counterexamples will provide the desired coverage.

Counterexample generation using NuSMV: The model and the trap properties are merged and given to the NuSMV tool. A Unix script invokes the NuSMV tool in interactive mode, reads the model, flattens the hierarchy, encodes the variables, and checks the specifications for the trap properties. After completing the script, we have collected the counterexample traces for all trap properties in a text file.

Concrete test case generation from NuSMV Output: For any counterexample, the trace information from NuSMV contains only delta changes in each subsequent state following the initial state. Therefore, to generate test sequences, we need to remember the value of the variables in the initial state configuration so that we can construct usable test cases by applying the delta changes to the initial configuration. The processing of the counterexamples and generation of an intermediate test representation is currently

achieved with a simple piece of software implemented in C. The intermediate test representation contains (1) the input in each step, (2) the expected state changes (to state variables internal to the RSML^{-e} model), and (3) the expected outputs (if any).

The NIMBUS tools discussed above all operate on the RSML^{-e} notation – RSML^{-e} is based on the Statecharts [12] like language Requirements State Machine Language (RSML) [18]. RSML^{-e} is a fully formal and synchronous data-flow language without any internal broadcast events (the absence of events is indicated by the $^{-e}$).

An RSML^{-e} specification consists of a collection of input variables, state variables, input/output interfaces, functions, macros, and constants; *input variables* are used to record the values observed in the environment, *state variables* are organized in a hierarchical fashion and are used to model various states of the control model, *interfaces* act as communication gateways to the external environment, and *functions and macros* encapsulate computations providing increased readability and ease of use.

Figure 4 shows a specification fragment of an RSML^{-e} specification of the Flight Guidance System². The figure shows the definition of a state variable, ROLL. ROLL is the default lateral mode in the FGS mode logic.

The conditions under which the state variable changes value are defined in the **TRANSITION** clauses in the definition. The condition tables are encoded in the macros, **Select_ROLL** and **Deselect_ROLL**. The tables are adopted from the original RSML notation – each column of truth values represents a conjunction of the propositions in the leftmost column (F represents the negation of the proposition and a ‘*’ represents a “don’t care” condition). If a table contains several columns, we take the disjunction of the columns; thus, the table is a way of expressing conditions in a disjunctive normal form.

6 Coverage Criteria

For the case study described in this report, we have selected to use three representative specification coverage criteria; state coverage, decision coverage (in the RSML^{-e} context called table coverage), and a version of MC/DC coverage [4] called clause-wise condition coverage.

In the following discussion, a *test case* is to be understood as a sequence of values for the input variables in an RSML^{-e} specification. This sequence of inputs will guide the RSML^{-e} specification from its initial state to the structural element, for example, a transition, the test cases was designed to cover. A *test suite* is simply a set of such test cases. As we briefly explained, trap properties are used to generate counter-examples using a model checker. These properties are derived from the structural coverage criteria. For the purposes of illustration, we use the FGS example discussed in Section 4.

² We use here the ASCII version of RSML^{-e} since it is much more compact than the more readable typeset version.

```

STATE_VARIABLE ROLL : Base_State
    PARENT          : Modes.On
    INITIAL_VALUE    : UNDEFINED
    CLASSIFICATION   : State

    TRANSITION UNDEFINED TO Cleared IF NOT Select_ROLL()
    TRANSITION UNDEFINED TO Selected IF Select_ROLL()
    TRANSITION Cleared TO Selected IF Select_ROLL()
    TRANSITION Selected TO Cleared IF Deselect_ROLL()

END STATE_VARIABLE

MACRO Select_ROLL() :
    TABLE
        Is_No_Nonbasic_Lateral_Mode_Active()      : T;
        Modes = On                                  : T;
    END TABLE
END MACRO

MACRO Deselect_ROLL() :
    TABLE
        When_Nonbasic_Lateral_Mode_Activated()     : T *;
        When(Modes = Off)                          : * T;
    END TABLE
END MACRO

```

Fig. 4. A small portion of the FGS specification in $RSML^{-e}$

6.1 State Coverage

Definition 1. *A test suite is said to achieve state coverage of a state variable in an $RSML^{-e}$ specification, if for each possible value of the state variable there is at least one test case in the test suite that assigns that value to the given variable. The test suite achieves state coverage of the specification if it achieves state coverage for each state variable.*

Consider, for example, the state variable `ROLL` in the FGS specification example:

```
STATE_VARIABLE ROLL : { Cleared, Selected, UNDEFINED };
```

A test suite would achieve state coverage on `ROLL`, if for each of its three different possible values, there is a test case in which `ROLL` takes that value. Note that a single test case might actually achieve this coverage by assigning different values to `ROLL` at different points in the sequence. To provide a comprehensive test suite, however, in this case study we generate one test case for each state variable value. One could use the following LTL formulas to generate the test cases:

1. $G \sim(\text{ROLL} = \text{Cleared})$
2. $G \sim(\text{ROLL} = \text{Selected})$
3. $G \sim(\text{ROLL} = \text{UNDEFINED})$

In each case, the property asserts that `ROLL` can never have a specific value and the counter-example produced is a sequence of values for the system variables starting from an initial state and ending in a state where `ROLL` has the specific value.

6.2 Decision Coverage (Table Coverage)

Definition 2. *A test suite is said to achieve decision coverage of a given RSML^{-e} specification, if each guard condition (specified as either an AND/OR table or as a standard Boolean expression) evaluates to true at some point in some test case and evaluates to false at some point in some other test case in the test suite.*

We also refer to this coverage criterion as table coverage since AND/OR tables typically are used for every decision in RSML^{-e} . As an example, consider the transition defined for the `ROLL` state variable in Figure 4.

If we consider the transitions to `Selected` guarded by the condition encapsulated in the `Select_ROLL()`, test cases to provide decision coverage of this decision can be generated using the following two trap properties.

1. $G((\text{Select_ROLL}()) \rightarrow \sim(\text{ROLL} = \text{Selected}))$
2. $G(\sim(\text{Select_ROLL}()) \rightarrow (\text{ROLL} = \text{Selected}))$

6.3 Clause-Wise Transition Coverage

Finally, to exercise the approach with a complex test coverage criterion, we look at the code based coverage criterion called modified condition/decision coverage (MC/DC) and define a similar criterion. MC/DC was developed to meet the need for extensive testing of complex boolean expressions in safety-critical applications [4]. Ideally, one should test every possible combination of values for the conditions, thus achieving *compound condition coverage*. Nevertheless, the number of test cases required to achieve this grows exponentially with the number of conditions and hence becomes huge or impractical for systems with tens of conditions per decision point. MC/DC was developed as a practical and reasonable compromise between decision coverage and compound condition coverage. It has been in use for several years in the commercial avionics industry. A test suite is said to satisfy MC/DC if executing the test cases in the test suite will guarantee that:

- every point of entry and exit in the program has been invoked at least once,
- every basic condition in a decision in the program has taken on all possible outcomes at least once, and
- each basic condition has been shown to independently affect the decision's outcome

where a basic condition is an atomic Boolean valued expression that cannot be broken into Boolean sub-expressions. A basic condition is shown to independently affect a decision's outcome by varying only that condition while holding all other conditions at that decision point fixed. Thus, a pair of test cases must exist for each basic condition in the test-suite to satisfy MC/DC. However, test case pairs for different basic conditions need not necessarily be disjoint. In fact, the size of MC/DC adequate test-suite can be as small as $N + 1$ for a decision point with N conditions.

If we think of the system as a realization of the specified transition relation, it evaluates each guard on each transition to determine which transitions are enabled and thus each guard becomes a decision point. The predicates in turn are constructed from clauses – the basic conditions.

Definition 3. *A test suite is said to achieve clause-wise transition coverage (CTC) for a given transition of a variable in an RSML^{-e} specification, if every basic Boolean condition in the transition guard is shown to independently affect the transition.*

Consider the following transition example adopted from an avionics system related to the FGS:

EQUALS PowerOn IF

TABLE

PREV_STEP(DOI) IN_STATE AttemptingOn	: F T;
PREV_STEP(DOI) IN_ONE_OF {PowerOff, Unknown}	: T F;
DOIStatus = On	: T T;
AltitudeStatus IN_STATE Below	: T *;
ivReset	: F F;

END TABLE

To show that each of the basic conditions in the rows independently affects the transition, one should produce a set of test cases in which for any given basic condition there are two test cases, such that one makes the basic condition *true* and the other makes it *false*, the rest of the basic conditions have the same truth values in both test cases, and in one test case the transition is taken while in the other it is not. For the purposes of this example, let us just consider the first column. We may generate the trap properties by examining the truth value for each row in the first column as follows:

0. $G(\sim R_1 \ \& \ R_2 \ \& \ R_3 \ \& \ R_4 \ \& \sim R_5) \rightarrow \sim(\text{POST});$
1. $G((R_1 \ \& \ R_2 \ \& \ R_3 \ \& \ R_4 \ \& \sim R_5) \rightarrow \text{POST});$
2. $G(\sim R_1 \ \& \sim R_2 \ \& \ R_3 \ \& \ R_4 \ \& \sim R_5) \rightarrow \text{POST};$
3. $G(\sim R_1 \ \& \ R_2 \ \& \sim R_3 \ \& \ R_4 \ \& \sim R_5) \rightarrow \text{POST};$
4. $G(\sim R_1 \ \& \ R_2 \ \& \ R_3 \ \& \sim R_4 \ \& \sim R_5) \rightarrow \text{POST};$
5. $G(\sim R_1 \ \& \ R_2 \ \& \ R_3 \ \& \ R_4 \ \& \ R_5) \rightarrow \text{POST};$

where R_i stands for the basic condition in the i^{th} row of the table and POST represents the post-state condition DOI = PowerOn.

MC/DC test cases come in *pairs*, one where the atomic condition evaluates to false and one where it evaluates to true, but no other atomic conditions in the Boolean expression are changed. In the example above, trap properties 0 and 1 provide coverage of R1. Unfortunately, the model checking approach to test case generation is incapable of capturing such constraints over two test sequences. To work around this problem, we have developed a novel alternative that leverages a model checker for complete and accurate MC/DC test case generation. We automatically rewrite the system model by introducing a small number of auxiliary variables to capture the constraints that span more than one test-sequence. We also introduce a special system reset transition to restore a system to its initial state. With these small modifications, a test constraint spanning two sequences in the original model can be expressed as a constraint on a single test-sequence in the modified model. Model-checking techniques can then be employed to generate this single test sequence which can be later factored into two separate test-sequences for the original model satisfying the actual test criteria. This process has been fully automated, and used to generate the MC/DC like tests in this case study.

To summarize, we have automated the generation of trap properties for a collection of structural coverage criteria of formal specifications. In this case study we are using the three representative criteria described above; state coverage, decision coverage, and clause-wise condition coverage. The experiential results of using model checkers to generate test suites to these coverage criteria are presented next.

7 Experimental Results and Discussion

The results of our case study are presented in Tables 2 through 4. Table 2 provides a count of the number of trap properties generated for each FGS model for each coverage criterion. Note that this number reflects the naive generation of trap properties – we simply generate one trap property for each structural element we aim to cover. Naturally, the desired coverage can typically be achieved with substantially fewer test cases – see discussion later in this section. Recall, however, that the aim of this case study was not to provide a minimal set of test cases providing the desired coverage, but instead to evaluate the scalability of using model checking techniques for test case generation – thus, we wanted to work with many properties to make our results representative of expected performance. Finally, note that each trap property for MC/DC coverage describes an *MC/DC pair* of test cases – we will have twice as many MC/DC test cases as we have trap properties.

Table 3 gives the performance figures in terms of time and memory of generating the suites to the three coverage criteria (a - in the table indicates that a run of the model checker was terminated after an excessively long run – more than 24 hours).

As the data in Table 3 illustrates, symbolic model checking does not seem to scale well beyond FGS03. For models FGS04 and FGS05, it quickly runs into problems. From Table 3 it is clear that memory usage is not the problem. To keep

Table 2. Trap Properties generated per test criterion for all FGS Models

	FGS00	FGS01	FGS02	FGS03	FGS04	FGS05
State Coverage	33	54	78	99	117	246
Table Coverage	40	68	98	136	196	342
MCDC Coverage	32	54	77	118	205	323

Table 3. Execution Times and Memory Usage for all FGS Models

	State Coverage				Table Coverage				MCDC Coverage			
	Symbolic		Bounded		Symbolic		Bounded		Symbolic		Bounded	
	Time (s)	Memory (MB)	Time (s)	Memory MB (depth)	Time (s)	Memory (MB)	Time (s)	Memory MB (depth)	Time (s)	Memory (MB)	Time (s)	Memory MB (depth)
FGS00	1.01	1	0.29	1 (10)	2.04	10	0.79	10 (10)	1.72	10	3.55	13 (5)
FGS01	4.58	11	0.61	11 (10)	7.3	15	2.14	14 (10)	6.72	15	11.4	19 (5)
FGS02	33.86	21	1.76	15 (10)	82.34	24	5.94	21 (10)	78.2	24	51.76	30 (5)
FGS03	251.37	27	3.74	19 (10)	469.91	33	11.99	29 (10)	520.95	32	137.34	49 (5)
FGS04	-	-	57.89	32 (12)	-	-	101.2	53 (12)	-	-	39167.8	110 (5)
FGS05	-	-	81.61	58 (12)	-	-	193.67	80 (12)	-	-	46196.91	165 (5)

memory usage small, we are using the dynamic BDD variable reordering feature of NuSMV – without this option, NuSMV would exhaust the available memory quickly. Nevertheless, the dynamic variable reordering is quite costly and this deteriorates the performance of NuSMV to a point where the time to reorder becomes unbearable. In addition, the cost of constructing counterexamples in a symbolic model checker becomes a serious issue when the model checker is used for test case generation since we need a large number of counterexamples.

The bounded model checker, on the other hand, scales well to all FGS Models. To determine the search depth for the bounded model checker, we used results from a previous study using symbolic model checking for verification of the FGS system models [5]. In this previous study, we found that the full state space of FGS00 through FGS03 could be explored with 5 steps and with 12 steps in FGS04 and FGS05. Therefore, when generating state and table coverage, we simply used the default setting of 10 steps for FGS00 through FGS03 and extended it to 12 for FGS04 and 05. We attempted the same settings when generating MC/DC coverage, but the time required to search to this depth was simply unacceptable. Note here that the majority of the time was spent searching for test cases that are infeasible – a certain MC/DC pair did not exist. Searching to depth 12 for such non-existent test cases is counterproductive. Instead, we observed that the average test case length is quite short (Table 4 shows just a little over 1 for table coverage) and we simply set the search depth to a prudent 5. We expected this to assure that we found a large number of test cases, but did not waste any time searching for the ones that did not exist. Naturally, we may still miss some test cases that are longer than 5 should they exist (see discussion below).

As can be seen from the performance data for the bounded model checker in Table 3, even with a reduced search depth, the performance deteriorated quite notably when generating tests for MC/DC coverage (orders of magnitude slower

than for the other coverage criteria). Two factors contribute to this phenomenon; (1) the length of the test sequences generated and (2) the complexity of the LTL properties to check.

Table 4 shows the average test case length we measured during our experiments. From the results it is clear that the test cases for MC/DC coverage were approximately three times as long as the ones for the other coverage criteria. Recall the short discussion on MC/DC in Section 6. The counterexample generated for an MC/DC trap property describes not one test case, but an MC/DC *test case pair* – the first test case takes a transition t out of state X with a particular truth assignment to the basic conditions, the second takes us back to state X but this time we have exactly *one* basic condition with a different truth assignment and we do not take transition t . Thus, the test case length is destined to be approximately twice the length of the test cases generated for the other criteria. The need for a deeper search dramatically decreases the performance of the bounded model checker.

Table 4. The average length of the test cases generated

	State Coverage		Table Coverage		MCDC Coverage	
	Symbolic	Bounded	Symbolic	Bounded	Symbolic	Bounded
FGS00	2.7	1.0	4.7	1.2	8.0	3.1
FGS01	3.0	1.0	4.8	1.2	7.7	3.0
FGS02	4.3	1.0	5.4	1.2	8.1	3.1
FGS03	3.7	1.0	5.4	1.1	7.7	3.1
FGS04	-	1.0	-	1.2	-	2.9
FGS05	-	1.0	-	1.1	-	2.9

In addition to the increased test case length, the LTL properties characterizing the test cases are significantly more complex for MC/DC coverage than for the other coverage criteria (again, see Section 6). The dramatically longer trap properties negatively affects the performance of the bounded model checker [24].

From this discussion we can conclude that a bounded model checker seems to be a suitable tool for test case generation from formal specifications; it scales well to systems of industrial relevance, it generates the shortest possible test cases, and it is fully automated. There are, however, some drawbacks. Most importantly, if the shortest test case needed to cover a specific feature in the model is longer than the search depth of the bounded model checker, we have no way of telling if the test case simply does not exist or if it is longer than the search depth. This is an issue particularly for MC/DC generation where there are a fair number of MC/DC pairs that simply do not exist – if the bounded model checker fails to find a test case, the determination if it indeed exists is now a manual process.

As mentioned previously, during the generation of the test suites we did not attempt to minimize the number of test cases to achieve a desired coverage. In fact, it is easy to see that our test case generation approach, where a test case is generated for each trap property, will lead to a large amount of duplicate coverage.

We performed a simple analysis to measure the level of duplication for the test suite generated to achieve table coverage – we simply executed the tests (sequentially) and kept track of the coverage achieved after each test. Most test cases did not increase the coverage of the test suite – a clear indication that tests are redundant with respect to achieving coverage. Some tests, on the other hand, produced a “jump” in the coverage indicating that they exercise a new portion of the software. We created a reduced test suite using the tests that caused this jump in the coverage. Although this is not necessarily the minimal set to achieve the desired coverage, we found the difference in size between the initial set and this set to be such that it clearly indicated a large degree of duplication over the generated tests. For example we found that for FGS00 and FGS03 respectively, only 3 out of the 27, and, 11 out of 95 test cases were required to achieve table coverage. Since the cost of resetting a system and executing a new test case in some applications is high, identifying a small test suite that provides adequate coverage is of some importance. In future work we will investigate an iterative approach to the test case generation in order to achieve smaller test suites. Of course, a smaller test suite might achieve the same coverage, but it may reduce the defect detection capability of the test suite. We have just initiated a study to investigate how test suite size impacts the defect detection capability of the suite.

As mentioned in Section 2, we intended our test case generation framework to allow an analyst to generate test sequences from a formal specification and then run the tests on the implementation. Should additional tests be needed, we would like to generate the additional input sequences from the code. To this effect we evaluated an explicit state code model checker.

8 Java PathFinder Results

We have done some preliminary experiments on using the Java PathFinder (JPF) code-level model checker [25] to do test case generation on Java programs automatically generated from the RSML^{-e} models. Our initial motivation for using a code-level model checker was to investigate whether one can use such a tool to discover test cases for covering code that was not being covered by the test cases derived from the RSML^{-e} specification. Here however, since we are doing an automatic translation of RSML^{-e} to Java, we will use our preliminary results to judge how an explicit-state model checker (such as JPF) compares to the symbolic and bounded model checker approaches for test case generation for RSML^{-e}. We studied test cases for branch coverage at the Java level, since branch coverage is an often used code coverage criteria and, due to the translation used, corresponds closely to table (decision) coverage at the RSML^{-e} level.

The test case generation process using JPF is currently not automated, in particular, the trap properties are assertions added by hand, and, there is no facility to extract the test inputs from each counterexample produced. We therefore will not be reporting any specific timing and memory usage results, but rather make general observations. In short, the explicit-state model checker did not perform as well as the symbolic and bounded model checking approaches. For

FGS00 the model checker could generate test cases to cover all branches within a matter of seconds while using an insignificant amount of memory (less than 1 Mb). Whereas for FGS03, it could not generate enough of the state-space to cover all the branches in a reasonable amount of time (3 hours). We did not attempt to generate tests for FGS04 or FGS05.

From the experiments it is clear that the explicit-state model checking is particularly sensitive to the length of the test cases required to achieve the desired coverage. For example, FGS03 has 10 boolean inputs at each input cycle (i.e., 2^{10} options), and the explicit-state model checker can at most deal with 3 such inputs.

Explicit-state model checking does however allow more control over the search, and we conjecture this can be exploited to do efficient test case generation. Specifically, one can use heuristic search [11] techniques to find the desired test cases – we will pursue this line of research in future work. Recently, the idea of combining symbolic execution with model checking to do test case generation has been proposed [17] – this allows one to mitigate the effect of longer test cases and should therefore allow for more efficient test case generation. This latter approach is in some ways similar to doing bounded model checking, and we will investigate how these techniques compare in future work.

9 Summary and Conclusions

To summarize, we have conducted a series of case studies evaluating how well model checking techniques will scale when used for test case generation. Our experiences point out limitations of symbolic as well as explicit state model checkers. A bounded model checker, however, performed very well and shows tremendous promise. The domain of interest in our study has been safety critical reactive systems – systems that lend themselves to modeling with various formalisms based on finite state machines. In this domain, test cases providing common coverage seem to be quite short, thus making bounded model checkers perform very well.

Naturally, there are still many challenges to address. There are systems where the cost of restarting the system to execute a new test sequence is quite high. In this situation it is highly desirable to have long test cases that provides extensive coverage so that we can minimize the number of system restarts required to execute the test suite. The bounded model checking approach discussed performing well in our case study provides the exact opposite – we will get many very short test cases. Techniques to effectively merge these test cases to longer test sequences would be highly desirable. Alternatively, techniques based on explicit state model checking and heuristic searches may be able to provide long test cases that provides extensive coverage of a model. We plan to investigate this approach in the context of Java PathFinder shortly.

The nature of bounded model checking makes it unsuitable for verification. Determining the appropriate search depth to guarantee that we find most (if not all) test cases without wasting time with deep searches for test cases do not exist remains a challenge.

Our case study example is atypical in that it only contains discrete variables, we have no integer or real variables in the model. Naturally, many models will have numerous numeric variables involved in various interrelated numeric constraints. Applying the model checking techniques on these systems will be a challenge. Recent advances bringing efficient decision procedures and bounded model checking together promises to help to some extent. Various abstraction techniques, for example, iterative refinement [14,8] and domain reduction abstraction [6], also holds promise in this regard. We hope to conduct experiments on systems with these characteristics shortly.

References

1. Paul E. Ammann and Paul E. Black. A specification-based coverage metric to evaluate test sets. In *Proceedings of the Fourth IEEE International Symposium on High-Assurance Systems Engineering*. IEEE Computer Society, November 1999.
2. Paul E. Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54. IEEE Computer Society, November 1998.
3. J. Callahan, F. Schneider, and S. Easterbrook. Specification-based testing using model checking. In *Proceedings of the SPIN Workshop*, August 1996.
4. J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, pages 193–200, September 1994.
5. Yunja Choi and Mats Heimdahl. Model checking RSML^{-e} requirements. In *Proceedings of the 7th IEEE/IEICE International Symposium on High Assurance Systems Engineering*, October 2002.
6. Yunja Choi, Sanjai Rayadurgam, and Mats Heimdahl. Automatic abstraction for model checking software systems with interrelated numeric constraints. In *Proceedings of the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE-9)*, pages 164–174, September 2001.
7. E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.
8. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification*, pages 154–169, July 2000.
9. A. Engels, L. M. G. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In *Proceedings of TACAS'97, LNCS 1217*, pages 384–398. Springer, 1997.
10. Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. *Software Engineering Notes*, 24(6):146–162, November 1999.
11. A. Groce and W. Visser. Model checking java programs using structural heuristics. In *Proceedings of the International Symposium on Software Testign and Analysis 2002*, Rome, Italy, July 2002.
12. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

13. Mats P.E. Heimdahl, Sanjai Rayadurgam, and Willem Visser. Specification centered testing. In *Proceedings of The First International Workshop on Automated Program Analysis, Testing and Verification, ICSE 2000*, 2000.
14. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Rupak Majumdar. Lazy abstraction. In *Proceedings of the 29th Symposium on Principles of Programming Languages*, January 2002.
15. Hyoung Seok Hong, Sung Deok Cha, Insup Lee, Oleg Sokolsky, and Hasan Ural. Data flow testing as model checking. In *Proceedings of 2003 International Conference on Software Engineering*, Portland, Oregon, May 2003.
16. Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A temporal logic based theory of test coverage and generation. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '02)*, Grenoble, France, April 2002.
17. S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Warsaw, Poland, April 2003.
18. N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering*, 20(9):684–706, September 1994.
19. NuSMV: A New Symbolic Model Checking.
Available at <http://nusmv.iirst.itc.it/>.
20. Sanjai Rayadurgam and Mats P.E. Heimdahl. Coverage based test-case generation using model checkers. In *Proceedings of the 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2001)*, pages 83–91. IEEE Computer Society, April 2001.
21. Sanjai Rayadurgam and Mats P.E. Heimdahl. Test-Sequence Generation from Formal Requirement Models. In *Proceedings of the 6th IEEE International Symposium on the High Assurance Systems Engineering (HASE 2001)*, Boca Raton, Florida, October 2001.
22. Jeffrey M. Thompson, Mats P.E. Heimdahl, and Steven P. Miller. Specification based prototyping for embedded systems. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, number 1687 in LNCS, pages 163–179, September 1999.
23. Jeffrey M. Thompson, Michael W. Whalen, and Mats P.E. Heimdahl. Requirements capture and evaluation in NIMBUS: The light-control case study. *Journal of Universal Computer Science*, 6(7):731–757, July 2000.
24. M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the 1st Symposium on Logic in Computer Science*, pages pp. 322–331, Cambridge, June 1986.
25. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proceedings of the 15th International Conference on Automated Software Engineering*, Grenoble, France, September 2000.

JMLAutoTest: A Novel Automated Testing Framework Based on JML and JUnit

Guoqing Xu and Zongyuan Yang

Software Engineering Lab, The Department of Computer Science,
East China Normal University
3663 N Zhongshan Rd., Shanghai 200062, P.R.China
{gqxu_02, yzyuan}@cs.ecnu.edu.cn

Abstract. Writing specifications using Java Modeling Language has been accepted for a long time as a practical approach to increasing the correctness and quality of Java programs. However, the current JML testing system (the JML and JUnit framework) can only generate skeletons of test fixture and test case class. Writing codes for generating test cases, especially those with a complicated data structure is still a labor-intensive job in the test for programs annotated with JML specifications.

This paper presents JMLAutoTest, a novel framework for automated testing of Java programs annotated with JML specifications. Firstly, given a method, three test classes (a skeleton of test client class, a JUnit test class and a test case class) can be generated. Secondly, JMLAutoTest can generate all nonisomorphic test cases that satisfy the requirements defined in the test client class. Thirdly, JMLAutoTest can avoid most meaningless cases by running the test in a double-phase way which saves much time of exploring meaningless cases in the test. This method can be adopted in the testing not only for Java programs, but also for programs written with other languages. Finally, JMLAutoTest executes the method and uses JML runtime assertion checker to decide whether its post-condition is violated. That is whether the method works correctly.

1 Introduction

Writing specifications of Java Modeling Language has been viewed as an effective and practical way of increasing the correctness and quality of Java programs for JML' great expressiveness and easy grammar which is similar to Java's. In addition, JML allows assertions to be intermixed with Java code [3, 6], which brings convenience to Java programmer. In the past few years, many tools have been implemented to support JML, including the compiler, runtime assertion checker [1] as well as its testing framework [2]. The current JML testing system (JML and JUnit testing framework) can generate the test fixture and the skeleton of the test case class automatically which sets programmers free from writing unit test codes, thus making unit test more accessible to programmers.

1.1 The Problem

However, programmers still need to spend a lot of time writing codes for generating test cases, especially those which represent complicated data structures (i.e. binary-tree, linkedlist). There have been some testing tools which can automatically generate test cases, such as Korat [4] and Alloy Analyzer (AA)[22], but they either supply the whole test case space generated to the test, never caring about how many test cases are meaningless¹ ones, or require programmers to write special predicates to get rid of meaningless test cases. For example, in Korat[4] programmers should write an additional method “public boolean repOk” in the input class to keep the test cases generated meaningful. In our opinions, at first, only identifying meaningless cases when test is run is not enough because a test with many meaningless inputs can tell little about the execution of tested method although maybe this test is based on a very large test case space and it might spend a lot of time dealing with meaningless ones. So avoiding meaningless test case before running the test is very important. In the second place, in many cases, the tester is not just the one who develops the class to be tested or it is a black box test, therefore, handling test cases totally depending on predicates provided by programmers is not a practical way.

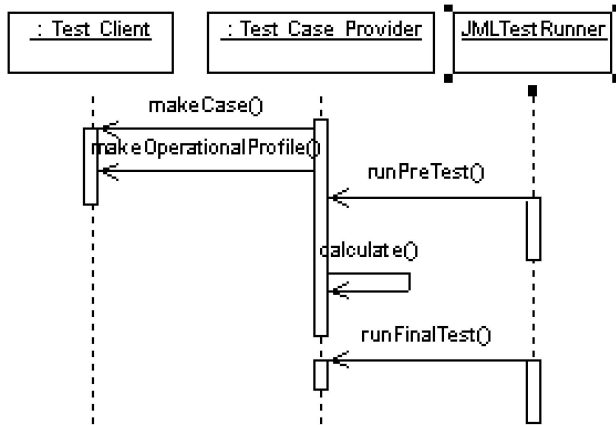


Fig. 1. The work flow of JMLAuto Test

1.2 How JMLAutoTest Deals with These Problems

In this paper, we present JMLAutoTest, an automatic testing framework, which can solve these problems well. Given a method annotated with JML specification, similar to the JML and JUnit testing framework [2], JMLAutoTest firstly generates a JUnit test class (*_JML_Test) which sets up test fixture and a test case provider class

¹ Meaningless test case here means the one which violates the pre-condition of method to be tested.

(*_JML_TestCase) which is a sub class of the test class. In addition, JMLAutoTest generates the skeleton of another class called test client (JMLTestClient), into which programmers can easily set class domains for classes of arguments of method to be tested or field domains for the fields in those classes. Then when test is run, the test case provider can get test cases from test client automatically.

JMLAutoTest can avoid most meaningless test cases by running the test in a double-phase way (runPreTest and runFinalTest in Figure 1). Double-phase way is a statistic based testing approach. Firstly, programmers should make a standard to divide the whole test case space into several partitions. This standard is somewhat like the “Operational Profile” in Cleanroom testing [5, 15], so here we also call it operational profile. Programmers can make the operational profile in the method makeOperationalProfile (shown in figure 1) in the test client. After the test case provider gets the generated test cases, it passes the whole test case spaces to the method makeOperationalProfile. Then this method divides the test case space into several partitions according to the criteria made by programmers and returns these partitions. During the first phase, tests with each group of test cases chosen from these partitions are run respectively. Each group only contains a relatively small number (a few dozen) of test cases.

Then based on the statistical principle we can know which partition of test case space produces the most meaningless test cases and which produces the second most ... Thus, the probability of meaningless test cases contained in each partition can be determined after the first phase test (pre-test). During the second phase, a large number of test cases should be taken out from each partition depending on proportion obtained after the first phase test. From this point of view, meaningless test cases can be avoided to a certain extent and programmer who runs the test only need to make the operational profile without knowing the details about the method to be tested. But the validity of this way is based on the quality of operational profile which is used to create different partitions. This way of testing can be applied to not only Java programs, but also programs with other languages.

The rest of this paper is organized as follows. Section 2 presents the algorithm and principle that JMLAutoTest uses to generate test case. Section 3 describes how double-phase testing works. Section 4 presents the test oracle generation for testing methods. Section 5 illustrates some experimental results. Section 6 reviews related work and Section 7 concludes.

2 Test Case Generation

The whole procedure of test case generation in JMLAutoTest includes three parts: finitization, validity checking and nonisomorphism. Figure 2 gives an overview of the algorithm of JMLAutoTest test case generation for the type “jmlautotest.example”. JMLObjectSequence is a utility class defined in the package org.jmlspecs.model [3, 6] which implements a sequence to contain objects. We use class Finitization (Section 2.1) to finish the work of generation. After a test case candidate is generated, JMLAutoTest uses JML runtime assertion checker to check its validity (Section 2.2)

in order to kill it if Invalid. Then JMLAutoTest visits existing test cases to make sure that this candidate is not isomorphic to a certain test case existing in the test case space (Section 2.3).

```
// The following method is defined in the test client
public JMLObectSequence makeCase_jmlautotest_example () {
    JMLObectSequence cases = new JMLObectSequence ();
    Finitization f= new Finitization (jmlautotest.example.class);
    //Create the value domain1 which contains 5 instances of
    //the class "example.field1.class" for the field //"field1"
    JMLObectSequence valuedomain1
        = f.createObjectSequence(example.field1.class, 10);
    // set value domain for the field "field1"
    f.set (example.field1, valuedomain1);
    f.generate();
    cases = f.getResult();
    return cases;
}
```

Fig. 2. The overview of the method `makeCase_*` in the test client

2.1 Finitization

JMLAutoTest provides a class `Finitization` for programmers to generate a finite test case space of any kinds. The whole process of the working of a finitization includes two parts: setting the value domain and generating.

Set Value Domain for Fields of the Input Class. Programmers can bind a certain field with a set of bounds by setting the value domain for the field. Then JMLAutoTest will create a candidate object by assigning to each field all possible values from its corresponding domain. The field domain is represented by an object either of the type `JMLObectSequence` which contains objects or of the type `JMLPrimSequence` which is another utility class to contain values of primitive types. All of these domains are inserted into a hash table for use in generating.

Generate Test Case Space. There are two kinds of method `generate` in class `Finitization`. One is to generate test case space for common classes and another is to generate test cases for special classes which implement a certain linked data structure such as `binarytree` and `linkedList`.

Figure 3 illustrates how to generate test case for the class which implements a linked data structure (linked list). The invariant clause at the beginning will be presented in Section 2.2. The method `generate` here is with two arguments. The first one is the name of the first node (or the root node) field in this recursive structure. The second one is an array of `String` which contains names of pointer fields in this data structure. In the example of `linkedList` shown in Figure 3, the first argument of method `generate` is the string `"first"`, the name of field `first` in class `LinkedList` which represents the first node of a linked list. The second one is a string array which only contains `"pointer"`, the name of the pointer field in `LinkedList`.

In the process of this kind of generation, the field *first* and the pointer field *pointer* share the same value domain. Each element of this value domain can only be used once except null. After one element is used, it will be removed from the domain. A special stack *visitedStack* is kept during the recursion to contain used elements of the value domain. Another stack called *fNames* is used to contain names of pointer fields. In the next recursion, one element in the value domain was assigned to the pointer field represented by the first element in *fNames* of the object represented by the first element in *visitedStack*.

For example, when JMLAutoTest is generating a binary tree, the situation of two stacks and value domain are illustrated in Figure 4. At the beginning, stack *visitedStack* only contains *binarytreeObj* which is an object of input class *BinaryTree*. The value domain contains four nodes: N_0 , N_1 , N_2 and null. During the first recursion, N_0 is assigned to the field *root* which is represented by the first element in *fNames* of the Object *binarytreeObj* represented by the first element in *visitedStack*. That means let *binarytreeObj.root* = N_0 . Then the first element in both *fNames* and *visitedStack* is removed. Also, the used element N_0 is inserted into *visitedStack* and its two pointer fields *left* and *right* are inserted into *fNames*. Recursion follows this algorithm until it reaches two states. One is that the value domain only contains null and another one is that all elements in *visitedStack* are null. The first state means finding a candidate while the second one means a failure.

To accelerate the generation of a linked data structure, programmers can choose to generate the test case space in a fast way. If the value of field *acceleratingEnabled* in the class *Finitization* is true, JMLAutoTest only considers a certain structure once regardless of other non-pointer fields. JMLAutoTest implements this optimization by assigning values to the pointer field only twice. For the first time, a non-null value in the value domain is assigned to the pointer field, and for the second time let the pointer field be null. Generating all cases of a binary tree with 7 nodes costs 1 second while in the normal way, it costs more than 1000 seconds because candidates handled by JMLAutoTest in the normal way are much more. However, the test case space generated in the fast way is so limited that it only contains all kinds of the structure without caring about non-pointer fields.

2.2 Validity Checking

After a candidate object is generated, JMLAutoTest checks its validity to judge whether it can be used as a test case. The invariant clause in Figure 3 says that if the field *first* is null, then the field *length* must be zero or if *first* is not null, then the field *length* must equal the real number of nodes in this list (Method *getLength* returns the value of field *length* and method *toObjectSet* functions at transforming the linked list to a set of nodes. Both of them are omitted in Figure 3). The validity checking in JMLAutoTest totally depends on the instance invariant specified in the input class.

The Invariant Clause in JML. An invariant [1] is a condition that remains true during the execution of a segment of code. A instance invariant, which constraints both static and non-static states of program execution, can refer to both static and

instance members. In JML, invariants belong to both pre-state and post-state specifications which are checked in both pre-state, i.e., right after a method call and argument passing but just before the evaluation of the method body and post-state, i.e., right after the evaluation of the method body but just before the method returns.

```

/*@ public invariant (first == null && getLength() == 0)
/*@ || (first != null && getLength() == toObjectSet().size());
public class LinkedList{
    public Node first; // the first node of a linked list;
    protected int length; // the length of this list
    ...
}
public class Node{
    public int ID; // node ID
    public Node pointer; // a pointer pointing to the next node
}

public JMLObjectSequence makeCase_LinkedList(){
    Finization f = new Finization_LinkedList(class);
    //create 3 instances of Node with an argument array [0,1,2] for the
    //constructor
    JMLObjectSequence nodes = f.createObjects(Node.class, new
        JMLPrimSequence(new int[]{0,1,2}), 3);
    f.add(nodes, null); // add null to this domain.
    f.set("first", nodes); // set the value domain for "first"
    // set domain for the field "length"
    f.set("length", new JMLPrimSequence(1,4));
}
// Generate candidates recursively
f.generate("first", new String[]{"pointer"});
return f.getResult();
}

```

Fig. 3. Generate the test case space for class LinkedList

Value domain of the field root and pointer fields “left” and “right”

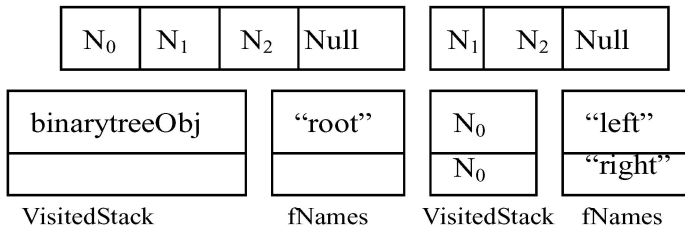


Fig. 4. The difference of situations of value domain and stacks between at the beginning and after a recursion when generating a binarytree

The Invariant Clause in JML. An invariant [1] is a condition that remains true during the execution of a segment of code. A instance invariant, which constraints both static and non-static states of program execution, can refer to both static and instance members. In JML, invariants belong to both pre-state and post-state specifications which are checked in both pre-state, i.e., right after a method call and

argument passing but just before the evaluation of the method body and post-state, i.e., right after the evaluation of the method body but just before the method returns.

```
public void checkInv$instance$T(){
    Boolean rac$v = false;
    [P, rac$v ]
    if (! rac$v){throw new JMLInvariantError() ;}
}
```

Fig. 5. The method to check the instance invariant translated by JML runtime assertion checker

```
public boolean checkValidity(Object obj){
    try{
        //get the name of the input class
        String className = obj.getClass().getName();
        //get the name of the method of checking invariant
        String invName = "checkInv$instance$" + className;
        Method thisMethod = obj.getClass().getMethod(invName, new
        Class[]{}); // get the method of checking invariant
        //invoke this method
        thisMethod.invoke(obj, new Object[]{});
    }
    catch(java.lang.NoSuchMethodException ex){
        throw new Exception("code for class
        "+obj.getClass().getName() +" was not compiled with jmlc so
        no assertions will be checked");
    } // There is no such a method in the class
    catch(JMLInvariantError ex){
        return false; // Invariant has been violated.
    }
    return true; // Okay
}
```

Fig. 6. Check whether the candidate is valid

Let T be a type with a set of instance invariants, $P_1 \dots P_n$. The invariants are first conjoined to form a single invariant predicate, i.e., $P \equiv P_1 \wedge \dots \wedge P_n$. The conjoined invariant predicates are translated into instance invariant methods, whose general structures are shown in Figure 5. The notation $[P, rac\$v]$ denotes translated code that evaluates the predicate P and stores the result into the variable $rac\$v$. The invariant methods evaluate the conjoined invariant predicates and throw invariant violation errors if the predicates do not hold.

Invariant Checking in JMLAutoTest. Figure 6 illustrates how invariant is checked in JMLAutoTest. If the method *checkInv\$instance\$T* is not found in the input class, this class was not compiled by JML compiler that a new exception was thrown.

If a `JMLInvariantError` exception is caught when the method is invoking, this candidate is not valid because the invariant is violated. If no exceptions are caught, this candidate is valid and the method *checkValidity* returns true. If no invariants specified in the input class, the method *checkValidity* always returns true.

2.3 Nonisomorphism

At the end of the process of generation, JMLAutoTest explores the space of generated test cases to make sure the candidate is not isomorphic to a certain existing test case. We do not define what isomorphism is in JMLAutoTest. Our solution is totally based on the method *equals* defined in the input class.

Let *Obj* be the candidate object and let *S* be the set of test cases generated. *Obj* is isomorphic to an existing test case iff $\exists c \in S. \text{Obj.equals}(c)$. There is an advantage of this solution that programmers can easily change the criteria to make different kinds of test cases by modifying the *equals* method.

3 Double-Phase Testing

This section presents how JMLAutoTest avoids meaningless test cases. After the generation of the test case space presented by the previous section, there might be many meaningless ones in the space whereas individual candidate itself is valid. The major idea of double-phase testing is to use two phases of testing based on statistics.

Double-phase testing is especially fit for the black box test and the test with a large test case space although it will spend some time running pre-test. If the test case space is not very large (maybe only contains a few dozen of cases), we do not need to use the double-phase testing. Programmers can decide whether to use this method by choosing whether to run the test with an argument “-pre”. Running the test case provider (class `*_JML_TestCase`) without any arguments means the test will be run in a conventional way.

3.1 Making an Operational Profile

There is a method *makeOperationalProfile* in the test client. Programmers can modify this method to divide the whole test case space into several partitions. JMLAutoTest can generate the skeleton codes of this method. Test case spaces of different types are put into a hash table spaces in the test case provider. Then this hash table is passed to the method *makeOperationalProfile* in test client. For example, we can put linked lists which contain more than three nodes into `partition_LinkedList[0]` and put those which contain less than three nodes into `partition_LinkedList[1]`. Variable *partition_LinkedList* is a `JMLObjectSequence` array, each element of which represents a partition of this test case space. Finally partitions of each type are put into a hash table *partitions* which will be returned by this method.

3.2 The First Phase

During the first phase, a small number of test cases taken from each partition randomly make several groups. The percentage of test cases which should be taken out from each partition is provided by the programmer as the second argument of the main method (i.e., if arguments are “-pre 0.3”, the percentage is 30%). Tests with each group are run respectively. Figure 7 illustrates the generated codes for the first-phase test. If the pre-test is not disabled by programmer, let *isPre* (a flag to show whether it is a pre-test or a final test) be true.

Taking Test Cases Out from Partitions. The technique of getting these test cases is based on the following principle.

Let p be the percentage provided by the programmer. Let P be a partition of a certain test case space, and Let C be a sub set of P . C is the set of cases which should be taken out iff $C.size == \lceil P.size * p \rceil$ and $\forall n \in [0, C.size-1], C.elementAt(n) == P.elementAt(\lfloor n/p \rfloor)$. The operator $==$ is Java’s comparison by object identity. The definition above illustrates that the process of getting test cases is totally based on the average distribution in statistic.

Algorithm Used in Running the Pre-test. In section 3.1, we have presented that each partition of test case space of a certain type is represented by an element in a *JMObjectSequence* array. We use an algorithm to record the number of meaningless test cases during the first phase that we shift the sequence of elements in a partition backward and let the first unit contain the number of meaningless cases. At first, we put zero into its first unit.

When the test suite is run, the sum of number of meaningless test cases taken from a certain partition in the current test and the value in its first unit is put into this unit of the partition. So during the first phase, the value in the first unit of a partition changes for several times. Through comparing the final value in the first unit of a partition with one another in the same test case space, we can get the approximate proportion of meaningless cases in this partition among all those meaningless in the whole test case space.

Method *runPreTest* shown in Figure 7 is a recursive method which explores every partition in test case space of each type. We keep a hash table *arg_ind* which represents a vector of indices of partitions in test case space of each type. We continue with the example method *findSubList*. There are two test case spaces generated for the type *LinkedList* and *Node*. If test case space of *LinkedList* has been divided into two partitions and the space of *Node* has been divided into three partitions, in the first phase, test suite should be run for six times. Each time before test suite is run; the value of *arg_ind* should be changed to show the indices of partitions in these two test case spaces. At the beginning, *arg_ind* is the vector (0, 0) which means test cases of both type *LinkedList* and *Node* should be taken out from No.0 partition in the two test case spaces. During the first recursion, *arg_ind* should be changed to (0, 1) and the

recursion ends when *arg_ind* reaches (1, 2) which means all partitions have been visited. When test suite is run, the *init_vT* methods presented in the next section get the test cases from different partitions according to the vector represented by *arg_ind*.

```
//The following codes appear in the test case provider
//(*_JML_TestCase.java) for testing the method "findSubList".

//an object of test client
JMLTestClient myClient = new JMLTestClient();
//Initialize hashtable param in order to receive test cases
Hashtable param = new Hashtable();
// Get test cases from JMLTestClient
param.put("LinkedList",
        myClient.makeTestCases_LinkedList());
// Divide the whole test case space into several partitions
Hashtable args_findSubList
        = myClient.makeOperationalProfile(param);
// If the pre-test has not been disabled by programmer.
if (args.length==2 && args[0]=="-pre"){
// A flag to show whether it's a pre-test or a final-test
boolean isPre = false;
//Percentage of test cases which should be taken from each
//partition
double percentage = Double.parseDouble(args[1]);
// run test suite for several times
runPreTest( percentage );
```

Fig. 7. Generated codes in test case provider (*_JML_TestCase) for the first phase test

3.3 The Second Phase

During the second phase, test cases from partitions should be reorganized and the final test is run.

Reorganization of Test Cases. After the first phase test, the first unit of each partition has contained the number of all meaningless test cases taken from this partition. Although this number might be more than the real one, it can reflect the real situation of each partition in a certain test case space. After the disposal of these numbers, we can get the proportion of test cases which should be taken out from each partition of a certain test case space in the final test. Let *S* be the set of such proportions. Let *C* be the set of values contained in the first unit of each partition in a test case space. The following algorithm is used in JMLAutoTest to get the test cases in the final test.

Algorithm: $s = \text{sum}(C)$;

forall $n \in [0, C.\text{size}-1]$, $S.\text{elementAt}(n) = (s - C.\text{elementAt}(n)) / s$;

Finally, in a certain test case space, let T_1 be the set of test cases taken from partition P_1 , T_2 be the set of cases taken from the partition $P_2, \dots$. T_n be the set taken from partition P_n . The operation of taking test cases from P_i is also based on the average distribution with the proportion $S.\text{elementAt}(i-1)$. Then we have the equation

$\forall i \in [1, n], T_i.size == \lfloor P_i.size * S.elementAt(i-1) \rfloor$. Let T be $T_0 \cup T_1 \dots \cup T_i$. Then T is the set of test cases to be supplied in the final test.

```

/** The following method is defined in the test class and
    is overridden in test case provider class. */
public void init_vLinkedList() {
    if (isPre) { // if it is in the first phase
        /** Get the index of partition from which test cases should
            be taken */
        int ind= ((Integer
                    arg_Ind.get("LinkedList")).intValue());
        /** Initialize vLinkedList with cases from the partition
            whose index is represented by ind */
        ...
    }
    else { // It is a final test
        /** Initialize vLinkeList with cases from all partitions
            according to the obtained proportions. */
    }
}

```

Fig. 8. The method `init_vT` in test case provider class

4 Test Oracle Generation

JMLAutoTest uses the same way of generating test oracles as JML+JUnit testing framework [2] which combines JML [3] and JUnit [7] to test individual method.

4.1 Setting up Test Fixture

The test fixture for the class C is defined as:

$C[]$ receivers; $T_1[]$ vT₁; ... ; $T_n[]$ vT_n;

The first array named receivers is for the set of receiver objects (i.e., objects to be tested) and the rest are for argument objects.

The receiver and argument objects are initialized by the method `init_receivers` and `init_vTi` in the test case provider class. Figure 8 describes generated codes in method `init_vLinkedList` for initializing `vLinkedList`. If it is in the first phase, test cases are taken from the partition, the index of which is represented by the value in the hash table `arg_ind` and test cases should be taken out from all partitions and mixed together in the second phase.

4.2 Testing a Method

For each instance (i.e., non-static) method of the form:

$T \ M(A_1 a_1, : : , A_n a_n) \text{ throws } E_1, \dots, E_m \ \{ /* \dots */ \}$

of the class C , a corresponding test method `testM` is generated in the test class `C_JML_Test`. Let n be the value of `vT1.length * vT2*...* vTn.length`. Then, the

method to be tested is executed for n times until each element in each array (vTi) has been visited. Pre-condition of the target method is checked firstly. If the pre-condition has been violated and the current test is the pre-test ($isPre==true$), let the variable meaningless shown in figure 8 increase. If the post-condition of the method is violated, JMLAutoTest handles it in different ways in two phases. During the first phase, this exception is ignored and the test continues since we just want to know the number of meaningless test cases not caring about whether the execution fails or succeeds. But during the second phase, this exception should be thrown to let programmer know execution of the method is not correct.

5 Experimental Results

This section presents performance of JMLAutoTest on testing a method. To monitor the process of test case generation and testing a method, JMLAutoTest uses a class JMLTestDataStat to record some key data. We use method `findSubTree(BinaryTree parentTree, Node thisNode)` whose function is to find a sub tree whose root is represented by *thisNode* in the parentTree as the benchmark for which we show JMLAutoTest's performance.

5.1 Generating Test Cases and Dividing Test Case Spaces

Figure 9 describes the definition of the BinaryTree and Node. The invariant clause tells us if root is null, the size must be 0. If root is not null, the size must equal the number of total nodes in the tree. What Figure 10 illustrates is the JML specifications for the method `public BinaryTree findSubTree(BinaryTree parentTree, Node thisNode)`. The pre-condition of the method requires that neither of its arguments can be null and there must be a node in parentTree whose ID equals the ID of thisNode.

We generate the test case space of type BinaryTree with a few nodes whose ID are ranging from 5 to 8. We also generate the case space of type Node which contains 12 nodes whose IDs are from 0 to 11.

For the test case space of type BinaryTree, We do not divide it and leave it as the only partition. For the space of type Node, We divide it into two partitions. The first one contains nodes whose ID varies from 0 to 5 and the second one contains the rest.

5.2 Test Results

Table 1 shows JMLAutoTest's performance when we test the method with binary trees containing nodes from 5 to 8. We generate the test case space of BinaryTree in the fast way, so the number of candidates considered is close to that of test cases generated. We use the arguments "-pre 0.25" to run the test. Note that for all kinds of binary trees listed in table 1, almost all test cases in the final test are meaningful.

```

public class BinaryTree{
  //@ public invariant (root ==null &&
  //@   getSize() ==0) || (root!=null&&getSize()!=0
  //@   &&toObjectSet(root).size() == getSize());
  public Node root;
  protected int size;
  public int getSize(){... }
  public JMLObjectSequence toObjectSet(){...}
  ... }
  public class Node{
    public Node left;
    public Node right;
    public int ID; }

```

Fig. 9. The Definition of class BinaryTree and Node

```

/*+@ public normal_behavior
  @ requires parentTree!=null && thisNode!=null &&
  @(\exists Node n; @parentTree.toObjectSet().has(n);
  @n.ID== thisNode.ID);
  @ assignable \nothing;
  @ ensures \result.root.ID == thisNode.ID ;
  @+*/

```

Fig. 10. The pre-condition of method findSubTree

Then we make a comparison between the performance of testing in double-phase way and the conventional way (Table 2). Note that for all binary trees with more than five nodes, total time of the test in double-phase way is less than the corresponding time in the conventional way and the more test cases are, the more time double-phase testing can save. Although some meaningful test cases have also been filtered out in double-phase testing, the test case space is still large enough to test the correctness of the method.

6 Related Work

There are now quite a few testing facilities and approaches based on formal specifications developed and advocated by many different research groups. One of the earliest papers by Goodenough and Gerhart [8] presents its importance. Approaches like automated test case generation from UML statecharts [9,21] and Z specifications [10,20] present ways of generating test cases from formal specifications. There are also some tools which can generate Java test cases like the TestEra framework [11,22] which requires programmers to learn a specification language based on which, test cases can be generated. All these specifications do not generate test cases for Java programs annotated with JML specification which is widely accepted as the ancillary tool tailored for Java to keep the correctness of programs.

Several researchers noticed that if a program is formally specified, it should be possible to use the specification as a test oracle [12, 13, 14]. Cheon and Leavens [2] present the JMLUnit framework which can generate test oracles for JUnit [17] from

JML specifications. This framework uses JML runtime assertion checker to decide whether methods are working correctly, thus automating the writing of unit test oracles. However it has not automated the generation of test cases which is still a labor-intensive for programmers.

Table 1. Performance of JMLAutoTest for testing the method findSubTree with arguments “-pre 0.25” (test cases is generated in the fast way).

nodes in binary tree	generated binary trees	candidates considered	meaningful cases in the final test	total test cases in the final test
5	42	64	410	492
6	132	196	1572	1572
7	429	625	5136	5136
8	1430	2055	17148	17148

Table 2. Performance comparison between the double-phase testing in JMLAutoTest and the conventional way in JMLUnit testing framework.

nodes in binary tree	Double-phase way			Conventional way		
	meaningful /total in final test	time in the first phase (s)	total time of the test (s)	meaningful /total in final test	time in the first phase (s) ²	total time of the test (s)
5	410/492	0.079	0.266	420/1008	0	0.219
6	1572/1572	0.188	0.422	1584/3168	0	0.468
7	5136/5136	0.36	0.766	6006/10296	0	1.25
8	17148/17148	0.703	2.016	22880/34320	0	3.484

Boyapati, Khurshid and Marinov describe Korat [4] which can finish automated generation of test cases for Java programs with formal specifications. Korat generates linked data structures based on additional Java predicates. However Korat requires that the programmer who runs the test must know well about the details of the program to be tested, therefore it is not fit for a black box test. Also, it can not keep meaningless test cases from being handled.

There are quite a few approaches to applying the statistical models to Testing [16,18,19]. Statistical testing has been widely adopted during the development of the Cleanroom software[5] in test cases acquisition, results evaluation and reliability modeling. So it is not a new idea to use the statistical analysis in testing. But in JMLAutoTest the novelty lies in applying the statistical analysis to filtering out meaningless cases. This idea can also be used in testing of programs written with other languages.

² All the time in this column is zero because there is no the first phase test in conventional testing.

7 Conclusions

This paper presents JMLAutoTest, a novel testing framework designed for Java programs annotated with JML specifications.

JMLAutoTest automatically generate three classes for a target method. In the test client, testers can generate test cases for any kinds of types including linked data structures and common types in either a fast way or a normal way very easily. JMLAutoTest verifies the validity of a candidate by checking its invariant with JML runtime assertion checker.

JMLAutoTest provides a double-phase testing way for the test of a method. It is the statistic based testing which filters out meaningless test cases without requiring testers to know the details of the method to be tested. According to the operational profile made by the tester, the generated test case space can be divided into several partitions. During the first phase, a small number of test cases are taken out from each partition. Then the test suite is run for several times to record the number of meaningless cases of each group. Based on statistical principles, we should estimate the approximate proportion of the meaningless test cases in each partition. During the second phase, test cases taken out from each partition according to these calculated proportions are mixed together and supplied to the test. Time spent visiting meaningless test cases in the final test is saved.

References

1. Y.Cheon. A Runtime Assertion Checker for the Java Modeling Language. Technical Report 03-09, Department of Computer Science, Iowa State University, Apr. 2003.
2. Y.Cheon and G.T.Leavens. A simple and practical approach to unit testing: The JML and JUnit way. Technical Report 01-12, Department of Computer Science, Iowa State University, Nov, 2001.
3. G.T.Leavens, A.L.Baker, and C.Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, June 1998. (last revision: Aug 2001).
4. C. Boyapati, S. Khurshid and D. Marinov. Korat: Automated Testing Based on Java Predicates. In *Proc. ACM International Symposium on Software Testing and Analysis (ISSTA 2002)*, pages 123-133., July 2002.
5. M.Deck and J.A.Whittaker. Lessons learned from fifteen years of Cleanroom Testing. *Software Testing, Analysis, and Review (STAR)* 97, May 5-9, 1997.
6. G.T.Leavens, A.L. Baker and C. Ruby. JML: A notation for detailed design. In *Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, Behavioral Specifications of Businesses and Systems*, chapter 12, pages 175-188. Kluwer, 1999.
7. K. Bech and E. Gamma. Tested infected: Programmers love writing tests. *Java Report*, 3(7), July 1998.
8. J. Goodenough and S. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, June 1975.
9. J.Offutt and A. Abdurazik. Generating tests from UML specifications. In *Proc. Second International Conference on the Unified Modeling Language*, Oct. 1999.

10. H. M. Horcker. Improving software tests using Z specifications. In *Proc. 9th International Conference of Z users, The Z Formal Specification Notation*, 1995.
11. D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proc, 16th IEEE Engineering (ASE)*, San Diego, CA, Nov, 2001.
12. D. Peters and D. L. Parnas. Generating a test oracle from program documentation. In *Proc, ISSTA 94*, Seattle, Washington, Aug. 1994.
13. D. J. Richardson. TAOS: Testing with analysis and oracle support. In *Proc, ISSTA 94*, Seattle, Washington, August, 1994.
14. P. Stocks and D. Carrington. Test template framework: A specification-based test case study. In *Proc, the 1993 International Symposium on Software Testing and Analysis (ISSTA)*, pages 11–18, Jun. 1993.
15. J. D. Musa. Operational Profiles in Software-Reliability Engineering. *IEEE Software*, pages 14-32, Mar.1993.
16. NIST/SEMATECH E-Handbook of Statistical Methods.
<http://www.itl.nist.gov/div898/handbook/>, May, 2003.
17. JUnit. <http://www.junit.org>.
18. R. Chillarege. Software testing best practice. Technical Report RC 21457, Center for Software Engineering, IBM Research, Apr.1999.
19. D. Banks, W. Dashiell, L. Gallagher, C. Hagwood, R. Kacker and L. Rosenthal. Software testing based on statistical methods. National Institute of Standards and Technology Information Technology Laboratory, Gaithersburg, MD, Mar.1998.
20. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.
21. J. Rumbaugh, I. Jacobson, and G. Booch. The Unified Modeling Language Reference Manual. Addison-Wesley Object Technology Series, 1998.
22. D. Jackson, I. Shlyakhter, and Il Shlyakhter. ALCOA: The Alloy constraint analyzer. In *Proc. 22nd International Conference on Software Engineering (ICSE)*, Limerick, Ireland, June 2000.

Compositional Testing with IOCO

Machiel van der Bijl^{1,*}, Arend Rensink¹, and Jan Tretmans²

¹ Software Engineering, Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands
{vdbijl, rensink}@cs.utwente.nl

² Software Technology Research Group, University of Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
tretmans@cs.kun.nl

Abstract. *Compositional testing* concerns the testing of systems that consist of communicating components which can also be tested in isolation. Examples are component based testing and interoperability testing. We show that, with certain restrictions, the *ioco*-test theory for conformance testing is suitable for compositional testing, in the sense that the integration of fully conformant components is guaranteed to be correct. As a consequence, there is no need to re-test the integrated system for conformance.

This result is also relevant for *testing in context*, since it implies that every failure of a system embedded in a test context can be reduced to a fault of the system itself.

1 Introduction

In this paper we study formal testing based on the *ioco*-test theory. This theory works on labeled transition systems (LTS) [1, 2]. The name *ioco*, which stands for *input/output conformance*, refers to the implementation relation (i.e., notion of correctness) on which the theory and the test generation algorithm have been built. A number of tools are based on the *ioco* theory, among which there are TGV [3], TESTGEN [4] and TorX [5].

Two open issues in testing theory in general, and the *ioco*-theory in particular, are *compositional testing* and *testing in context*. For instance, for the testing theory based on Finite-State-Machines (FSM) this issue has been studied in [6].

Compositional testing considers the testing of communicating components that together form a larger system. An example is component based testing, i.e., integration testing of components that have already been tested separately. An example from the telecom sector is interoperability testing, i.e., testing if systems from different manufacturers, that should comply with a certain standard, work together; for example GSM mobile phones. The question is what can be concluded from the individual tests of the separate components, and what should be (re)tested on the integration or system level. With the current theory it is unclear what the relation between the correctness of the components and the integrated system is.

* This research was supported by Ordina Finance and by the dutch research programme PROGRESS under project: TES5417: Atomyste – ATOM splitting in eMbedded sYSTEMs TESting.

Another scenario, with similar characteristics, is *testing in context*. This refers to the situation that a tester can only access the implementation under test through a *test context* [7–9]. The test context interfaces between the implementation under test and the tester. As a consequence the tester can only indirectly observe and control the IUT via the test context. This makes testing weaker, in the sense that there are fewer possibilities for observation and control of the IUT. With testing in context, the question is whether faults in the IUT can be detected by testing the composition of IUT and test context, and whether a failure of this composition always indicates a fault of the IUT. This question is the converse of compositional testing: when testing in context we wish to detect errors in the IUT — a component — by testing it in composition with the test context, whereas in compositional testing we wish to infer correctness of the integrated system from conformance of the individual components.

This paper studies the above mentioned compositionality properties of *ioco* for two operations on labeled transition systems: parallel composition and hiding. If *ioco* has this compositionality property for these operations, it follows that correctness of the parts (the components) implies correctness of the whole (the integrated system), or that a fault in the whole (IUT and test context) implies a fault in the component (IUT). This compositionality property is formally called a pre-congruence.

We show that *ioco* is a pre-congruence for parallel composition and hiding in the absence of underspecification of input actions. One way to satisfy this condition is to only allow specifications which are *input enabled*. Another way is to make the underspecification explicit by *completion*. We show that, in particular, *demonic completion* is suitable for this purpose. As a final result we show how to use the original (uncompleted) specifications and still satisfy the pre-congruence property. This leads to a new implementation relation, baptized *ioco_U* which is slightly weaker than *ioco*.

This paper has two main results. First we show a way to handle underspecification of input actions when testing communicating components with the *ioco* theory. This idea is new for LTS testing. It is inspired by [10] and similar work done in FSM testing [11]. Second we establish a formal relation between the components and the integrated system. As far as we know this result is new for both LTS testing and FSM testing.

Overview. The next section recalls some basic concepts and definitions about transition systems and *ioco*. Section 3 sets the scene and formalizes the problems of compositional testing and testing in context. Section 4 studies the pre-congruence properties of *ioco* for parallel composition and hiding. Section 5 discusses underspecification, and approaches to complete specifications with implicit underspecification. Section 6 concludes with some final remarks and an assessment of the results. For a full version of this paper with all the proofs, we refer to [12].

2 Formal Preliminaries

This section recalls the aspects of the theory behind *ioco* that are used in this paper; see [1] for a more detailed exposition.

Labeled Transition Systems. A labeled transition system (LTS) description is defined in terms of states and labeled transitions between states, where the labels indicate what

happens during the transition. Labels are taken from a global set $\mathbf{L}$. We use a special label $\tau \notin \mathbf{L}$ to denote an internal action. For arbitrary $L \subseteq \mathbf{L}$, we use L_τ as a shorthand for $L \cup \{\tau\}$. We deviate from the standard definition of labeled transition systems in that we assume the label set of an LTS to be partitioned in an input and an output set.

Definition 1. A labeled transition system is a 5-tuple $\langle Q, I, U, T, q_0 \rangle$ where Q is a non-empty countable set of states; $I \subseteq \mathbf{L}$ is the countable set of input labels; $U \subseteq \mathbf{L}$ is the countable set of output labels, which is disjoint from I ; $T \subseteq Q \times (I \cup U \cup \{\tau\}) \times Q$ is a set of triples, the transition relation; $q_0 \in Q$ is the initial state.

We use L as shorthand for the entire label set ($L = I \cup U$); furthermore, we use Q_p, I_p etc. to denote the components of an LTS p . We commonly write $q \xrightarrow{\lambda} q'$ for $(q, \lambda, q') \in T$. Since the distinction between inputs and outputs is important, we sometimes use a question mark before a label to denote input and an exclamation mark to denote output. We denote the class of all labeled transition systems over I and U by $\mathcal{LTS}(I, U)$. We represent a labeled transition system in the standard way, by a directed, edge-labeled graph where nodes represent states and edges represent transitions.

A state that cannot do an internal action is called *stable*. A state that cannot do an output or internal action is called *quiescent*. We use the symbol δ ($\notin \mathbf{L}_\tau$) to represent quiescence: that is, $p \xrightarrow{\delta} p$ stands for the absence of any transition $p \xrightarrow{\lambda} p'$ with $\lambda \in U_\tau$. For an arbitrary $L \subseteq \mathbf{L}_\tau$, we use L_δ as a shorthand for $L \cup \{\delta\}$.

An LTS is called *strongly responsive* if it always eventually enters a quiescent state; in other words, if it does not have any infinite U_τ -labeled paths. For technical reasons we restrict $\mathcal{LTS}(I, U)$ to strongly responsive transition systems. Systems that are not strongly responsive may show live-locks (or develop live-locks by hiding actions). So one can argue that it is a favorable property if a specification is strongly responsive. However, from a practical perspective it would be nice if the constraint can be lessened. This is probably possible, but needs further research.

A *trace* is a finite sequence of observable actions. The set of all traces over L ($\subseteq \mathbf{L}$) is denoted by L^* , ranged over by σ , with ϵ denoting the empty sequence. If $\sigma_1, \sigma_2 \in L^*$, then $\sigma_1 \cdot \sigma_2$ is the concatenation of σ_1 and σ_2 . We use the standard notation with single and double arrows for traces: $q \xrightarrow{a_1 \dots a_n} q'$ denotes $q \xrightarrow{a_1} \dots \xrightarrow{a_n} q'$, $q \xRightarrow{\epsilon} q'$ denotes $q \xrightarrow{\tau \dots \tau} q'$ and $q \xRightarrow{a_1 \dots a_n} q$ denotes $q \xRightarrow{\epsilon} \xrightarrow{a_1} \xRightarrow{\epsilon} \dots \xrightarrow{a_n} \xRightarrow{\epsilon} q'$ (where $a_i \in \mathbf{L}_{\tau\delta}$).

We will not always distinguish between a labeled transition system and its initial state. We will identify the process $p = \langle Q, I, U, T, q_0 \rangle$ with its initial state q_0 , and we write, for example, $p \xRightarrow{\sigma} q_1$ instead of $q_0 \xRightarrow{\sigma} q_1$.

Input-Output Transition Systems. An *input-output transition system* (IOTS) is a labeled transition system that is completely specified for input actions. The class of input-output transition systems with input actions in I and output actions in U is denoted by $\mathcal{IOTS}(I, U)$ ($\subseteq \mathcal{LTS}(I, U)$). Notice that we do not require IOTS's to be strongly responsive.

Definition 2. An input-output transition system $p = \langle Q, I, U, T, q_0 \rangle$ is a labeled transition system for which all inputs are enabled in all states: $\forall q \in Q, a \in I : q \xRightarrow{a}$

Composition of Labeled Transition Systems. The integration of components can be modeled algebraically by putting the components in parallel while synchronizing their

common actions, possibly with internalizing (hiding) the synchronized actions. In process algebra, the synchronization and internalization are typically regarded as two separate operations. The synchronization of the processes p and q is denoted by $p \parallel q$. The internalization of a label set V in process p , or *hiding* V in p as it is commonly called, is denoted by **hide** V **in** p . Below we give the formal definition.

Definition 3. For $i = 1, 2$ let $p_i = \langle Q_i, I_i, U_i, T_i, p_i \rangle$ be a transition system.

- If $I_1 \cap I_2 = U_1 \cap U_2 = \emptyset$ then $p_1 \parallel p_2 =_{\text{def}} \langle Q, I, U, T, p_1 \parallel p_2 \rangle$ where
 - $Q = \{q_1 \parallel q_2 \mid q_1 \in Q_1, q_2 \in Q_2\}$;
 - $I = (I_1 \setminus U_2) \cup (I_2 \setminus U_1)$;
 - $U = U_1 \cup U_2$.
 - T is the minimal set satisfying the following inference rules ($\mu \in L_\tau$):

$$\begin{array}{lcl} q_1 \xrightarrow{\mu} q'_1, \mu \notin L_2 & & \vdash q_1 \parallel q_2 \xrightarrow{\mu} q'_1 \parallel q_2 \\ q_2 \xrightarrow{\mu} q'_2, \mu \notin L_1 & & \vdash q_1 \parallel q_2 \xrightarrow{\mu} q_1 \parallel q'_2 \\ q_1 \xrightarrow{\mu} q'_1, q_2 \xrightarrow{\mu} q'_2, \mu \neq \tau & \vdash & q_1 \parallel q_2 \xrightarrow{\mu} q'_1 \parallel q'_2 \end{array}$$

- If $V \subseteq U_1$, then **hide** V **in** $p_1 =_{\text{def}} \langle Q, I_1, U_1 \setminus V, T, \text{hide } V \text{ in } p_1 \rangle$ where
 - $Q = \{\text{hide } V \text{ in } q_1 \mid q_1 \in Q_1\}$;
 - T is the minimal set satisfying the following inference rules ($\mu \in L_\tau$):

$$\begin{array}{lcl} q_1 \xrightarrow{\mu} q'_1, \mu \notin V & \vdash & \text{hide } V \text{ in } q_1 \xrightarrow{\mu} \text{hide } V \text{ in } q'_1 \\ q_1 \xrightarrow{\mu} q'_1, \mu \in V & \vdash & \text{hide } V \text{ in } q_1 \xrightarrow{\tau} \text{hide } V \text{ in } q'_1 \end{array}$$

Note that these constructions are only *partial*: there are constraints on the input and output sets. Moreover, parallel composition may give rise to an LTS that is not strongly responsive, even if the components are. For the time being, we do not try to analyze this but implicitly restrict ourselves to cases where the parallel composition is strongly responsive (thus, this is another source of partiality of the construction).

In this paper we restrict ourselves to binary parallel composition. N-ary parallel composition may be an interesting extension. One may wonder however what this means in our input output setting, since an output action is uniquely identified by its sender. From this perspective only the synchronization of many receivers to one sender (broadcast) seems an interesting extension.

Proposition 1. Let $p, q \in \mathcal{LTS}(I_i, U_i)$ for $i = p, q$, with $I_p \cap I_q = U_p \cap U_q = \emptyset$, and let $V \subseteq U_p$.

1. If $p \parallel q$ is strongly responsive then $p \parallel q \in \mathcal{LTS}((I_p \setminus U_q) \cup (I_q \setminus U_p), U_p \cup U_q)$; moreover, $p \parallel q \in \mathcal{IOTS}$ if $p, q \in \mathcal{IOTS}$.
2. **hide** V **in** $p \in \mathcal{LTS}(I_p, U_p \setminus V)$; moreover, **hide** V **in** $p \in \mathcal{IOTS}$ if $p \in \mathcal{IOTS}$.

Conformance. The testing scenario on which **io**co is based assumes that two things are given: 1) An LTS constituting a specification of required behavior. And 2) an implementation under test. We treat the IUT as a black box. In order to reason about it we assume it can be modeled as an IOTS (an IUT is an object in the real world). This assumption is referred to as the test hypothesis [7]. We want to stress that we do not need to *have* this model when testing the IUT. We only *assume* that the implementation *behaves* as an IOTS.

Given a specification s and an (assumed) model of the IUT i , the relation $i \text{ ioco } s$ expresses that i conforms to s . Whether this holds is decided on the basis of the *suspension traces* of s : it must be the case that, after any such trace σ , every output action (and also quiescence) that i is capable of should be allowed according to s . This is formalized by defining $p \text{ after } \sigma$ (the set of states that can be reached in p after the suspension trace σ), $\text{out}(p)$ (the set of output and δ -actions of p) and $\text{Straces}(p)$ (the suspension traces of p).

Definition 4. Let $p, s \in \mathcal{LTS}(I, U)$, let $i \in \mathcal{IOTS}(I, U)$, let $P \subseteq Q_p$ be a set of states in p and let $\sigma \in L_\delta^*$.

1. $p \text{ after } \sigma =_{\text{def}} \{ p' \mid p \xrightarrow{\sigma} p' \}$
2. $\text{out}(p) =_{\text{def}} \{ x \in U \mid p \xrightarrow{x} \} \cup \{ \delta \mid p \xrightarrow{\delta} \}$
3. $\text{out}(P) =_{\text{def}} \bigcup \{ \text{out}(p) \mid p \in P \}$
4. $\text{Straces}(p) =_{\text{def}} \{ \sigma \in L_\delta^* \mid p \xrightarrow{\sigma} \}$

The following defines the implementation relation **ioco**, modulo a function $\mathcal{F}$ that generates a set of test-traces from a specification. In this definition 2^X denotes the powerset of X , for an arbitrary set X .

Definition 5. Given a function $\mathcal{F} : \mathcal{LTS}(I, U) \rightarrow 2^{L_\delta^*}$, we define the implementation relation $\text{ioco}_\mathcal{F} \subseteq \mathcal{IOTS}(I, U) \times \mathcal{LTS}(I, U)$ as follows:

$$i \text{ ioco}_\mathcal{F} s \iff \forall \sigma \in \mathcal{F}(s) : \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma)$$

So $i \text{ ioco}_{\text{Straces}} s$ means $\forall \sigma \in \text{Straces}(s) : \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma)$. We use **ioco** as an abbreviation for $\text{ioco}_{\text{Straces}}$. For more detailed information about **ioco** we refer to [1].

3 Approach

In this section we want to clarify compositional testing with the formal framework presented in the previous section. The consequences for testing in context will be discussed in the final section.

We study systems that consist of communicating components. These components can be tested individually and while working together (in the case of testing in context the components are the IUT and its test context). The behavior of such a system is described by the parallel composition of the individual transition systems. Output actions of one component that are in the input label set of another component are *synchronized*, resulting in a single, internal transition of the overall system. Actions of a component that are not in the label set of another component are not synchronized, resulting in a single observable transition of the overall system. This gives rise to the scenario depicted in Figure 1. The figure will be explained in the next example.

3.1 Example

To illustrate compositional testing, we use two components of a coffee machine: a “money component” (*mon*) that handles the inserted coins and a “drink component” (*drk*) that takes care of preparing and pouring the drinks, see Figure 1.

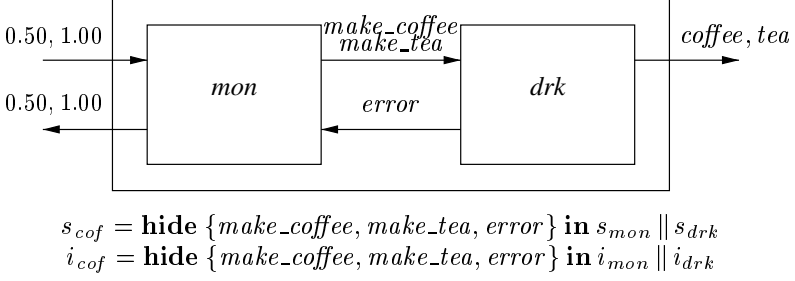


Fig. 1. Architecture of coffee machine in components

The money component accepts coins of €1 and of €0.50 as input from the environment. After insertion of a €0.50 coin (respectively €1 coin), the money component orders the drink component to make tea (respectively coffee).

The drink component interfaces with the money component and the environment. If the money component orders it to make tea (respectively coffee) it outputs tea (respectively coffee) to the environment. If anything goes wrong in the drink making process, the component gives an error signal.

The coffee machine is the parallel composition of the money component and the drink component, in which the “make coffee” command, the “make tea” command and the “error” signal are hidden. One can think of the parallel composition as establishing the connection between the money component and the drink component, whereas hiding means that the communication between the components is not observable anymore; only communication with the environment can be observed.

Models. In Figure 2 we show the behavioral specification of the money component s_{mon} and the drink component s_{drk} as LTS’s. Note that the money component is underspecified for the *error* input label and that the drink component cannot recover from an error state, and while in the error state it cannot produce tea or coffee. Figure 3 shows implementation models of the money component, i_{mon} , and the drink component, i_{drk} . We have used transitions labeled with ‘?’ as an abbreviation for all the non-specified input actions from the alphabet of the component. The money component has input label set, $I_{mon} = \{0.50, 1.00, error\}$, output label set $U_{mon} = \{make_coffee, make_tea, 0.50, 1.00\}$; specification $s_{mon} \in \mathcal{LTS}(I_{mon}, U_{mon})$, and implementation $i_{mon} \in \mathcal{IOTS}(I_{mon}, U_{mon})$. $I_{drk} = \{make_coffee, make_tea\}$ and $U_{drk} = \{coffee, tea, error\}$ are the input and output label set respectively and $s_{drk} \in \mathcal{LTS}(I_{drk}, U_{drk})$, $i_{drk} \in \mathcal{IOTS}(I_{drk}, U_{drk})$.

In the implementations of the components we choose to improve upon the specification, by adding functionality. This is possible since *ioco* allows partial specifications. Implementers are free to make use of the underspecification. The extra functionality of i_{mon} compared to its specification s_{mon} is that it can handle error signals: it reacts by returning €1.00. i_{drk} is also changed with respect to its specification s_{drk} : making tea never produces an error signal. Since implementations are input enabled, we have chosen that all non specified inputs are ignored, i.e., the system remains in the same state.

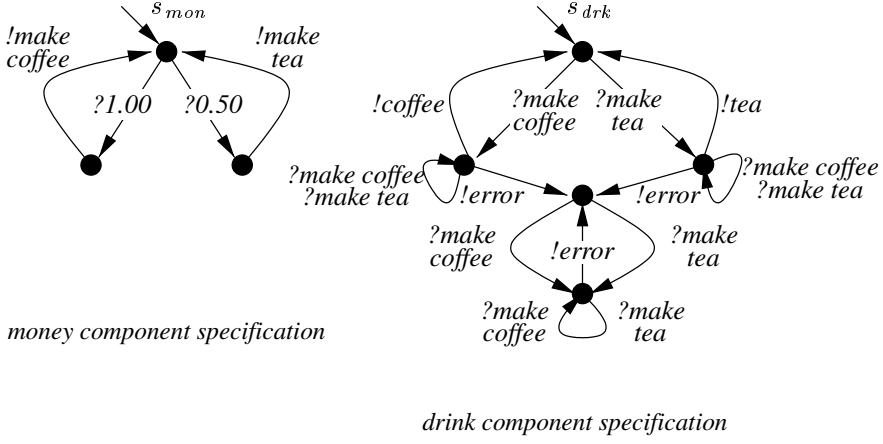


Fig. 2. Specification of money and drink components as LTS's

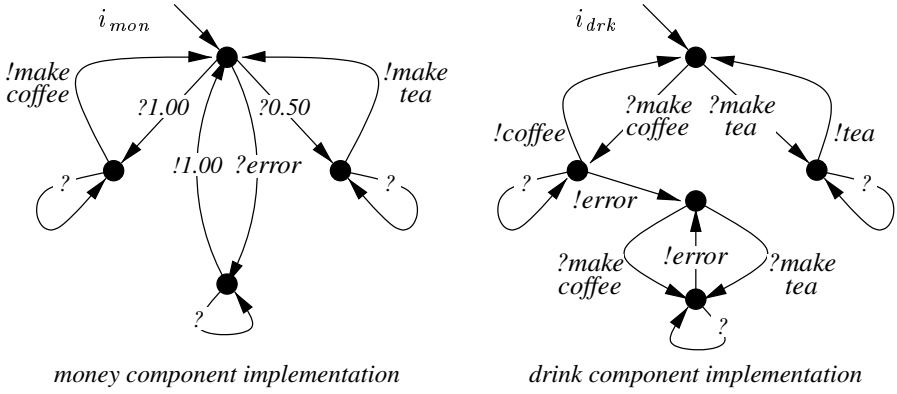


Fig. 3. Implementation of the money and drink components as IOTS's

We have $i_{mon} \text{ ioco } s_{mon}$ and $i_{drk} \text{ ioco } s_{drk}$. The question now is whether the integrated implementation, as given by i_{cof} in Figure 1, is also **ioco** correct with respect to the integrated specification s_{cof} . We discuss this in section 4, to illustrate the compositionality properties discussed there.

3.2 Compositional Testing

We now paraphrase the question of compositional testing, discussed in the introduction, as follows: “Given that the components p and q have been tested to be **ioco**-correct (according to their respective specifications), may we conclude that their integration is also **ioco**-correct (according to the integrated specification)?” If the component specifications are LTS's, the component implementations are modeled by IOTS's, and their

integration by parallel composition followed by hiding, this boils down to the following questions in our formal framework (where $i_k \in \mathcal{IOTS}(I_k, U_k)$ and $s_k \in \mathcal{LTS}(I_k, U_k)$ for $k = 1, 2$, with $I_1 \cap I_2 = U_1 \cap U_2 = \emptyset$):

Q1: Given $i_k \text{ ioco } s_k$ for $k = 1, 2$, is it the case that $i_1 \parallel i_2 \text{ ioco } s_1 \parallel s_2$?

Q2: Given $i_1 \text{ ioco } s_1$, is it the case that $(\text{hide } V \text{ in } i_1) \text{ ioco } (\text{hide } V \text{ in } s_1)$ for arbitrary $V \subseteq U_1$?

If the answer to both questions is “yes”, then we may conclude that **ioco** is suitable for compositional testing as stated in the following corollary.

Conjecture 1. If $i_k \in \mathcal{IOTS}(I_k, U_k)$ and $s_k \in \mathcal{LTS}(I_k, U_k)$ for $k = 1, 2$ with $I_1 \cap I_2 = U_1 \cap U_2 = \emptyset$ and $V = (I_1 \cap U_2) \cup (U_1 \cap I_2)$, then

$$i_1 \text{ ioco } s_1 \wedge i_2 \text{ ioco } s_2 \Rightarrow (\text{hide } V \text{ in } i_1 \parallel i_2) \text{ ioco } (\text{hide } V \text{ in } s_1 \parallel s_2) .$$

We study the above pre-congruence questions in the next section. We will show that the answer to Q1 and Q2 in general is *no*. Instead, we can show that the answer to Q1 and Q2 is *yes* if s_1 and s_2 are completely specified.

4 Compositionality for Synchronization and Hiding

In this section we address the questions Q1 and Q2 formulated above (Section 3.2), using the coffee machine example to illustrate our results.

4.1 Synchronization

The property that we investigate for parallel composition is: *if* we have two correct component implementations according to **ioco**, *then* the implementation remains correct after synchronizing the components. It turns out that in general this property does not hold, as we show in the following example.

Example 1. Regard the LTS's in figure 4. On the left hand side we show the specifications and on the right hand side the corresponding implementations. The models have the following label sets: $s_1 \in \mathcal{LTS}(\{x\}, \emptyset)$, $i_1 \in \mathcal{IOTS}(\{x\}, \emptyset)$, $s_2 \in \mathcal{LTS}(\emptyset, \{x\})$ and $i_2 \in \mathcal{IOTS}(\emptyset, \{x\})$. The suspension traces of s_1 are given by $\delta^* \cup \delta^*?x\delta^*$ and the suspension traces of s_2 are given by $\{\epsilon, !x\} \cup !x!\delta^*$. We have $i_1 \text{ ioco } s_1$ and $i_2 \text{ ioco } s_2$.

After we take the parallel composition of the two specifications we get $s_1 \parallel s_2$, see figure 4 (the corresponding implementation is $i_1 \parallel i_2$). We now see the following: $\text{out}(i_1 \parallel i_2 \text{ after } !x) = \{!x\} \not\subseteq \text{out}(s_1 \parallel s_2 \text{ after } !x) = \{\delta\}$; this means that the parallel composition of the implementations is not **ioco**-correct: $i_1 \parallel i_2 \not\text{ ioco } s_1 \parallel s_2$.

Analysis shows that $i_1 \text{ ioco } s_1$, because **ioco** allows underspecification of input actions. However, the semantics of the parallel composition operator does not take underspecification of input actions into account. Although s_2 can output a second x , it cannot do so in $s_1 \parallel s_2$, because s_1 cannot input the second x .

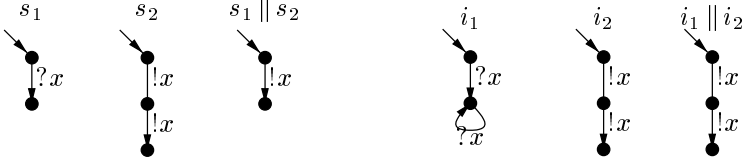


Fig. 4. Counter example to compositionality for parallel composition; see Example 1

It turns out that if we forbid implicit underspecification, i.e., if the specification explicitly prescribes for any possible input what the allowed responses are, then we do not have this problem. In fact in that case we have the desired compositionality property. This property is expressed in the following theorem. For a proof see [12].

Theorem 1. *Let $s_1, i_1 \in \mathcal{IOTS}(I_1, U_1)$, $s_2, i_2 \in \mathcal{IOTS}(I_2, U_2)$, with $I_1 \cap I_2 = U_1 \cap U_2 = \emptyset$.*

$$i_1 \text{ ioco } s_1 \wedge i_2 \text{ ioco } s_2 \Rightarrow i_1 \parallel i_2 \text{ ioco } s_1 \parallel s_2$$

Our running example (Section 3.1) shows the same problem illustrated in example 1. Although the implementations of the money component and the drink component are **ioco** correct with respect to their specifications, it turns out that the parallel composition of i_{mon} and i_{drk} is not:

$$\begin{aligned} \text{out}(i_{mon} \parallel i_{drk} \text{ after } ?1.00.!make_coffee) &= \{!coffee, !error\} \\ \text{out}(s_{mon} \parallel s_{drk} \text{ after } ?1.00.!make_coffee) &= \{!coffee\} \end{aligned}$$

Note that the internal signals are still visible as output actions. To turn them into internal actions is the task of the *hiding* operator, discussed below.

4.2 Hiding

The property that we investigate for hiding is the following: *if we have a correct implementation according to **ioco**, then the implementation remains correct after hiding (some of the) output actions*. It turns out that, as for synchronization, in general this property does not hold.

Example 2. Consider the implementation i and specification s in Figure 5, both with input set $\{a\}$ and output set $\{x, y\}$. The suspension traces of s are $\{\epsilon\} \cup ?a\delta^* \cup !x\delta^*$. We see that $i \text{ ioco } s$.

We get the specification **hide** $\{x\}$ **in** s , and implementation **hide** $\{x\}$ **in** i after hiding the output action x . After the input a we get: $\text{out}(\text{hide } \{x\} \text{ in } i \text{ after } a) = \{\delta, y\} \not\subseteq \text{out}(\text{hide } \{x\} \text{ in } s \text{ after } a) = \{\delta\}$; in other words the **ioco** relation does not hold: **hide** $\{x\}$ **in** $i \text{ ioco } \text{hide } \{x\}$ **in** s .

An analysis of the above example shows that s was underspecified, in the sense that it fails to prescribe how an implementation should behave after the trace $!x?a$. The proposed implementation i uses the implementation freedom by having an unspecified y -output after $!x?a$. However, if x becomes unobservable due to hiding, then

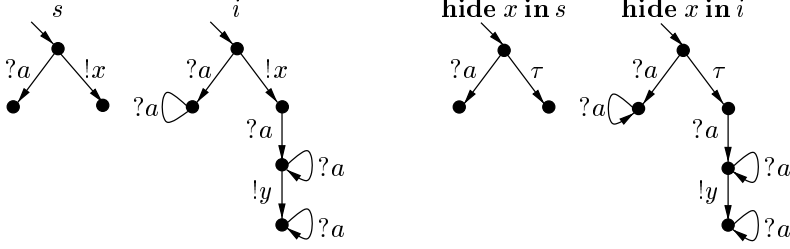


Fig. 5. Counter-example to compositionality for hiding; see Example 2

the traces $!x?a$ and $?a$ collapse and become indistinguishable: in **hide** $\{x\}$ in s and **hide** $\{x\}$ in i they both masquerade as the trace $?a$. Now **hide** $\{x\}$ in s appears to specify that after $?a$, only quiescence (δ) is allowed; however, **hide** $\{x\}$ in i still has this unspecified y -output. In other words, hiding creates confusion about what part of the system is underspecified.

It follows that if we rule out underspecification, i.e., we limit ourselves to specifications that are IOTS's then this problem disappears. In fact, in that case we do have the desired congruence property. This is stated in the following theorem. For a proof see [12].

Theorem 2. *If $i, s \in \mathcal{IOTS}(I, U)$ with $V \subseteq U$, then:*

$$i \text{ ioco } s \Rightarrow (\text{hide } V \text{ in } i) \text{ ioco } (\text{hide } V \text{ in } s)$$

5 Demonic Completion

We have shown in the previous section that **ioco** is a pre-congruence for parallel composition and hiding when restricted to $\mathcal{IOTS} \times \mathcal{IOTS}$. However, in the original theory [1] $\text{ioco} \subseteq \mathcal{IOTS} \times \mathcal{LTS}$; the specifications are LTS's. The intuition behind this is that **ioco** allows underspecification of input actions. In this section we present a function that transforms LTS's into IOTS's in a way that complies with this notion of underspecification. We will show that this leads to a new implementation relation that is slightly weaker than **ioco**.

Underspecification comes in two flavors: underspecification of input actions and underspecification of output actions. Underspecification of output actions is always explicit; in an LTS it is represented by a choice between several output actions. The intuition behind this is that we do not know or care which of the output actions is implemented, as long as at least one is. Underspecification of input actions is always implicit; it is represented by absence of the respective input action in the LTS. The intuition behind underspecification of input actions is that after an unspecified input action we do not know or care what the behavior of the specified system is. This means that in an underspecified state — i.e., a state reached after an unspecified input action — every action from the label set is correct, including quiescence. Following [13] we call this kind of behavior *chaotic*.

In translating LTS's to IOTS's, we propose to model underspecification of input actions explicitly. Firstly, we model chaotic behavior through a state q_χ with the property: $\forall \lambda \in U : q_\chi \xrightarrow{\lambda} q_\chi$ and $\forall \lambda \in I : q_\chi \xrightarrow{\delta^*, \lambda} q_\chi$ (where χ stands for chaos). Secondly, we add for every stable state q (of a given LTS) that is underspecified for an input a , a transition (q, a, q_χ) . This turns the LTS into an IOTS. After [10] we call this procedure *demonic completion* — as opposed to *angelic completion*, where unspecified inputs are discarded (modeled by adding self-loop transitions). Note that demonic completion results in an IOTS that is not strongly convergent. However the constraint of strong convergence only holds for LTS's.

Definition 6.

$\Xi : \mathcal{LTS}(I, U) \rightarrow \mathcal{IOTS}(I, U)$ is defined by $\langle Q, I, U, T, q_0 \rangle \mapsto \langle Q', I, U, T', q_0 \rangle$, where

$$\begin{aligned} Q' &= Q \cup \{q_\chi, q_\Omega, q_\Delta\}, \text{ where } q_\chi, q_\Omega, q_\Delta \notin Q \\ T' &= T \cup \{(q, a, q_\chi) \mid q \in Q, a \in I, q \not\xrightarrow{a}, q \not\xrightarrow{\tau}\} \\ &\quad \cup \{(q_\chi, \tau, q_\Omega), (q_\chi, \tau, q_\Delta)\} \cup \{(q_\Omega, \lambda, q_\chi) \mid \lambda \in L\} \cup \{(q_\Delta, \lambda, q_\chi) \mid \lambda \in I\} \end{aligned}$$

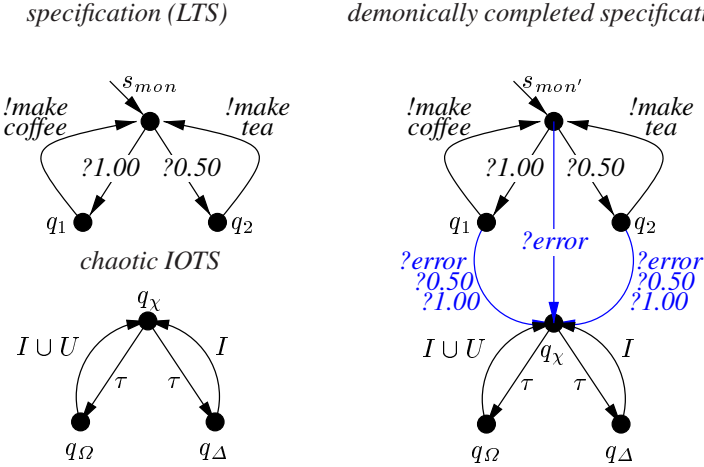


Fig. 6. Demonic completion of an LTS specification

Example 3. To illustrate the demonic completion of implicit underspecification, we use the money component of section 3.1. The LTS specification of the money component is given in the top left corner of Figure 6. The IOTS that models our chaos property is given in the bottom left corner. For every stable state of the specification that is underspecified for an input action, the function Ξ adds a transition with that input action to state q_χ . For example, every state is underspecified for input action *error*, so we add a transition from every state to q_χ for *error*. The states q_1 and q_2 are underspecified for 0.50 and 1.00, so we add transitions for these inputs from q_1 and q_2 to q_χ . The resulting demonically completed specification is given on the right hand side of Figure 6.

An important property of demonic completion is that it only adds transitions from *stable* states with underspecified inputs in the original LTS to q_χ . Moreover, it does not delete states or transitions. Furthermore, the chaotic IOTS acts as a kind of sink: once one of the added states (q_χ , q_Ω or q_Δ) has been reached, they will never be left anymore.

Proposition 2. *Let $s \in \mathcal{LTS}(I, U)$. $\forall \sigma \in L_\delta^*$, $q' \in Q_s : s \xrightarrow{\sigma} q' \Leftrightarrow \Xi(s) \xrightarrow{\sigma} q'$*

We use the notation “ $\mathbf{ioco} \circ \Xi$ ” to denote that before applying $\mathbf{ioco}$, the LTS specification is transformed to an IOTS by Ξ ; i.e., $i(\mathbf{ioco} \circ \Xi)s \Leftrightarrow i \mathbf{ioco} \Xi(s)$. This relation is slightly weaker than $\mathbf{ioco}$. This means that previously conformant implementations are still conformant, but it might be that previously non-allowed implementations are allowed with this new notion of conformance.

Theorem 3. $\mathbf{ioco} \subseteq \mathbf{ioco} \circ \Xi$

Note that the opposite is not true i.e., $i(\mathbf{ioco} \circ \Xi)s \not\Leftarrow i \mathbf{ioco} s$ (as the counterexamples of section 4 show). Furthermore this property is a consequence of our choice of the demonic completion function. Other forms of completion, such as angelic completion, result in variants of $\mathbf{ioco}$ which are incomparable to the original relation.

Testing. The testing scenario is now such that an integrated system can be tested by comparing the individual components to their *demonically completed* specifications. If the components conform, then the composition of implementations also conforms to the composition of the demonically completed specifications.

Corollary 1. *Let $s_1, s_2 \in \mathcal{LTS}(I, U)$ and $i_1, i_2 \in \mathcal{IOTS}(I, U)$*

$$i_1 \mathbf{ioco} \Xi(s_1) \wedge i_2 \mathbf{ioco} \Xi(s_2) \Rightarrow i_1 \parallel i_2 \mathbf{ioco} \Xi(s_1) \parallel \Xi(s_2)$$

Test Restriction. A disadvantage of demonic completion is that it destroys information about underspecified behavior. On the basis of the underspecified LTS, one can conclude that traces including an unspecified input need not be tested because every implementation will always pass; after completion, however, this is no longer visible, and so automatic test generation will yield many spurious tests.

In order to avoid this, we characterize $\mathbf{ioco} \circ \Xi$ directly over LTS's. In other words, we extend the relation from $\mathcal{IOTS} \times \mathcal{IOTS}$ to $\mathcal{IOTS} \times \mathcal{LTS}$, in such a way as to obtain the same testing power but to avoid these spurious tests. For this purpose, we restrict the number of traces after which we test.

Definition 7. *Let $s \in \mathcal{LTS}(I, U)$.*

$$Utraces(s) =_{\text{def}} \{ \sigma \in L_\delta^* \mid s \xrightarrow{\sigma} \wedge (\nexists q', \sigma_1 \cdot a \cdot \sigma_2 = \sigma : a \in I \wedge s \xrightarrow{\sigma_1} q' \wedge q' \not\xrightarrow{a}) \}$$

Intuitively, the *Utraces* are the *Straces* without the underspecified traces. A trace σ is underspecified if there exists a prefix $\sigma_1 \cdot a$ of σ , with $a \in I$, for which $s \xrightarrow{\sigma_1} q'$ and $q' \not\xrightarrow{a}$. We use $\mathbf{ioco}_U$ as a shorthand for $\mathbf{ioco}_{Utraces}$. In the following proposition we state that $\mathbf{ioco}_U$ is equivalent to $\mathbf{ioco} \circ \Xi$. This equivalence is quite intuitive. $\mathbf{ioco} \circ \Xi$ uses extra states to handle underspecified behavior, which are constructed so as to display chaotic behavior. If $\Xi(s)$ reaches such a state, then all behavior is considered correct. $\mathbf{ioco}_U$, on the other hand, circumvents underspecified behavior, because it uses *Utraces*.

Theorem 4. $\mathbf{ioco}_U = \mathbf{ioco} \circ \Xi$

6 Conclusions

The results of this paper imply that **io**co can be used for compositional testing if the specifications are modeled as IOTS's; see theorems 1 and 2.

We proposed the function Ξ to complete an LTS specification; i.e., transform an LTS to an IOTS in a way that captures our notion of underspecification. This means that the above results become applicable and the **io**co theory with completed specifications can be used for compositional testing. The resulting relation is slightly weaker than the original **io**co relation; previously conformant implementations are still conformant, but it might be that previously non-conformant implementations are allowed under the modified notion of conformance.

Testing after completion is in principle (much) more expensive since, due to the nature of IOTS's, even the completion of a finite specification already displays infinite testable behavior. As a final result of this paper, we have presented the implementation relation **io**co_U. This relation enables us to use the original component specifications, *before* completion, for compositional testing (see theorem 4).

The insights gained from these results can be recast in terms of *underspecification*. **io**co recognizes two kinds of underspecification: omitting input actions from a state (which implies a *don't care* if an input does occur) and including multiple output actions from a state (which allows the implementation to choose between them). It turns out that the first of these two is not compatible with parallel composition and hiding.

Testing in Context. We have discussed the pre-congruence properties mainly in the context of compositional testing, but the results can easily be transposed to testing in context. Suppose an implementation under test i is tested via a context c . The tester interacts with c , and c interacts with i ; the tester cannot directly interact with i . Then we have $I_i \subseteq U_c$ and $U_i \subseteq I_c$, and L_i is not observable for the tester, i.e., hidden. The tester observes the system as an implementation in a context in the following way: $\mathcal{C}[i] = \text{hide } (I_i \cap U_c) \cup (I_c \cap U_i) \text{ in } c \parallel i$. Now theorem 1 and 2 directly lead to the following corollary for testing in context.

Corollary 2. *Let $s, i \in \text{IOTS}$ occur in test context $\mathcal{C}[_]$. $\mathcal{C}[i] \text{ io}\text{co } \mathcal{C}[s] \Rightarrow i \text{ io}\text{co } s$*

Hence, an error detected while testing the implementation in its context is a real error of the implementation, but not the other way around: an error in the implementation may not be detectable when tested in a context. This holds of course under the assumption that the test context is error free.

Relevance. We have shown a way to handle underspecification of input actions when testing communicating components with the **io**co theory. This idea is new for LTS testing. It is inspired by [10] and work done on partial specifications in FSM testing [11].

Furthermore we have established a pre-congruence result for **io**co for parallel composition and hiding. This is important because it shows that **io**co is usable for compositional testing and testing in context. It establishes a formal relation between the components and the integrated system. As far as we know this result is new for both LTS testing and FSM testing. In FSM testing there are so called Communicating FSM's to model the integration of components. However we have not found any relevant research on the relation between conformance with respect to the CFSM and conformance with respect to its component FSM's.

Traditionally conformance testing is seen as the activity of checking the conformance of a single black box implementation against its specification. The testing of communicating components is often considered to be outside the scope of conformance testing. The pre-congruence result shows that the *ioco* theory can handle both problems in the same way.

Future Work. The current state of affairs is not yet completely satisfactory, because the notion of composition that we require is not defined on general labeled transition systems but just on IOTS's. Testing against IOTS's is inferior, in that these models do not allow the "input underspecification" discussed above: for that reason, testing against an IOTS cannot take advantage of information about "don't care" inputs (essentially, *no* testing is required after a "don't care" input, since by definition every behavior is allowed). We intend to solve this issue by extending IOTS's with a predicate that identifies our added chaotic states. Testing can stop when the specification has reached a chaotic state.

Acknowledgments

We want to thank D. Lee and A. Petrenko for sharing their knowledge of FSM testing and for their insightful discussions.

References

1. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools* **17** (1996) 103–120
2. Tretmans, J.: Testing concurrent systems: A formal approach. In Baeten, J., Mauw, S., eds.: *CONCUR'99*. Volume 1664., LNCS, Springer-Verlag (1999) 46 – 65
3. Fernandez, J.C., Jard, C., Jéron, T., Viho, C.: An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming – Special Issue on COST247, Verification and Validation Methods for Formal Descriptions* **29** (1997) 123–146
4. He, J., Turner, K.: Protocol-Inspired Hardware Testing. In Csopaki, G., Dibuz, S., Tarnay, K., eds.: *Int. Workshop on Testing of Communicating Systems 12*, Kluwer Academic Publishers (1999) 131–147
5. Belinfante, A., Feenstra, J., Vries, R.d., Tretmans, J., Goga, N., Feijs, L., Mauw, S., Heerink, L.: Formal test automation: A simple experiment. In Csopaki, G., Dibuz, S., Tarnay, K., eds.: *12th Int. Workshop on Testing of Communicating Systems*, Kluwer Academic Publishers (1999) 179 – 196
6. Petrenko, A., Yevtushenko, N.: Fault detection in embedded components. In Kim, M., Kang, S., Hong, K., eds.: *Tenth Int. Workshop on Testing of Communicating Systems*, Chapman & Hall (1997) 272–287
7. ISO/IEC JTC1/SC21 WG7, ITU-T SG 10/Q.8: Information Retrieval, Transfer and Management for OSI; Framework: Formal Methods in Conformance Testing. Committee Draft CD 13245-1, ITU-T proposed recommendation Z.500. ISO – ITU-T, Geneve (1996)
8. Jard, C., Jéron, T., Tanguy, L., Viho, C.: Remote testing can be as powerful as local testing. In: *Formal Description Techniques and Protocol Specification, Testing and Verification FORTE XI /PSTV XVIII '99*, Kluwer Academic Publishers (1999)

9. Petrenko, A., Yevtushenko, N., Von Bochman, G.: Fault models for testing in context. In Gotzhein, R., Brederke, J., eds.: FORTE. Volume 69 of IFIP Conference Proceedings., Kluwer (1996) 163 – 178
10. De Nicola, R., Segala, R.: A process algebraic view of Input/Output Automata. *Theoretical Computer Science* **138** (1995) 391–423
11. Petrenko, A., Bochmann, G.v., Dssouli, R.: Conformance relations and test derivation. In Rafiq, O., ed.: Sixth Int. Workshop on Protocol Test Systems. Number C-19 in IFIP Transactions, North-Holland (1994) 157–178
12. van der Bijl, M., Rensink, A., Tretmans, J.: Component based testing with ioco. Technical report, University of Twente (2003)
URL: <http://wwwhome.cs.utwente.nl/~vdbijl/papers/CBT.pdf>.
13. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. *Journal of the Association for Computing Machinery* **31** (1984) 560–599

Defining Observation Objectives for Reactive and Distributed Systems*

Timo Aaltonen

Tampere University of Technology
Institute of Software Systems
P.O.BOX 553, FIN-33101 Tampere, Finland
timo.aaltonen@tut.fi

Abstract. Observation objectives are behaviours that an implementation under test is expected to exhibit during testing. It is desirable to express the objectives at a high level of behavioural abstraction. Unfortunately, current specification methods do not offer proper expressiveness for this. In this paper we demonstrate how observation objectives can be declared when the specification of a system consists of a formal abstraction hierarchy.

1 Introduction

[13] defines *observation objectives* as a set of behaviours¹ that one attempts to produce in an implementation under test (IUT). These objectives are not directly related to the correctness of an IUT, but they are sets of interesting behaviours or behaviours likely to contain errors. Formally, they are simply sets of behaviours that intersect the set of behaviours induced by the specification of the system.

Observation objectives are created manually in terms of a specification. Unfortunately, detailed models of complex systems are themselves complex and, therefore, they are hard to grasp. The complexity is managed with abstractions. By omitting details from the models they become more understandable. This holds also for observation objectives – defining them at a more abstract level than the low-level complex specification makes them more understandable.

Creating abstractions for low-level specifications has been a popular research topic [11,8]. The idea in abstracting is to reduce the complexity of a concrete specification by hiding details. For example, if a specification is a finite state automaton an abstraction can be created by grouping several lower level states into abstract states [10,9]. Then different concrete level behaviours appear similar at the more abstract level, and one of them can be selected as a representative for the whole group of behaviours.

Proper behavioural abstractions do not appear in specifications accidentally but the specifier must have had them in mind when writing the specification. Abstractions in current specification methods concentrate on traditional units of modularity, like classes or packages, not on behaviours. Behavioural abstractions need to be produced afterwards by abstracting low-level specifications. This *post-abtracting* is not always feasible, especially if knowledge of the abstractions is not explicitly stated.

* This work has been partly funded by Academy of Finland (project 5100005).

¹ A behaviour is a possibly infinite sequence of states $b = \langle s_0, s_1, s_2, \dots \rangle$, where each state s_i consists of an assignment of values to the variables of the specification.

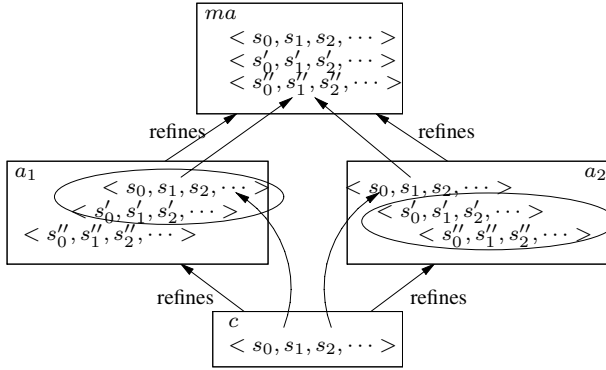


Fig. 1. Mapping of behaviours

In this paper we demonstrate how observation objectives can be created by utilizing a specification method whose specifications consist of abstraction hierarchies [5]. By giving specifications in this way post-abtracting is not needed. We define the notion of a *declarative test case* (DTC) to be an expression determining a set of behaviours that the IUT is enforced to exhibit during testing. The set is an observation objective. DTCs can be given at any level in the specification hierarchy.

The formal abstraction hierarchy enables a rigorous mapping from the behaviours of a concrete specification to the behaviours of a more abstract one. Utilizing this, it is possible to observe a behaviour of the IUT at a concrete level and then map the observed behaviour to any level in the abstraction hierarchy and check if an observed behaviour satisfies a given DTC. Figure 1 illustrates the setting. Rectangles are specifications (c , a_1 , a_2 and ma) forming an abstraction hierarchy. Each specification induces a set of behaviours depicted inside the rectangles. The two ellipses stand for observation objectives. In the figure, the only depicted concrete level behavior is mapped to a behaviour in both specifications which are refined by the most concrete specification. The behaviour satisfies the observation objective given in specification a_1 but not the objective declared in a_2 .

We begin by defining basic notions of observation objectives and declarative test cases in Sect. 2. The section discusses how DTCs are observed in behaviours. Section 3 introduces the specification method DisCo, and Sect. 4 describes how DTCs are created in practice. Section 5 concludes the paper.

2 Declarative Test Cases

In the following we concentrate on specifications that induce possibly infinite sets of infinite sequences of states called *behaviours*. In [2] a formal basis for testing is presented, utilizing joint-action specifications. The testing is justified with a *testing hypothesis*, which states that every implementation under test (IUT) has a formal *model* (m_{IUT}) that is indistinguishable from IUT if they are both put in a black box. In the paper a formal meaning is assigned for correct and incorrect implementations; an implementation is correct if all behaviours that m_{IUT} induces are legal, and incorrect if illegal behaviours exist. Testing is an attempt to produce the latter behaviours.

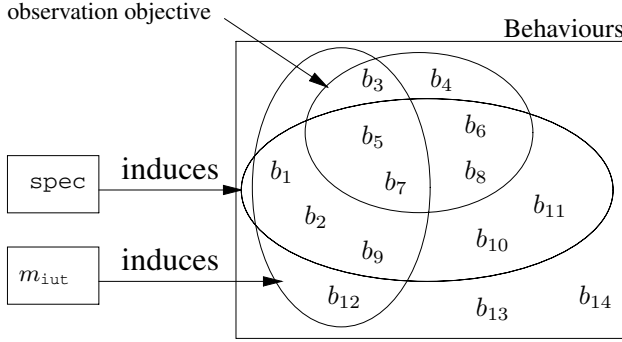


Fig. 2. Venn diagram of behaviours

An *observation objective* [13] is a set of behaviours that one wishes to observe during testing. They are defined in terms of a specification. Whether an IUT exhibits an observation objective or not has nothing to do with correctness, but the two matters are orthogonal. Observed behaviours are divided into three categories with respect to an observation objective. A legal behaviour satisfying an observation objective is flagged as *successful*, an illegal scenario as *failed*, and a legal scenario not satisfying the objective as *inconclusive*.

The Venn diagram in Fig. 2 depicts the different sets of behaviours. The rectangle on the right hand side is the universe of behaviours containing all legal and illegal behaviours in terms of specification $spec$. The specification induces a set of legal behaviours depicted as a large horizontal ellipse in the middle; the horizontal ellipse on the left-hand side is the set of behaviours that the implementation (or its model m_{iut} to be precise) induces. An observation objective is depicted as the small horizontal ellipse. Behaviours b_5 and b_7 are successful, behaviours b_3 and b_{12} are failed, and behaviours b_1 , b_2 and b_9 are inconclusive.

2.1 Defining DTCs

Observation objectives are declared with *declarative test cases* (DTC), which consist of propositions over states. A DTC is a quantified sequence of state expressions.

$$\begin{aligned} \text{DTC} = \exists o_0, o_1, \dots, o_{k-1} \in \text{Objects} : \\ \langle p_0(o_0, o_1, \dots, o_{k-1}), \\ p_1(o_0, o_1, \dots, o_{k-1}), \\ \dots, \\ p_{n-1}(o_0, o_1, \dots, o_{k-1}) \rangle, \end{aligned}$$

where *Objects* is a set of all objects of the specification instance (we assume an object-oriented specification method).

IUT is said to satisfy a DTC if it exhibits a behaviour where, beginning from the initial state s_0 , expression p_0 is satisfied (for some binding to the free variables) until p_1 is satisfied (for the same binding), and so on until p_{n-1} is satisfied.

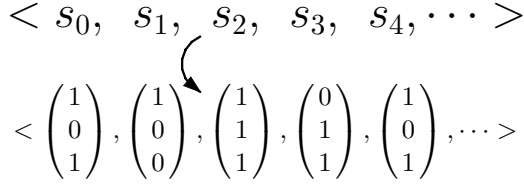


Fig. 3. Mapping from states of behaviour to three tuples

2.2 Satisfying DTC

The computational complexity of a decision problem 'does a given behaviour satisfy a given DTC' is hard. Given a behaviour b , solving the decision problem requires binding the actual objects to quantified variables and then finding a subsequence of states in the behaviour satisfying the DTC. This leads to combinatorial explosion when the number of quantified variables and the number of expressions grows. In this subsection we only scratch the surface of the complexity issues.

Let us first consider the case where the quantified objects are bound:

$$\begin{aligned}
 \widehat{\text{DTC}} = & \langle p_0(\hat{o}_0, \hat{o}_1, \dots, \hat{o}_{k-1}), \\
 & p_1(\hat{o}_0, \hat{o}_1, \dots, \hat{o}_{k-1}), \\
 & \dots, \\
 & p_{n-1}(\hat{o}_0, \hat{o}_1, \dots, \hat{o}_{k-1}) \rangle
 \end{aligned}$$

When validating if a given behaviour $\langle s_0, s_1, \dots \rangle$ satisfies the DTC, the behaviour is mapped to a sequence of n -tuples² $\langle t_0, t_1, \dots \rangle$ so that each state is mapped to an n -tuple $(b_0, b_1, \dots, b_{n-1})$ of booleans. If p_i is satisfied in a state then b_i is 1 and 0 otherwise. Now, regular expression $(1xx\dots x)^+(x1x\dots x)^+\dots(xxx\dots 1)^+$ (where ' x ' denotes either 0 or 1 and '+' denotes repetition of one or more times) defines a language on n -tuples. The language consists of the sequences of n -tuples which are images of the behaviours that satisfy the corresponding $\widehat{\text{DTC}}$. The regular expression simply defines that an arbitrary (but not zero) number of states, starting from the initial state, satisfies the first state expression of the DTC, until starting from some state the second state expression is satisfied, and so on until the last state expression is satisfied.

Figure 3 illustrates the setting for the case where the DTC consists of three propositions p_0, p_1, p_2 . Each state s_i of the sample behaviour (on the top) is mapped to a three-tuple t_i of booleans (in the bottom) according to the sketched rules. Propositions p_0 and p_2 are satisfied in the initial state s_0 , but p_1 is not. Therefore, s_0 is mapped to tuple $(1, 0, 1)$. The second state, which satisfies only p_0 is mapped to tuple $(1, 0, 0)$, and so on. In this case the regular expression to be matched to the sequence of the tuples is $(1xx)^+(x1x)^+(xx1)^+$. It matches for instance to the subsequence t_0, t_1, t_2, t_3 , which means that $\widehat{\text{DTC}}$ is satisfied by the behaviour. The satisfaction can be checked for example with a non-deterministic finite state automaton depicted in Fig. 4.

² Remember that n is the number of state expressions in the DTC.

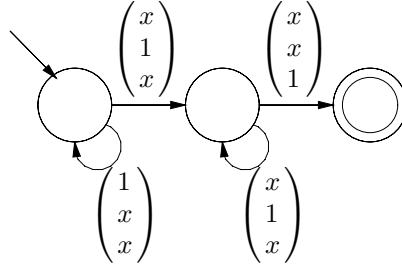


Fig. 4. Automaton for checking whether a DTC is satisfied

In the general case we must, in principle, utilize the described method for each possible binding of objects $o_0, o_1, \dots, o_{k-1}$. In the worst case the number of the needed automata is $l \times (l - 1) \times \dots \times (l - k + 1)$, where l is the number of objects and k is the number of quantified variables.

The required computational resources can be reduced by compromising exactness of notifying the satisfaction of a DTC. One way of doing this is based on selecting some bindings and utilizing only the corresponding automata. The generation of actual test cases can then be driven with this knowledge.

3 The DisCo Method

DTCs themselves do not offer aid to the post-abstraction dilemma, but a specification method that supports abstraction hierarchies is needed. The specifications must be structured into behavioural units – not into traditional units like packages or objects.

One method that fulfills these prerequisites is DisCo method [1], which is used for specifying abstraction hierarchies in [5]. The application area of the method is reactive and distributed systems. DisCo specifications consist of layered abstraction hierarchies, where a more concrete specification is derived by superposition from a more abstract one. The method comprises a specification language, whose semantics is given in terms of TLA logic [12], a compiler and an animation tool [3].

In the following subsections we introduce how abstraction hierarchies are defined with DisCo. A simple specification of automatic teller machines (ATM for short) is used as a running example. The specification is a simplified version of our solution to the specification competition in the Formal Methods '99 conference [4].

3.1 Basics of DisCo

DisCo specifications consist of class and multi-object *joint action* [7,6] definitions. Classes are sets of object sharing the same structure, and they can be inherited in an object-oriented manner. Executing actions is the only way to change the states of objects. An action consists of a name, *roles* in which object can *participate*, a *guard* which is a boolean valued expression usually referring to the participating objects, and a *body* which consists of parallel assignments for attributes of the participating objects. If a guard evaluates to *true* for some participant combination, the action is said to be *enabled*.

Only enabled actions can be executed. The idea behind a joint-action is to model joint behaviour of the participating objects at an abstract level and hide details on how the behaviour is implemented.

Below is an example of a DisCo specification, consisting of classes `Account` and `Till`. The specification is the most abstract view to ATM. `Account` objects have an attribute `balance`. The specification layer has a global assertion `legalBalance` stating that the balance is always non-negative. The assertion does not hold by construction, but – if we are formal – it must be verified. Action `withdraw` models a withdrawal. The action has two roles `acc` and `till`, and one parameter `amount`, modelling the withdrawn amount of money; the guard ensures that the participating account has enough money; in the body of the action the amount is decremented from the account. Detailed description of the action `deposit` is omitted:

```
layer tills is

  class Account is
    balance: integer;
  end;

  class Till is end;

  assert legalBalance is  $\forall a: \text{Account} :: a.\text{balance} \geq 0$ ;

  action withdraw(acc: Account; till: Till; amount: integer) is
    when amount > 0  $\wedge$  acc.balance  $\geq$  amount do
      acc.balance := acc.balance - amount;
    end;

    action deposit(a: Account; amount: integer) is
      when amount > 0 do
        a.balance := a.balance + amount;
      end;
    end;
end;
```

3.2 Refinement

DisCo specifications can be refined by superposition. Among other things, our variant of superposition allows adding new attributes to classes, strengthening guards of actions, giving totally new actions, and adding new assignments to actions. The new assignments given in a refinement are restricted to refer only to newly introduced attributes. This implies that safety properties of the specification are preserved. Behaviours of the refined specification have a unique image in behaviours of all predecessor specification layers. A superposition step can be considered to define a *feature* of the system.

Below is an example of a simple refinement of specification `tills`. The specification layer adds aspects related to bank cards to the specification. A totally new class `Card` is given in the layer. The existing class `Till` is extended with a state machine having states `noCard` and `hasCard`, the latter containing variable `card`, which is a reference to the card inserted in the till. Relation `cardAcc` defines that there exists one account for each card and an arbitrary number of cards for each account. Newly given action `insertCard` models inserting a card to a till. Action `withdraw` is a refinement of the action `tills.withdraw`. The refinement adds a new role `card` to the action and strengthens the guard of the original action with additional conjuncts. The ellipsis in the guard denotes the original guard, and in the body it refers to the original body:

```

layer cards is

    import till;

    class Card is end;

    extend Till by
        state: (noCard, hasCard);
        extend hasCard by
            card: reference Card;
        end;
    end;

    relation cardAcc(Card, Account) is *:1;

    action insertCard(t: Till; c: Card) is
    when t.state'noCard do  $\wedge$ 
        not ( $\exists$  t: Till :: t.state'hasCard.card = c) do
            t.state := hasCard(c);
        end;

    refined withdraw(acc: Account; till: Till; amount: integer; card: Card)
    of withdraw(acc, till, amount) is
    when ... till.state'hasCard  $\wedge$ 
        cardAcc(card, acc) do
        ...
    end;

    action ejectCard(t: Till; c: Card) is
    when t.state'hasCard do
        t.state := noCard();
    end;

end;
    
```

3.3 Composition

Specifying independent refinements of a common *root specification* leads to several parallel refinements. These parallel specifications (or their refinements) can then be conjoined to one composite specification. Then data parts of the component specifications are merged and actions that have a common ancestor in their refinement histories are conjoined.

Specification layer `customers` adds features related to users of the ATM system (they have a wallet in which the withdrawn money is inserted). The specification is, similarly to layer `cards`, a refinement of the root specification `tills`. Figure 5 depicts the final abstraction hierarchy of the ATM system.

A DisCo specification need not fix the number of objects and their initial states. One specification can then be instantiated with various initial states and an arbitrary number of objects.

When an abstraction hierarchy has been constructed, a testing engineer can select any abstraction in the hierarchy and create DTCs with respect to the selected abstraction. This is a natural way to focus testing on selected features of the system.

4 Producing Declarative Test Cases in Practice

DTCs could, in principle, be created manually. However, in real life testing this would be too error-prone and the need for tool support is obvious. Fortunately, the DisCo method

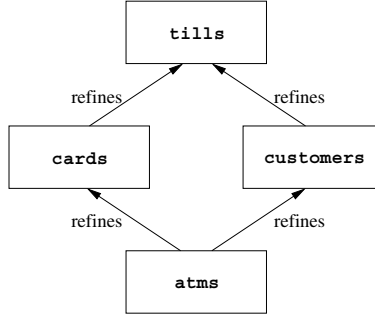


Fig. 5. The layered specification hierarchy of the ATM system

comprises an animation tool for animating (even abstract) specifications on a computer screen. The tool can be utilized for the task.

Before animating a specification it needs to be instantiated, which means creation of a legal initial state with some number of objects in their local initial states. This is carried out by dragging and dropping object on the canvas. After instantiation the tool verifies that the initial conditions and assertions are not violated and (if this verification is successful) the tool changes to execution mode. Now, actions can be executed manually, or the tool can select the next action randomly. A sequence of executed actions can be saved as a scenario and rerun afterwards. The states of the specification instance can also be saved after any executed action.

4.1 Seed of DTC

The behaviours induced by a specification are used for creating DTCs. A behaviour which is used in the creation of a DTC is called a *seed of DTC*. A seed could be utilized as a declarative test case by requiring the IUT to exhibit the same sequence of states. This would be very restrictive: as the IUT corresponds to a specific instance of the specification, the seed should be created using a similar instance, which might not be known at specification time. Moreover, as such it is a legal behaviour and requiring the exhibition of legal behaviours is against the idea of testing.

Let $seed = \langle s_0, s_1, s_2, \dots, s_n \rangle$ be a sequence of states $s_i = o_0^i \oplus o_1^i \oplus \dots \oplus o_{k-1}^i$, where o_j^i is the local state of object o_j in state s_i . In order to create a DTC, the seed is abstracted so that we restrict ourselves only to the objects which participated in the executed actions. These objects become existentially quantified in DTC. Let objects o_0, o_1, o_2 be the objects that participated in the actions when the seed was created. The automatically created DTC is $dtc(seed) = \exists o'_0, o'_1, o'_2 : \langle p_0(o'_0, o'_1, o'_2), p_1(o'_0, o'_1, o'_2), p_2(o'_0, o'_1, o'_2), \dots, p_n(o'_0, o'_1, o'_2) \rangle$, where $p_i(o'_0, o'_1, o'_2)$ states that quantified objects have the same value as “the corresponding” objects in state i of the seed: $p_i(o'_0, o'_1, o'_2) \equiv o'_1 = o_1^i \wedge o'_2 = o_2^i \wedge o'_3 = o_3^i$, where o_j^i is the local state of object o_j in state s_i of the seed. The DTC can be abstracted further manually, for example, by removing requirements for some uninteresting attributes.

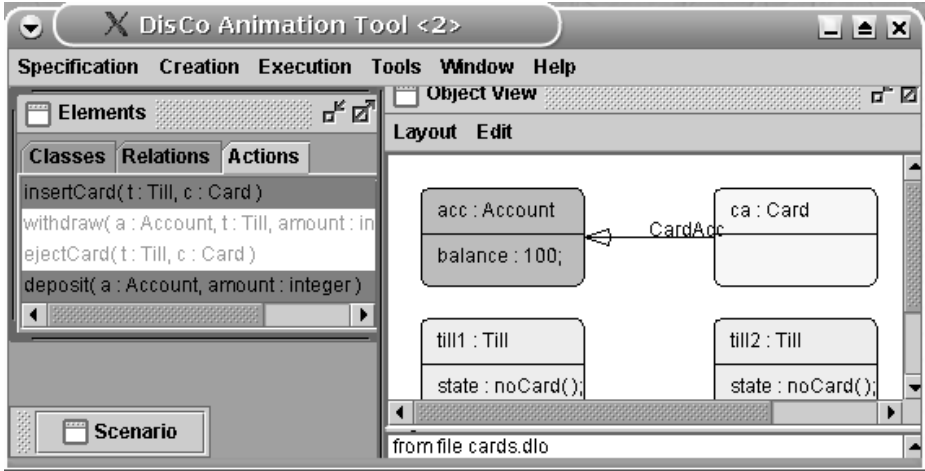


Fig. 6. The initial state in the DISCO ANIMATION TOOL

DISCO ANIMATION TOOL is utilized in creating DTCs. The specifier

1. selects any specification layer,
2. instantiates it,
3. executes a sequence of actions with the tool and
4. saves the initial state and the state after each executed action. This sequence of states is a seed for DTC.

4.2 Example

As an example we show how DTC is created for the ATM specified in Sect. 3.

Specification layer *cards* (in the left hand side of Fig. 5) is selected for the abstraction level (1st item in the enumerated list in Subsection 4.1).

The instance (2nd item) of the specification consists of an account *acc* whose balance is 100, one card *ca*, and two tills *till1* and *till2*. *Card1* is related to account *acc1*. The initial state has been loaded in the DISCO ANIMATION TOOL in Fig. 6. The canvas on the top right-hand corner shows the objects and the relations. Actions are depicted in the top left-hand corner; enabled actions (*insertCard* and *deposit*) are highlighted in green.

The scenario (items 3 and 4) in this example is a basic case where a card is inserted to a till, money is withdrawn and the card is ejected:

```
<insertCard(till1, card1)
  withdraw(acc1, till1, 100, card1)
  eject(till1, card1)>
```

Running the scenario and saving the states after each action execution (and the initial state) with the animator leads to the following seed:

$$\left\langle \begin{pmatrix} acc(1000) \\ ca() \\ till1(noCard) \\ till2(noCard) \end{pmatrix}, \begin{pmatrix} acc(1000) \\ ca() \\ till1(hasCard(ca)) \\ till2(noCard) \end{pmatrix}, \begin{pmatrix} acc(900) \\ ca() \\ till1(hasCard(ca)) \\ till2(noCard) \end{pmatrix}, \begin{pmatrix} acc(900) \\ ca() \\ till1(noCard) \\ till2(noCard) \end{pmatrix} \right\rangle$$

The actual DTC is created by abstracting the seed by quantifying the participants. Object `till2` is dropped out from the DTC, because it did not participate in any of the executed action: $DTC(seed) = \exists a \in Account; c \in Card; t \in Till : \langle p_0(a, c, t), p_1(a, c, t), p_2(a, c, t), p_3(a, c, t) \rangle$, where $p_i(a, c, t)$ states that objects a, c and t have the same values as objects acc, ca and $till1$ in the state i in the seed. Proposition p_0 is given as an example: $p_0(a, c, t) \equiv a = acc(1000) \wedge c = ca() \wedge t = till1(noCard)$.

4.3 Running the Tests

In order to reduce the needed computational resources the DTC can be instantiated for selected assignments to the quantified variables in actual testing. In the ATM example a few cards can first be randomly selected. Relation `CardAcc`, which is a function from `Card` to `Account`, is used to map the selected cards to the corresponding accounts. Tills can be selected randomly.

The actual testing with all the required scaffolding to the IUT is out of scope of this paper. It is just assumed that the behaviour of IUT is observed in terms of the most concrete specification `atms`. During testing the DISCO ANIMATION TOOL executes the same actions as the IUT.

4.4 Test Cases

Specification of an experiment for IUT is called a *test case*. The main component of a test case in our setting is a scenario of executed actions at the lowest-level specification in the specification hierarchy, which is then used for building the required scaffolding to the IUT.

Test cases are created from abstract declarative test cases manually by augmenting abstract scenarios with the required details. In the ATM example augmenting the scenario declared in Subsection 4.2 means only adding `Customer` objects to their roles:

$$\begin{aligned} &\langle insertCard(cust1, till1, card1) \\ &\quad withdraw(cust1, acc1, till1, 100, card1) \\ &\quad eject(cust1, till1, card1) \rangle \end{aligned}$$

Currently an automatic test generation is not an option, but we believe that for typical cases a script language can be developed to carry out the augmentation. For example, if a declarative test case focuses on the charging feature of a telecommunications switch by stating that, whenever a call is routed from terminal A to terminal B, the owner of the terminal A is charged. Now, a low-level scenario is produced by augmenting the abstract scenario with the actions actually taking care of the routing. This is quite simple, because in normal cases the routing procedures follow the same pattern. This is left as future work.

Another approach for creating test cases can be based on utilizing DISCO ANIMATION TOOL and a probabilistic execution mode of actions. In this mode actions are given priorities, and more highly prioritized actions are more likely get selected for execution.

Now, referring to the telecommunications switch example, the testing engineer first divides the actions into two categories: interface actions (i.e., actions initiating a call or hanging up the phone) and actions that are internal to the switch (i.e., actions taking care of the routing). By giving interface actions lower priority than the internal ones, the calls are likely to get routed properly and the aforementioned observation objective becomes satisfied. We believe that it is quite typical that if the action modelling disabling some started procedure (hanging up the phone) are set lower priority it is possible to create test cases utilizing observation objectives. Also in reality many systems spend most of their time executing their internal actions and receive stimuli from the environment only occasionally. However, using priorities in the creation of test cases is also left as future work.

4.5 Animator as a Validation Tool

In [2] we defined an abstraction function which maps behaviours of a more concrete specification to behaviours of a more abstract one. The mapping is utilized in validating the satisfaction of the DTCs. In the ATM example behaviours of specification `atms` are mapped to behaviours of `cards`. At this level of abstraction the satisfaction of the DTC is validated with the animation tool.

The correctness of behaviours is verified at the most concrete level (specification layer `atms` in the example). The DISCO ANIMATION TOOL verifies that executed actions really were enabled and that the states reported by IUT are legal. If they are not, an error has been discovered. The error need not have anything to do with the DTC being satisfied, since the behaviour may be erroneous with respect to other abstractions of the system.

For example, an implementation of the ATM might exhibit a behaviour where a card is inserted to a till, money is withdrawn (but the money does not appear in customer's wallet) and the card is ejected. The behaviour satisfies the DTC defined in Subsection 4.2 but is erroneous according to abstraction `cards`, which requires that the customer receives the withdrawn money.

5 Conclusions and Future Work

In this paper we have proposed a method for defining observation objectives for complex software systems whose specifications tend to be complex as well. An observation objective is a set of behaviours that an implementation under test is expected to exhibit. They are defined by a declarative test case, which is a quantified sequence of state expressions:

$$\begin{aligned} \text{DTC} = & \exists o_0, o_1, \dots, o_{k-1} \in \text{Objects} : \\ & \langle p_0(o_0, o_1, \dots, o_{k-1}), \\ & p_1(o_0, o_1, \dots, o_{k-1}), \\ & \dots, \\ & p_{n-1}(o_0, o_1, \dots, o_{k-1}) \rangle \end{aligned}$$

A DTC is satisfied by a behaviour whose states satisfy p_0 until p_1 is satisfied and so on until p_{n-1} is satisfied.

We demonstrated how objectives of interesting behaviours can be given conveniently if the specification of IUT consists of an abstraction hierarchy. This circumvents the post-abstraction problem present in currently utilized methods. DisCo method was utilized in the example.

The proposed method is formally rigorous but still a practical way for declaring observation objectives for testing. The DisCo toolset offers a decent platform for specifying complex systems. DisCo's way of structuring specifications into behavioural units allows declaring objectives related features of the system, since each layer can be considered to model a feature.

The proposed method requires instantiating the specification for some number of objects in some initial states. If not carried out carefully this can lead to inferior DTCs. For example, if the instantiation does not match the actual implementation, it might be the case that the required behaviours cannot be run at all by the IUT. Every implementation corresponds uniquely to some instance of its DisCo specification. This instance can be loaded to DisCo ANIMATION TOOL and then the testing engineer can verify that the IUT is able to exhibit the required behaviour.

The proposed method does not necessarily lead to an exhaustive test suite with respect to traditional coverage methods. The testing engineer is allowed to create any collection of DTCs, that she wishes. In this paper we have used a specification method whose state space can be infinite, but if some finite state method is utilized, also the traditional coverage measures are available. The state expressions of DTCs are created in such a way that state coverage, for instance, can be managed.

5.1 Future Work

Augmenting the tools with testing facilities is work to be done. There is no formal mapping from DisCo specifications to actual implementations, but they are still made manually. Therefore, some guidelines are needed how an implementation is created, and especially when it comes to testing, how an implementation is observed properly. Currently the observations of an implementation are based on decisions made by the engineer who implements the system. Automatic code generation is, of course, a branch for future studies.

In Subsection 4.3 we sketched few a ways for creating actual test cases based on observation objectives. They can be produced manually by augmenting an abstract behaviour to a more concrete one, by a script language, which takes care of the augmentations, or by utilizing a probabilistic execution mode of actions, which produces the wanted behaviour with some likelihood. These aspects need more research before we can implement them in the toolset.

Acknowledgments

Contributions by other members of the DisCo team – especially Joni Helin and Tommi Mikkonen – are gratefully acknowledged.

References

1. The DisCo project WWW page. At <http://disco.cs.tut.fi> on the World Wide Web, 2001.
2. Timo Aaltonen and Joni Helin. Formal basis for testing with joint-action specifications. In Rob Hierons and Thierry Jéron, editors, *Proceedings of Formal Approaches To Testing 2002 (FATES'02)*, A satellite workshop of *CONCUR'02*, pages 65–77, August 2002.
3. Timo Aaltonen, Mika Katara, and Risto Pitkänen. DisCo toolset – the new generation. *Journal of Universal Computer Science*, 7(1):3–18, 2001. <http://www.jucs.org>.
4. Timo Aaltonen, Pertti Kellomäki, and Risto Pitkänen. Specifying Cash-Point with DisCo. *Formal Aspects of Computing*, 12(4):231–232, 2000.
5. Timo Aaltonen and Tommi Mikkonen. Managing software evolution with a formalized abstraction hierarchy. In Danielle C. Martin, editor, *Proceedings of the Eighth IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2002*, pages 224–231, Greenbelt, Maryland, USA, Dec 2002. IEEE Computer Society.
6. R. J. R. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, October 1988.
7. R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. *Distributed Computing*, 3:73–87, 1989.
8. Jules P. Bergmann and Mark Horowitz. Improving coverage analysis and test generation for large designs. In *Proceedings of The International Conference on Computer-Aided Design (ICCAD'99)*, pages 580–583, November 1999.
9. G. Friedman, A. Hartman, K. Nagin, and T. Shiran. Projected state machine coverage for software testing. *ACM SIGSOFT Software Engineering Notes*, 27(4):134–143, 2002.
10. Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Generating finite state machines from abstract state machines. *ACM SIGSOFT Software Engineering Notes*, 27(4):112–122, 2002.
11. S. Lam and A. Shankar. Protocol verification via projections. *IEEE Transactions on Software Engineering*, SE-10(4):325–342, July 1984.
12. Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.
13. René G. Vries and Jan Tretmans. Towards formal test purposes. In Ed Brinksma and Jan Tretmans, editors, *Proceedings of workshop on Formal Approaches to Testiong of Software 2001*, pages 61–76, August 2001.

Time-Optimal Real-Time Test Case Generation Using UPPAAL

Anders Hessel¹, Kim G. Larsen², Brian Nielsen², Paul Pettersson¹, and Arne Skou²

¹ Department of Information Technology, Uppsala University
P.O. Box 337, SE-751 05 Uppsala, Sweden
{hessel, paupet}@it.uu.se

² Department of Computer Science, Aalborg University
Fredrik Bajersvej 7E, DK-9220 Aalborg, Denmark
{kgl, bnielsen, ask}@cs.auc.dk

Abstract. Testing is the primary software validation technique used by industry today, but remains ad hoc, error prone, and very expensive. A promising improvement is to automatically generate test cases from formal models of the system under test.

We demonstrate how to automatically generate real-time conformance test cases from timed automata specifications. Specifically we demonstrate how to *efficiently generate* real-time test cases with *optimal* execution time i.e test cases that are the *fastest* possible to execute. Our technique allows time optimal test cases to be generated using manually formulated test purposes or generated automatically from various coverage criteria of the model.

1 Introduction

Testing is the execution of the system under test in a controlled environment following a prescribed procedure with the goal of measuring one or more quality characteristics of a product, such as functionality or performance. Testing is the primary software validation technique used by industry today. However, despite the importance and the many resources and man-hours invested by industry (about 30% to 50% of development effort), testing remains quite ad hoc and error prone.

We focus on conformance testing i.e., checking by means of execution whether the behavior of some black box implementation conforms to that of its specification, and moreover doing this within minimum time. A promising approach to improving the effectiveness of testing is to base test generation on an abstract formal model of the system under test (SUT) and use a test generation tool to (automatically or user guided) generate and execute test cases. Model based test generation has been under scientific study for some time, and practically applicable test tools are emerging [4,16,18,10]. However, little is still known in the context of real-time systems.

An important principal problem in generating real-time test cases is to compute *when* to stimulate the system and expect response, and to compute the associated correct verdict. This usually requires (symbolic) analysis of the model which in turn may lead to the state explosion problem. Another problem is *how to select* a very limited set of test cases to be executed from the extreme large number (usually infinitely many) of potential ones.

This paper demonstrates how it is possible to generate *time-optimal* test cases and test suites, i.e. test cases and suites that are guaranteed to take the least possible time to execute. The required behavior is specified using a deterministic and output urgent class of UPPAAL style timed automata. The UPPAAL model checking tool implements a set of efficient data-structures and algorithms for symbolic reachability analysis of timed automata. We then use the fastest diagnostic trace facility of the UPPAAL tool to generate time optimal test sequences. Test cases can either be selected through manually formulated test purposes or automatically from three natural coverage criteria—such as transition or location coverage—of the timed automata model.

Time optimal test suites are interesting for several reasons. First, reducing the total execution time of a test suite allows more behavior to be tested in the (limited) time allocated to testing. Second, it is generally desirable that regression testing can be executed as quickly as possible to improve the turn around time between software revisions. Third, it is essential for product instance testing that a thorough test can be performed without testing becoming the bottleneck, i.e., the test suite can be applied to all products coming of an assembly line. Finally, in the context of testing of real-time systems, we hypothesize that the fastest test case that drives the SUT to some state, also has a high likelihood of detecting errors, because this is a stressful situation for the SUT to handle.

The rest of the paper is organized as follows: Section 2 discusses related work, and Section 3 introduces our framework for testing real-time systems based on a testable subclass of timed automata. Section 4 and 5 describe how to encode test purposes and test criteria, and report experimental results respectively. Section 6 concludes the paper.

2 Related Work

Relatively few proposals exist that deal explicitly and systematically with testing real-time properties [11,9,6,17,8,5,7,14,15]. In [5,8,17] test sequences are generated from a timed automata (TA) by applying variations of finite state machine (FSM) checking sequence techniques (see eg. [13]) to a discretization of the state space. Experience shows that this approach suffers seriously from the state explosion problem and resulting large number of test sequences. The work in [9] and [11] also use checking sequences, but is based on different structures and state verification methods. Both assume determinism, but not output urgency. To distinguish sequences that can always be executed to completion independent on output timing and sequences that may be executed to completion, [9] defines may- and must-traceability of transition sequences in a TA. The unique IO sequence (UIOv) method is then applied to a FSM derived from the TA by simply removing the clock conditions on transitions. The sequences are then checked for their may- and must-traceability, and the procedure is re-iterated when necessary. This may result in many iterations and in incomplete test-suites. The work in [11] assumes a further restricted TA model where all transitions with the same observable action resets the same set of clocks. The TA is first translated into a (larger) alternative automaton where clock constraints are represented as set-timer and expire-timer events. Based on this, the generalized Wp method is used to compute checking sequences.

In most FSM based approaches, tests are *selected* based on a fault-model identifying implementation faults that is desired to be (or can be) detected during testing. Little or no evidence is given to support that the real-time fault models correspond to faults

that occur frequently in practice. Another problem is the required assumptions about the number of states in the SUT, which in general is difficult to estimate. The coverage approach guarantees that the test suite is derived systematically and that it provides a certain level of thoroughness, which is important in industrial practice. It is important to stress that this is a practically founded heuristic *test selection* technique. Similarly, when time optimal sequences are generated, this is also a level of test selection, where only the fastest to execute are selected. Our goal is *not* full fault coverage that will in principle guarantee that the SUT is correct if it passes all generated tests.

A different approach to test generation and selection is [6] where a manually stated test purpose is used to define the desired sequences to be observed on the SUT. A synchronous product of the test purpose and TA model is first formed and used to extract a symbolic test sequence with timing constraints that reach a goal state of the test purpose. This symbolic trace can be interpreted at execution time to give a final verdict. This work does not address test suite optimization or time optimality, does not address test generation without an explicit test purpose, and does not appear to be implemented in a tool. [15] proposes a fully automatic method for generation of real-time test sequences from a subclass of TA called event-recording automata which restricts how clocks are reset. The technique is based on symbolic analysis and coverage of a coarse equivalence class partitioning of the state space.

Our work is based on existing efficient and well proven symbolic analysis techniques of TA, and unlike others addresses time optimal testing. Most other work on optimizing test suites, e.g [1,19,10], focus on minimizing the length of the test suite which is not directly linked to the execution time because some events take longer to produce or real-time constraints are ignored. Others have used (untimed) model-checking tools to produce test suites for various model coverage criteria e.g., [10].

The main contributions of the paper are 1) application of time and cost optimal reachability analysis algorithms to the context of *time-optimal test case generation*, 2) an *automatic* technique to generate time optimal covering test suites for three *important coverage criteria*, 3) through creative use of the diagnostic trace facility of UPPAAL, a *test generation tool* exists that is based on efficient and well-proven algorithms, and finally 4) we provide *experimental evidence* in that the proposed technique has practical merits.

3 Timed Automata and Testing

We will assume that both the system under test (SUT) and the environment in which it operates are modeled as TA.

3.1 Timed Automata

Let X be a set of non-negative real-valued variables called *clocks*, and $Act = \mathcal{I} \cup \mathcal{O} \cup \{\tau\}$ a set of input actions $\mathcal{I}$ and output-actions $\mathcal{O}$, (denoted $a?$ and $a!$), and the non-synchronizing action (denoted τ). Let $\mathcal{G}(X)$ denote the set of *guards* on clocks being conjunctions of simple constraints of the form $x \bowtie c$, and let $\mathcal{U}(X)$ denote the set of *updates* of clocks corresponding to sequences of statements of the form $x := c$,

where $x \in X$, $c \in \mathbb{N}$, and $\bowtie \in \{\leq, <, =, \geq\}^1$. A *timed automaton* (TA) over (Act, X) is a tuple (L, ℓ_0, I, E) , where L is a set of locations, $\ell_0 \in L$ is an initial location, $I : L \rightarrow \mathcal{G}(X)$ assigns invariants to locations, and E is a set of edges such that $E \subseteq L \times \mathcal{G}(X) \times Act \times \mathcal{U}(X) \times L$. We write $\ell \xrightarrow{g, \alpha, u} \ell'$ iff $(\ell, g, \alpha, u, \ell') \in E$.

The semantics of a TA is defined in terms of a timed transition system over states of the form $p = (\ell, \sigma)$, where ℓ is a location and $\sigma \in \mathbb{R}_{\geq 0}^X$ is a clock valuation satisfying the invariant of ℓ . Intuitively, there are two kinds of transitions: delay transitions and discrete transitions. In delay transitions, $(\ell, \sigma) \xrightarrow{d} (\ell, \sigma + d)$, the values of all clocks of the automaton are incremented with the amount of the delay, d . Discrete transitions $(\ell, \sigma) \xrightarrow{\alpha} (\ell', \sigma')$ correspond to execution of edges $(\ell, g, \alpha, u, \ell')$ for which the guard g is satisfied by σ . The clock valuation σ' of the target state is obtained by modifying σ according to updates u . We write $p \xrightarrow{\gamma}$ as a short for $\exists p'. p \xrightarrow{\gamma} p', \gamma \in Act \cup \mathbb{R}_{\geq 0}$. A timed trace is a sequence of alternating time delays and actions in Act .

A *network of TA* $\mathcal{A}_1 \parallel \dots \parallel \mathcal{A}_n$ over (Act, X) is defined as the parallel composition of n TA over (Act, X) . Semantically, a network again describes a timed transition system obtained from those of the components by requiring synchrony on delay transitions and requiring discrete transitions to synchronize on complementary actions (i.e. $a?$ is complementary to $a!$).

3.2 UPPAAL and Time Optimal Reachability Analysis

UPPAAL is a verification tool for a TA based modeling language. Besides dense clocks, the tool supports both simple and complex data types like bounded integers and arrays as well as synchronization via shared variables and actions. The specification language supports safety, liveness, deadlock, and response properties.

To produce test sequences, we shall make use of UPPAAL's ability to generate diagnostic traces witnessing a submitted safety property. Currently UPPAAL supports three options for diagnostic trace generation: *some trace* leading to the goal state, the *shortest trace* with the minimum number of transitions, and *fastest trace* with the shortest accumulated time delay. The underlying algorithm used for finding time-optimal traces is a variation of the A*-algorithm [2,12]. Hence, to improve performance it is possible to supply a heuristic function estimating the remaining cost from any state to the goal state.

Throughout the paper we use UPPAAL syntax to illustrate TA, and the figures are direct exports from UPPAAL. Initial locations are marked using a double circle. Edges are by convention labeled by the triple: guard, action, and assignment in that order. The internal τ -action is indicated by an absent action-label. Committed locations are indicated by a location with an encircled "C". A committed location must be left immediately as the next transition taken by the system. Finally, bold-faced clock conditions placed under locations are location invariants.

3.3 Deterministic, Input Enabled and Output Urgent TA

To ensure time optimal testability, the following semantic restrictions turn out to be sufficient. Following similar restrictions as in [17], we define the notion of deterministic,

¹ To simplify the presentation in the rest of the paper, we restrict to guards with non-strict lower bounds on clocks.

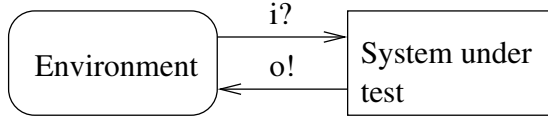


Fig. 1. Test Specification

input enabled and output urgent TA, DIEOU-TA, by restricting the underlying timed transition system defined by the TA as follows:

1. *Determinism.* Two transitions with the same label leads to the same state, i.e., for every semantic state $p = (\ell, \sigma)$ and action $\gamma \in Act \cup \{\mathbb{R}_{\geq 0}\}$, whenever $p \xrightarrow{\gamma} p'$ and $p \xrightarrow{\gamma} p''$ then $p' = p''$.
2. *(Weak) input enabled.* At any time any input action is enabled, i.e., whenever $p \xrightarrow{d}$ for some delay $d \in \mathbb{R}_{\geq 0}$ then $\forall a \in \mathcal{I}. p \xrightarrow{a}$.
3. *Isolated Outputs.* If an output (or τ) is enabled then no other input or output transition is enabled, i.e., $\forall \alpha \in \mathcal{O} \cup \{\tau\}. \forall \beta \in \mathcal{O} \cup \mathcal{I} \cup \{\tau\}$ whenever $p \xrightarrow{\alpha}$ and $p \xrightarrow{\beta}$ then $\alpha = \beta$.
4. *Output urgency.* When an output (or τ) is enabled, it will occur immediately, i.e., whenever $p \xrightarrow{\alpha}, \alpha \in \mathcal{O} \cup \{\tau\}$ then $p \not\xrightarrow{d}, d \in \mathbb{R}_{\geq 0}$.

We assume that the test specification is given as a closed network of TA that can be partitioned into one subnetwork modeling the behavior of the SUT, and one modeling the behavior of its environment (ENV), see Figure 1. Often the SUT operates in specific environments, and it is only necessary to establish correctness under the (modeled) environment assumptions; otherwise the environment model can be replaced with a completely unconstrained one that allows all possible interaction sequences.

We assume that the tester can take the place of the environment and control the SUT via a distinguished set observable input and output actions. For the SUT to be testable the subnetwork modeling it should be *controllable* in the sense that it should be possible for an environment to drive the subnetwork model through all of its syntactical parts (e.g. transitions and locations). We therefore assume that the SUT specification is a DIEOU-TA, and that the SUT can be modeled by some unknown DIEOU-TA (this assumption is commonly referred to as the testing hypothesis). The environment model need not be a DIEOU-TA.

We use the simple light switch controller in Figure 2 to illustrate the concepts. The user interacts with the controller by touching a touch sensitive pad. The light has three intensity levels: OFF, DIMMED, and BRIGHT. Depending on the timing between successive touches (recorded by the clock x), the controller toggles the light levels. For example, in dimmed state, if a second touch is made quickly (before the switching time $T_{sw} = 4$ time units) after the touch that caused the controller to enter dimmed state (from either off or bright state), the controller increases the level to bright. Conversely, if the second touch happens after the switching time, the controller switches the light off. If the light controller has been in off state for a long time (longer than or equal to $T_{idle} = 20$), it should reactivate upon a touch by going directly to bright level. We leave it to the reader to verify for herself that the conditions of DIEOU-TA are met by the model given.

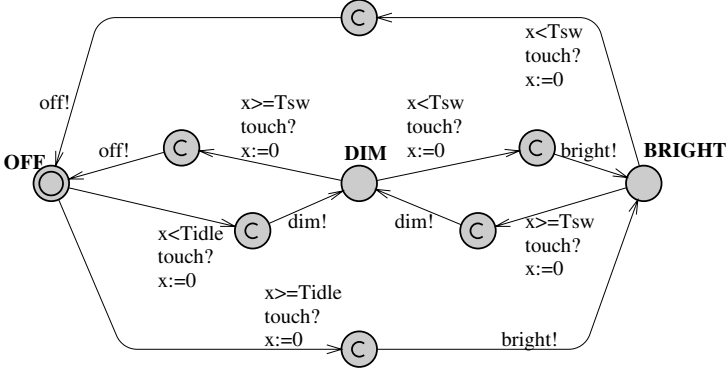


Fig. 2. Light Controller

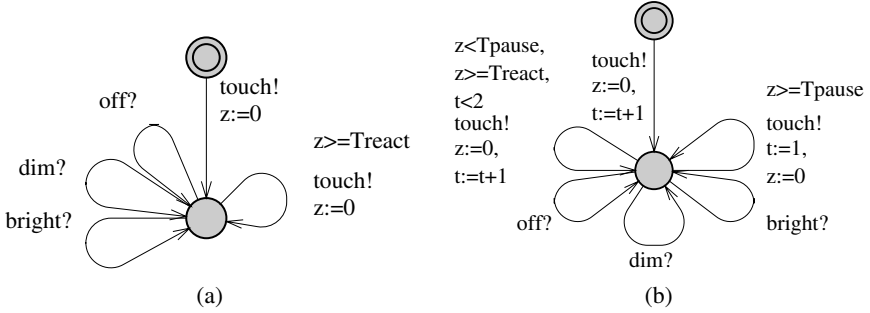


Fig. 3. Two possible environment models for the simple light switch

The environment model shown in Figure 3(a) models a user capable of performing any sequence of touch actions. When the constant T_{react} is set to zero he is arbitrarily fast. A more realistic user is only capable of producing touches with a limited rate; this can be modeled setting T_{react} to a non-zero value. Figure 3(b) models a different user able to make two quick successive touches (counted by integer variable t), but which then is required to pause for some time (to avoid cramp), e.g., $T_{pause} = 5$.

3.4 From Diagnostic Traces to Test Cases

Let A be the TA network model of the SUT together with its intended environment ENV. A diagnostic trace produced by UPPAAL for a given reachability question on A demonstrates the sequence of moves to be made by each of the system components and the required clock constraints needed to reach the targeted location. A (concrete) diagnostic trace will have the form:

$$(S_0, E_0) \xrightarrow{\gamma_0} (S_1, E_1) \xrightarrow{\gamma_1} (S_2, E_2) \xrightarrow{\gamma_2} \dots (S_n, E_n)$$

where S_i, E_i are states of the SUT and ENV, respectively, and γ_i are either time-delays or synchronization (or internal) actions. The latter may be further partitioned into purely SUT or ENV transitions (hence invisible for the other part) or synchronizing transitions between the SUT and the ENV (hence observable for both parties).

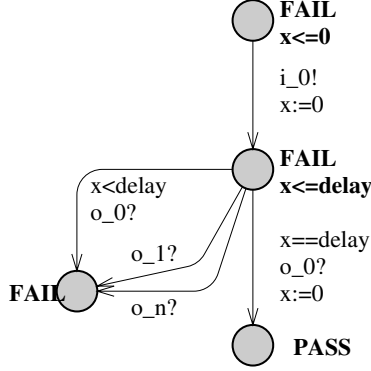


Fig. 4. Test case automaton for the sequence $i_0! \cdot \text{delay} \cdot o_0?$

For DIEOU-TA a *test sequence* is an alternating sequence of concrete delay actions and observable actions. From the diagnostic trace above a *test sequence*, λ , may be obtained simply by projecting the trace to the ENV-component, while removing invisible transitions, and summing adjacent delay actions. Finally, a *test case* to be executed on the real SUT implementation may be obtained from λ by the addition of *verdicts*.

Adding the verdicts require some comments on the chosen correctness relation between the specification and SUT. In this paper we require timed trace inclusion, i.e. that the timed traces of the implementation are included in the specification. Thus after any input sequence, the implementation is allowed to produce an output only if the specification is also able to produce that output. Similarly, the implementation may delay (thereby staying silent) only if the specification also may delay. The test sequences produced by our techniques are derived from diagnostic traces, and are thus guaranteed to be included in the specification.

To clarify the construction we may model the test case itself as a TA A_λ for the test sequence λ . Locations in A_λ are labeled using two distinguished labels, **pass** and **fail**. The execution of a test case is now formalized as a parallel composition of the test case automaton A_λ and SUT A_S .

$$S \text{ passes } A_\lambda \text{ iff } A_\lambda \parallel A_S \not\rightarrow \text{fail}$$

A_λ is constructed such that a *complete execution* terminates in a **fail** state if the SUT cannot perform λ and such that it terminates in a **pass** state if the SUT can execute all actions of λ . The construction is illustrated in Figure 4.

4 Test Generation

4.1 Single Purpose Test Generation

A common approach to the generation of test cases is to first manually formulate a set of informal test purposes and then to formalize these such that the model can be used to generate one or more test cases for each test purpose. A test purpose is a specific test objective (or purpose) that the tester would like to observe on the SUT.

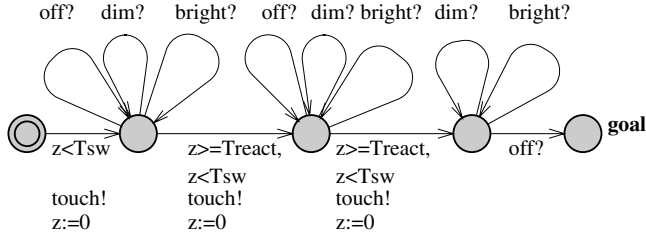


Fig. 5. Test Environment for TP2

Because we use the diagnostic trace facility of a model-checker based on reachability analysis, the test purpose must be formulated as a property that can be checked by reachability analysis of the combined ENV and SUT model. We propose different techniques for this. Sometimes the test purpose can be directly transformed into a simple location reachability check. In other cases it may require decoration of the model with auxiliary flag variables. Another technique is to replace the environment model with a more restricted one that matches the behavior of the test purpose only.

TP1: Check that the light can become bright.

TP2: Check that the light switches off after three successive touches.

TP1 can be formulated as a simple reachability property: $E<> \text{LightController} . \text{bright}$ (i.e. eventually in some future the `lightController` automata enters location `bright`).

Generating the *shortest* diagnostic trace results in the test sequence: $20 \cdot \text{touch!} \cdot 0 \cdot \text{bright?}$. However, the *fastest sequence* satisfying the purpose is $0 \cdot \text{touch!} \cdot 0 \cdot \text{dim?} \cdot 0 \cdot \text{touch!} \cdot 0 \cdot \text{bright?}$.

TP2 can be formalized using the restricted environment model² in Figure 5 with the property $E<> \text{tpEnv} . \text{goal}$. The fastest test sequence is $0 \cdot \text{touch!} \cdot 0 \cdot \text{dim?} \cdot 0 \cdot \text{touch!} \cdot 0 \cdot \text{bright?} \cdot 0 \cdot \text{touch!} \cdot 0 \cdot \text{off?}$.

4.2 Coverage Based Test Generation

Often the tester is interested in creating a test suite that ensures that the specification or implementation is covered in a certain way. This ensures that a certain level of systemacy and thoroughness has been achieved in the test generation process. Here we explain how test sequences with guaranteed coverage of the SUT model can be computed using reachability analysis, effectively giving automated tool support. In the next subsection, we show how to generalize the technique to generate sets of test sequences.

A large suite of coverage criteria have been proposed in the literature, such as statement, transition, and definition-use coverage, each with its own merits and application domain. We explain how to apply some of these to TA models.

Edge Coverage: A test sequence satisfies the *edge-coverage criterion* if, when executed on the model, it traverses every edge of the selected TA-components. Edge coverage

² It is possible to use UPPAAL's committed location feature to compose the test purpose and environment model in a compositional way. Space limitations prevents us from elaborating on this approach.

can be formulated as a reachability property in the following way: add an auxiliary variable e_i of type boolean (initially false) for each edge to be covered (typically realized as a bit array in UPPAAL), and add to the assignments of each edge i an assignment $e_i := \text{true}$; a test suite can be generated by formulating a reachability property requiring that all e_i variables are true: $E <> (e_0 == \text{true} \text{ and } e_1 == \text{true} \dots e_n == \text{true})$. The auxiliary variables are needed to enable formulation of the coverage criterion as a reachability property using the UPPAAL property specification language which is a restricted subset of CTL.

The light switch in Figure 2 requires a bit-array of 12 elements (one per edge). When the environment can touch arbitrarily fast the generated fastest edge covering test sequence has the accumulated execution time 28. The solution (there might be more traces with the same fastest execution time) generated by UPPAAL is:

EC: $0 \cdot \text{touch!} \cdot 0 \cdot \text{dim?} \cdot 0 \cdot \text{touch!} \cdot 0 \cdot \text{bright?} \cdot 0 \cdot \text{touch!} \cdot 0 \cdot \text{off?} \cdot 20 \cdot \text{touch!} \cdot 0 \cdot \text{bright?} \cdot 4 \cdot \text{touch!} \cdot 0 \cdot \text{dim?} \cdot 4 \cdot \text{touch!} \cdot 0 \cdot \text{off?}$.

Location Coverage: A test sequence satisfies the *location-coverage criterion* if, when executed on the model, it visits every location of the selected TA-components. To generate test sequences with location coverage, we introduce an auxiliary variable s_i of type boolean (initially false for all locations except the initial) for each location ℓ_i to be covered. For every edge with destination ℓ_i : $\ell' \xrightarrow{g,a,u} \ell_i$ add to the assignments u $s_i := \text{true}$; the reachability property will then require all s_i variables to be true.

Definition-Use Pair Coverage: The definition-use pair criterion is a data-flow coverage technique where the idea is to cover paths in which a variable is *defined* (i.e. appears in the left-hand side of an assignment) and later is *used* (i.e. appears in a guard or the right-hand side of an assignment). Due to space-limitation, we restrict the presentation to clocks, which can be *used* in guards only.

We use (v, e_d, e_u) to denote a *definition-use pair* (DU-pair) for variable v if e_d is an edge where v is defined and e_u is an edge where v is used. A DU-pair (v, e_d, e_u) is valid if e_u is reachable from e_d and v is not redefined in the path from e_d to e_u . A test sequence covers (v, e_d, e_u) iff (at least) once in the sequence, there is a valid DU-pair (v, e_d, e_u) . A test sequence satisfies the (all-uses) DU-pair coverage criterion of v if it covers all valid DU-pairs of v .

To generate test sequences with definition-use pair coverage, we assume that the edges of a model are enumerated, so that e_i is the number of edge i . We introduce an auxiliary data-variable v_d (initially false) with value domain $\{\text{false}\} \cup \{1 \dots |E|\}$ to keep track of the edge at which variable v was last defined, and a two-dimensional boolean array du of size $|E| \times |E|$ (initially false) to store the covered pairs. For each edge e_i at which v is defined we add $v_d := e_i$, and for each edge e_j at which v is used we add the conditional assignment *if* $(v_d \neq \text{false})$ *then* $du[v_d, e_j] := \text{true}$. Note that if v is both used and defined on the same edge, the array assignment must be made before the assignment of v_d .

The reachability property will then require all $du[i, j]$ representing valid DU-pairs to be true for the (all-uses) DU-pair criterion. Note that a test sequence satisfying the DU-pair criterion for several variables can be generated using the same encoding, but extended with one auxiliary variable and array for each covered variable.

4.3 Test Suite Generation

Often a single covering test sequence cannot be obtained for a given test purpose or criterion (e.g. due to dead-ends in the model). To solve this problem, we allow for the model (and SUT) to be *reset* to its initial state, and to continue the test after the reset to cover the remaining parts. The generated test will then be interpreted as a test suite consisting of a set of test sequences separated by resets (assumed to be implemented correctly in the SUT).

To introduce resets in the model, we shall allow the user to designate some locations as being reset-able. Obviously, performing a reset may take some time T_r that must be taken into consideration when generating time optimal test sequences. Reset-able locations can be encoded into the model by adding reset transitions leading back to the initial location. Let x_r be an additional clock used for reset purposes, and let ℓ be a reset-able location. Two reset-edges and a new location ℓ' must then be added from ℓ to the initial location ℓ_0 , i.e.,

$$\ell \xrightarrow{\text{reset!}, x_r := 0} \ell'_{(x_r \leq T_r)} \xrightarrow{x_r == T_r, \tau, u_0} \ell_0$$

Here u_0 are the assignment needed to reset clocks and other variables in the model (excluding auxiliary variables encoding test purpose or coverage criteria³). If more than one component is present in either the SUT-model or environment model, the reset-action must be communicated atomically to all of them. This can be done using the committed location feature of UPPAAL. Further note that it may be possible to obtain faster (covering) test suites, if more reset-able locations are added, obviously depending on the time required to perform the reset, at the expense of increased model size.

4.4 Environment Behavior

A potential problem of the techniques presented above is that the generated test sequences may be non-realizable, in that they may require the environment of SUT to operate infinitely fast. In general, it is only necessary to establish correctness of SUT under the (modeled) environment assumptions. Therefore assumptions about the environment can be modeled explicitly and will then be taken into account during test sequence generation. In the following, we demonstrate how different environment assumptions affect the generated test sequences.

Consider an environment where the user takes at least 2 time units between each touch action; such an environment can be obtained by setting the constant T_{react} to 2 in Figure 3(a). The fastest test sequences become:

TP1: $0 \cdot \text{touch!} \cdot 0 \cdot \text{dim?} \cdot 2 \cdot \text{touch!} \cdot 0 \cdot \text{bright?}$

TP2: $0 \cdot \text{touch!} \cdot 0 \cdot \text{dim?} \cdot 2 \cdot \text{touch!} \cdot 0 \cdot \text{bright?} \cdot 2 \cdot \text{touch!} \cdot 0 \cdot \text{off?}$.

Also reexamine the test suite **EC** generated by edge coverage, and compare with the one of execution time 32 generated when T_{react} equals 2:

EC': $0 \cdot \text{touch!} \cdot 0 \cdot \text{dim?} \cdot 4 \cdot \text{touch!} \cdot 0 \cdot \text{off?} \cdot 20 \cdot \text{touch!} \cdot 0 \cdot \text{bright?} \cdot 4 \cdot \text{touch!} \cdot 0 \cdot \text{dim?} \cdot 2 \cdot \text{touch!} \cdot 0 \cdot \text{bright?} \cdot 2 \cdot \text{touch!} \cdot 0 \cdot \text{off?}$.

³ In the encoding of DU-pair coverage, the variables v_d should be set to **false** at resets.

When the environment is changed to the pausing user (can perform 2 successive quick touches after which he is required to pause for some time: reaction time 2, pausing time 5), the fastest sequence has execution time 33, and follows a completely different strategy, that ensures that one of the additional waiting times T_{pause} is overlapped with a position where the tester needed to wait anyway.

EC”: $0 \cdot touch! \cdot 0 \cdot dim? \cdot 2 \cdot touch! \cdot 0 \cdot bright? \cdot 5 \cdot touch! \cdot 0 \cdot dim? \cdot 4 \cdot touch! \cdot 0 \cdot off? \cdot 20 \cdot touch! \cdot 0 \cdot bright? \cdot 2 \cdot touch! \cdot 0 \cdot off?$.

5 Experiments

In the previous section we presented techniques to compute time optimal covering test suites. In the following we show empirically that the performance of our technique is sufficient for practically relevant examples, and to indicate how heuristic search methods can be used to compute optimal or near optimal test cases from very large models. We are concerned with both the *execution time* of the generated test sequence, and the time and memory used to *generate* it.

5.1 The Touch Sensitive Switch

Most of the experiments reported here are based on a model of a touch sensitive light switch (TSS). It has Max levels of brightness (0 corresponds to off). The lamp is operated by touching its wire, i.e. the wire can be grasped and released. The behavior of the controller can be expressed as follows: If the light is on, then a single grasp and release of the wire, will switch off the light. If the light is off, then a single grasp and release will switch on the light at the previous brightness level. Continuous holding of the wire increases the brightness (resp. decreases) if it was previously decreasing (resp. increasing). Once the maximum (resp. minimum) level is reached the brightness level decrease (resp. increase).

In reality a user can only perform two actions on the wire: *grasp* and *release*, and the time-separation between the two events is translated into either nothing (if the separation is very short), *touch* if it is short, and into a *starthold* and *endhold* pair if the separation is long. In the UPPAAL-model this translation is done by the interface component, shown in Figure 6(a). The dimmer component shown in Figure 7 reacts to *starthold* and *endhold* actions with a dimming effect. When changing the brightness level L , it is assumed that some maximum time (*Delay*) will elapse between two levels. The switch component shown in Figure 6(b) reacts to *touch* events by switching the light on to the previous light level OL , or off. The user is modeled in Figure 6(c).

We vary the model in two ways. First, the user may be *patient* or *impatient*. The impatient user insists on requiring interaction at least every $Wait = 15$ time units controlled by the invariant in user – this makes it harder for the user to change the intensity because he “gives up” the hold after just increasing the light one level. This invariant is removed in the patient user. Secondly, we vary the number of light levels from $Max = 10$ and up.

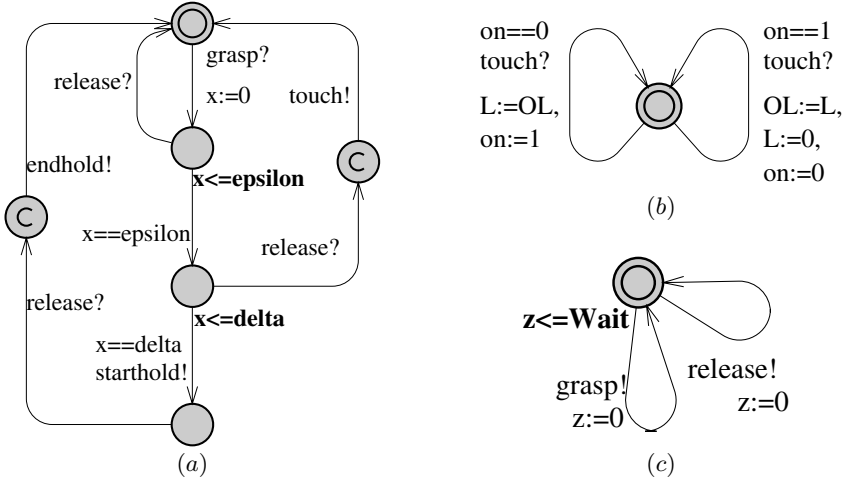


Fig. 6. Interface Automaton (a), Switch Automaton (b), and User Automaton (c)

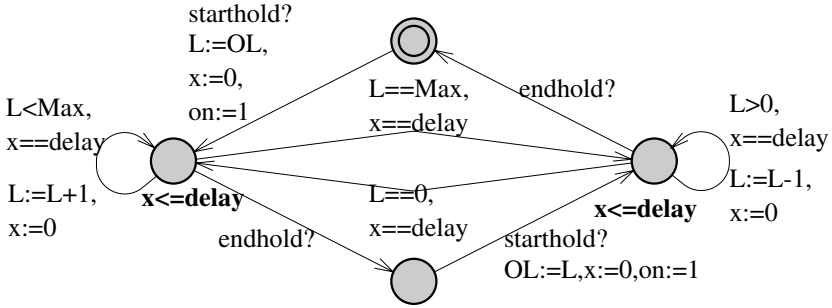


Fig. 7. Dimmer Automaton

Table 8 shows the optimal execution times (in time units) for test suites generated from different coverage criteria of the TSS, or selected subsets of components thereof, and the length (number of transitions) of the generated test suite. We notice that the patient user results in shorter and faster traces in our experiments.

5.2 System Size and Environment Behavior

To see how our technique scales, we increase the number of light levels in the TSS model. The result, listed in Table 9, shows that the particular example scales well: execution time (in time units), generation time, and memory usage for the impatient user increase essentially linearly with the number of light-levels. This is not surprising as the system size is varied by adjusting a counter, and not the number of parallel components.

It is more interesting to compare the patient and impatient user. Consider the system with 50 light levels. The optimal execution time for the impatient user is high (1183 time

Table 8. Optimal execution time and suite length for various coverage criteria

Coverage Criterion	Impatient		Patient	
	Execution time	Suite length	Execution time	Suite length
Location _{Dimmer}	20	12	20	12
Location _{Dimmer, Switch, Interface}	25	17	25	17
Edge _{Dimmer}	253	176	53	38
Edge _{Interface}	15	14	15	14
Edge _{Dimmer, Switch, Interface}	263	188	63	50
Edge _{Interface} +Location _{Dimmer}	25	19	25	19
Def-Use _{on}	40	34	40	34
Def-Use _{OL}	45	34	45	34

Table 9. Cost of obtaining edge coverage of the TSS with increasing light levels

Levels	Impatient			Patient		
	Execution time	Generation time(s)	Memory usage (MB)	Execution time	Generation time (s)	Memory usage(MB)
10	263	2.06	9.1	63	3.19	10.1
20	493	3.68	11.4	93	12.40	20.1
30	723	5.29	12.6	123	28.17	40.4
50	1183	8.59	17.4	183	78.30	86.9
100	2333	16.76	28.0	333	339.52	314.9
200	4633	34.45	44.3	633	1494.35	1233.8
400	9233	66.03	77.1	N/A	>7000	>4180.6

units), the reason being that the light level is increased only by one before he gives up, and starts the hold action again. Obtaining coverage therefore requires many interactions (trace of length 828). In contrast, the optimal execution time for the patient user is 183 time units (and the trace length is 130). If we compare the generation time, it can be seen that it is much cheaper to compute the (very long) optimal solution for the impatient user than to compute the (short) optimal solution for the patient user.

Although this is surprising, there is a potential general explanation for this. The patient user environment poses no restrictions on the solution, and the test generator has complete freedom to find the optimal solution. This means that test generator has to evaluate all possible behaviors of this liberal environment. The impatient user is a more restricted environment, thus containing less possible behaviors. Therefore, searching the more liberal environment takes longer but also produces faster solutions.

There are two lessons to be learned. First, the relevance of an accurate model of the environment assumptions. Secondly, the use of the environment model to control test generation: restrict the environment to handle larger systems, but at the cost of more expensive solutions.

We have also created a DIEOU-TA version of the Philips audio control protocol [3] frequently studied in the context of model checking. The system consists of a sender and a receiver communicating over a shared bus. The sender inputs a sequence of bits to be transmitted, Manchester encodes them, and transmits them as high and low voltage on

Table 10. Results for the Philips audio protocol

Coverage Criterion	Execution time (μ s)	Generation time (s)	Memory usage (KB)
Edge _{Sender}	212350	2.2	9416
Edge _{Receiver}	18981	1.2	4984
Edge _{Sender,Bus,Receiver}	114227	129.0	331408

Table 11. Cost of edge coverage of TSS ($Max=30$) using different search orders

Search order	First Solution			Optimal Solution	
	Execution time	Generation time (s)	Memory usage (MB)	Generation time	Memory usage (MB)
BF	123	27.91	40.8	N/A	N/A
DF	791	0.15	4.9	N/A	N/A
C_BF	123	30.44	42.6	31.31	43.3
C_DF	791	0.15	6.5	248.64	127.0
C_BF_R	123	30.70	42.6	30.87	42.9
C_DF_R	791	0.15	6.4	21.62	32.1
C_MC	123	25.87	39.3	26.19	39.5
C_MC_R	123	3.23	13.0	3.32	13.1

the bus. Further, it checks for collisions by checking that the bus is indeed low when it is itself sending a low signal. The receiver is triggered by low-to-high transitions on the bus, and decodes the bits based on this information.

Table 10 summarizes the results. The first row contains results for the protocol tested with an environment consisting of a bus that may spontaneously go high to emulate a collision, and a sender buffer producing any legal input-bit sequence. The second row shows results for a receiver tested in an environment consisting of a bus, and a buffer to hold the received bits. The third row is the results for the receiver tested in an environment consisting of a sender component with sender buffer, a bus, and receiver buffer. Thus the last row represents a rather large system. In all cases the time optimal covering test sequence could be computed in reasonable time.

5.3 Search-Order and Guiding

UPPAAL allows the state space to be traversed in several different orders with different performance characteristics w.r.t. execution time of the generated test suite and the size of the system that can be handled. In particular, the A^* algorithm has potential significant impact. We here demonstrate how it can be employed for test generation to efficiently compute edge coverage in the TSS model.

The measured numbers are listed in Table 11. BF (DF) denotes breadth-first (depth-first) search order. The optimal execution time remains identical at 123 time units for all search orders. We note that using depth-first search during time optimal analysis (C_DF) UPPAAL produces (many) solutions quickly, but consumes long time to ultimately find the optimal one. During time optimal reachability analysis UPPAAL (symbolically) computes for each reached state the time C accumulated so far. Let C_g be the fastest

time to a goal state found so far. When another state is found during exploration with an accumulated time $C \geq C_g$ further exploration from that state is unnecessary, and the search can be pruned. Minimum accumulated time-first (MC) explores states ordered by their minimum accumulated time. To increase the efficiency further, it is possible to provide a safe estimate of the time that remains R from a given state to the goal state. Pruning can then be performed when a state is found with $C + R \geq C_g$. In Table 11 a search order combined with a remaining estimate is suffixed by an “_R”.

It is easy to see in the dimmer component that the most time consuming edge to reach is the edge with guard $L = Max$. As estimate of remaining time, we use $(Max - L) \times delay$ if level $Max = L$ has not been reached, and 0 otherwise. Intuitively, the remaining time equals at least the number of light levels from Max value times the time to increase the light one level ($delay$). This formula has the feature that it can prune searches that turns back to lower light levels.

Compared to C_BF minimum accumulated time first search (C_MC) offers slightly improved generation time and memory usage. However, enabling remaining time estimate combined with this search order (C_MC_R) has a dramatic positive effect, and outperforms any of the other evaluated search orders.

6 Conclusions and Future Work

In this paper, we have presented a new technique for generating timed test sequences for a restricted class of timed automata. It is able to generate time optimal test sequences from either a single test purpose or a coverage criterion using the time optimal reachability feature of UPPAAL. Though a number of examples we have demonstrated how our technique works and performs. We conclude that it can generate practically relevant test sequences for practically relevant sized systems. However, we have also found a number of areas where our technique can be improved.

The DIEOU-TA model is quite restrictive, and a generalization will benefit many real-time systems. We are working on loosening the output urgency requirement. It may also be interesting to formulate coverage criteria that considers clock constraints.

Adding the required annotations for various coverage criteria by hand, and manually formulating the associated reachability property is tedious and error prone. We are working on a tool that performs these tasks automatically. Finally, we have found that the bit-vector annotations for tracking coverage and remaining time estimates may increase the state space significantly, and consequently also generation time and memory. The extra bits does not influence model behavior, and should therefore be treated differently in the verification engine. We are working on techniques that ignore these bits when possible, and that takes advantage of them to prune states with “less” coverage.

References

1. Alfred V. Aho, Anton T. Dahbura, David Lee, and M. Ümit Uyar. An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours. *IEEE Transactions on Communications*, 39(11):1604–1615, 1991.

2. Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim G. Larsen, Paul Pettersson, and Judi Romijn. Efficient Guiding Towards Cost-Optimality in UPPAAL. In T. Margaria and W. Yi, editors, *Proc. of TACAS 2001*, number 2031 in Lecture Notes in Computer Science, pages 174–188. Springer–Verlag, 2001.
3. D. Bosscher, I. Polak, and F. Vaandrager. Verification of an Audio-Control Protocol. In *Proc. of Formal Techniques in Real-Time and Fault-Tolerant Systems*, number 863 in Lecture Notes in Computer Science, 1994.
4. Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Claude Jard, Thierry Jéron, Alain Kerbrat, Pierre Morel, and Laurent Mounier. Verification and Test Generation for the SSCOP Protocol. *Science of Computer Programming*, 36(1):27–52, 2000.
5. Rachel Cardell-Oliver. Conformance Testing of Real-Time Systems with Timed Automata. *Formal Aspects of Computing*, 12(5):350–371, 2000.
6. R. Castanet, Ousmane Koné, and Patrice Laurençot. On The Fly Test Generation for Real-Time Protocols. In *International Conference in Computer Communications and Networks*, Lafayette, Louisiana, USA, October 12-15 1998. IEEE Computer Society Press.
7. Duncan Clarke and Insup Lee. Automatic Test Generation for the Analysis of a Real-Time System: Case Study. In *3rd IEEE Real-Time Technology and Applications Symposium*, 1997.
8. Abdeslam En-Nouaary, Rachida Dssouli, Ferhat Khendek, and A. Elqortobi. Timed Test Cases Generation Based on State Characterization Technique. In *19th IEEE Real-Time Systems Symposium (RTSS'98)*, pages 220–229, December 2–4 1998.
9. Teruo Higashino, Akio Nakata, Kenichi Taniguchi, and Ana R. Cavalli. Generating Test Cases for a Timed I/O Automaton Model. In Gyula Csopaki, Sarolta Dibuz, and Katalin Tarnay, editors, *Testing of Communicating Systems: Method and Applications, IFIP TC6 12th International Workshop on Testing Communicating Systems (IWTCS), September 1-3, 1999, Budapest, Hungary*, volume 147 of *IFIP Conference Proceedings*, pages 197–214. Kluwer, 1999.
10. Hyoungh Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A Temporal Logic Based Theory of Test Coverage and Generation. In J.-P. Katoen and P. Stevens, editors, *TACAS 2002*, pages 327–341. Kluwer Academic Publishers, April 2002.
11. Ahmed Khoumsi. A method for testing the conformance of real-time systems. In Werner Damm and Ernst-Rüdiger Olderog, editors, *IEEE International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems. (FTRTFT)*, volume 2469 of *LNCS*. Springer-Verlag, september 2002.
12. Kim G. Larsen, Gerd Behrmann, Ed Brinksma, Ansgar Fehnker, Thomas Hune, Paul Pettersson, and Judi Romijn. As cheap as possible: Efficient cost-optimal reachability for priced timed automata. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. of CAV 2001*, number 2102 in Lecture Notes in Computer Science, pages 493–505. Springer–Verlag, 2001.
13. David Lee and Mihalis Yannakakis. Principles and Methods of Testing Finite State Machines—A Survey. *Proceedings of the IEEE*, 84(8):1090–1123, august 1996.
14. Dino Mandrioli, Sandro Morasca, and Angelo Morzenti. Generating Test Cases for Real-Time Systems from Logic Specifications. *ACM Transactions on Computer Systems*, 13(4):365–398, 1995.
15. Brian Nielsen and Arne Skou. Automated Test Generation from Timed Automata. *International Journal on Software Tools for Technology Transfer (STTT)*, 4, 2002. Digital Object Identifier (DOI) 10.1007/s10009-002-0094-1. To appear.
16. Jan Peleska. Hardware/Software Integration Testing for the new Airbus Aircraft Families. In A. Wolis I. Schieferdecker, H. König, editor, *Testing of Communicating Systems XIV. Application to Internet Technologies and Services*, pages 335–351. Kluwer Academic Publishers, 2002.

17. J. Springintveld, F. Vaandrager, and P.R. D'Argenio. Testing Timed Automata. *Theoretical Computer Science*, 254(1-2):225–257, March 2001.
18. J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99: 7th European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 8–12, 1999. EuroStar Conferences, Galway, Ireland.
19. M. Ümit Uyar, Marius A. Fecko, Adarsphal S. Sethi, and Paul D. Amar. Testing Protocols Modeled as FSMs with Timing Parameters. *Computer Networks: The International Journal of Computer and Telecommunication Networking*, 31(18):1967–1998, 1999.

Test Cases Generation for Nondeterministic Real-Time Systems

Ahmed Khoumsi^{1,*}, Thierry Jéron², and Hervé Marchand²

¹ University of Sherbrooke, Canada
Ahmed.Khoumsi@USherbrooke.ca

² IRISA, Campus de Beaulieu, France

Abstract. We study the generation of test cases for nondeterministic real-time systems. We define a class of Determinizable Timed Automata (DTA), in order to specify the system under test. The principle of our test method consists of two steps. In Step 1, we express the problem in a non-real-time form, by transforming a DTA into an equivalent finite state automaton. The latter uses two additional types of events, *Set* and *Exp*. In Step 2, we adapt a non-real-time test generation method.

Keywords: Real-time systems, Conformance test cases generation, Determinizable Timed Automata (DTA), Set-Exp-Automata (SEA), Test architecture.

1 Introduction

(*Conformance*) *testing* aims to check whether an implementation conforms to a specification. Testing activity is realized by: 1) *synthesizing* test cases from the specification, and 2) *executing* them on the implementation under test (**IUT**). We study especially the *synthesis phase*, but we also propose a test architecture for the *execution phase*. We consider the case of real-time systems, where the specification describes *order* and *timing* constraints of the interactions between **IUT** and its environment. For developing a rigorous test method, the specification of **IUT** must be described in a formal way, e.g. by Timed automata (TA) [1]. We define a class of *determinizable* TA (DTA), in order to describe the specification of **IUT**. The principle of our test method is as follows:

- First, we express the problem into a non-real-time form, by using a method that transforms a DTA into a finite-state-automaton (FSA), called Set-Exp-Automaton (SEA) [2]. The latter uses two additional types of events: *Set* and *Exp*. Intuitively, a *Set* (resp. *Exp*) event corresponds to the programming (resp. occurrence) of an alarm. Let *SetExp* denote the transformation operation, and *SetExp*(*A*) denote the SEA obtained by transformation of a DTA *A*. *A* and *SetExp*(*A*) are *equivalent* in the sense that they specify the same order and timing constraints of events.
- Then, we use and adapt a non-real-time method of test generation, called *Test Generation with Verification technology* (TGV) [3]¹.

* Visiting at IRISA from Sept.2002 to Sept. 2003

¹ Actually, TGV is a software tool. But here, TGV denotes the theoretical method that underlies the tool.

The test problem we have to solve is to synthesize test cases that allow (by executing them) to check the conformance of a real-time **IUT** to its specification. The rest of the paper is organized as follows. Sect. 2 describes the DTA model used to describe the specification of **IUT**. In Sect. 3, we define the test problem to be solved. Sect. 4 presents the SEA model and “*SetExp*: $DTA \mapsto SEA$ ” used by the test method of Sect. 6. In Sect. 5, we propose a test architecture, and present a proposition that allows to transform the test problem into a non-real-time form. Sect. 6 presents a method that solves the test problem. And in Sect. 7, we conclude the paper.

2 Determinizable Timed Automata (DTA)

Let us present the DTA model used to describe **IUT** and its specification.

2.1 A Few Concepts Used in DTA

Nondeterminism: Nondeterministic systems are systems whose current state is not necessarily determined by observing their execution. In our study, the specification of **IUT** can be nondeterministic.

Internal actions are actions that are unobservable by the environment of **IUT**. Note that such unobservability may cause nondeterminism. In our study, the specification of **IUT** can contain internal actions.

Forced transitions: In a real-time system, a first objective of timing constraints is to define intervals of time where an action *can* occur. The notion of *forced transition* allows to define intervals of time where an action *must* occur.

Quiescence: Basically, conformance testing allows to check that **IUT** executes only what is permitted by its specification. The notion of *quiescence* has been added to specify when **IUT** is allowed to stop its execution, and thus, to become quiescent.

2.2 A Class of Determinizable Timed Automata (DTA)

A clock c_i is a real variable whose value can be reset with the occurrence of an event and such that, between two resets, its derivative (w.r.t. time) is equal to 1. Let $\mathcal{C} = \{c_1, \dots, c_{N_c}\}$ be a set of clocks.

A Clock Condition (CC) consists of one or several formulas in the form “ $c_i \sim k$ ” where c_i is a clock, $\sim \in \{<, >, \leq, \geq, =\}$ and $k \geq 0$. Let $\Phi_{\mathcal{C}}$ denote the set of CCs depending of clocks of $\mathcal{C}$ (we consider that $True \in \Phi_{\mathcal{C}}$).

A clock reset is any subset of $\mathcal{C}$, and $2^{\mathcal{C}}$ denotes the set of clock resets.

Syntax of DTA. A *DTA* is defined by $(\mathcal{L}, \Sigma, \mathcal{C}, \mathcal{T}, l_0)$, where: $\mathcal{L}$ is a finite set of locations, l_0 is the initial location, $\mathcal{C}$ is a finite set of clocks, and Σ is a finite set of events. There are three types of events: the reception of an *input* i (written $?i$), the sending of an *output* o (written $!o$), and the occurrence of an *internal action* a (written ϵ_a). $\mathcal{T}$ is a set of transitions, and a transition of DTA is defined

by $\text{Tr} = \langle q; \sigma; r; CC; Z_\sigma; x \rangle$, where: q and r are origin and destination locations, σ is an event of, CC is a clock condition, Z_σ is a clock reset, and $x \in \{f, uf\}$. Tr is said *forced* (resp. *unforced*) if $x = f$ (resp. $x = uf$) (see semantics of DTA for more details).

The only restrictions used to define DTA from TA are: 1) the index σ in Z_σ means that all the transitions (from any locations) that have the same event, have also the same clock reset; and 2) Z_σ is empty when σ is an internal action.

Semantics of DTA. Let us define the semantics of a DTA $A = (\mathcal{L}, \Sigma, \mathcal{C}, \mathcal{T}, l_0)$ by the set of timed traces accepted by A . The term “sequence of transitions $\text{Tr}_1 \text{Tr}_2 \dots \text{Tr}_i \dots$ of A ” means: l_0 is the origin location of Tr_1 , and the destination location of Tr_i is the origin location of Tr_{i+1} . We consider that events are instantaneous, i.e., their duration is negligible and can be assumed equal to zero.

Notation 1 Let X be a set of sequences and $X_f (\subseteq X)$ be the set of finite sequences of X . $\overline{X}$ denotes the set of finite prefixes of X . Note that $X_f \subseteq \overline{X}$.

Enabled/disabled transition: a transition $\text{Tr} = \langle q; \sigma; r; CC; Z_\sigma; x \rangle$ is said *enabled* when q is the current location and CC evaluates to *True*. Tr is said *disabled* when it is not enabled.

Execution of event: for every $\text{Tr} = \langle q; \sigma; r; CC; Z_\sigma; x \rangle$ of A , the event σ is executed only when Tr is enabled; and after the execution of σ , location r is reached and the clocks in Z_σ are reset.

A timed sequence is a (finite or infinite) sequence “ $(e_1, \tau_1) \dots (e_i, \tau_i) \dots$ ”, where e_i are events, each τ_i is the time of occurrence of e_i , and $0 < \tau_1 < \dots < \tau_i < \dots$.

A timed trace is obtained from a timed sequence by removing all the timed internal events, i.e., the (e_i, τ_i) such that e_i is an internal event.

At initial time $\tau_0 = 0$, A is at location l_0 with all clocks equal to 0.

Acceptance of the empty sequence λ_0 : A accepts λ_0 iff every outgoing transition of l_0 is unforced, or disabled when all clocks are equal with each other.

Intuitively, acceptance of λ_0 by A means that A can be “totally quiescent”.

Acceptance of a finite timed sequence $\lambda_n = (e_1, \tau_1) \dots (e_n, \tau_n)$, for $e_1, \dots, e_n \in \Sigma$. Let λ_i be the prefix of λ_n of length i , for $1 \leq i < n$. λ_n is *accepted* by A iff there exists a sequence of transitions $\text{Tr}_1 \text{Tr}_2 \dots \text{Tr}_n$ of A s.t.:

1. $\forall i \in \{1, \dots, n\}$: the event of Tr_i is e_i ; after the execution of λ_{i-1} , Tr_i is enabled at time τ_i ; and no forced transition is enabled at a time $\tau \in]\tau_{i-1}, \tau_i[$ and disabled at τ_i . Intuitively, A can execute λ_n .
2. $\forall$ forced transition Tr and $\forall u > \tau_n$ s.t. Tr is enabled at time u after the execution of λ_n , we have: $\forall \tau \geq u$, Tr is enabled at time τ .

Intuitively, A can be “quiescent forever” after the execution of λ_n .

Acceptance of an infinite timed sequence λ_∞ is defined from acceptance of finite timed sequence by: removing Item 2 and replacing n by ∞ . Intuitively, A can execute λ_∞ .

Acceptance of a timed trace $\mu = (e_1, \tau_1)(e_2, \tau_2) \cdots$. μ is accepted by A iff μ is obtained by removing all the timed internal actions of a timed sequence accepted by A . Intuitively, A can have an execution observed as μ .

The timed observable language of A (TOL_A^{DTA}) is the set of timed traces accepted by A . That is, TOL_A^{DTA} models the observable behaviour of A .

The timed quiescence observable language of A ($TQOL_A^{DTA}$) is obtained from TOL_A^{DTA} by keeping only the *finite* timed traces. Intuitively, after the observation of $\mu \in TQOL_A^{DTA}$, A accepts to be quiescent forever, i.e., we may wait forever without observing any event. Formally: $TQOL_A^{DTA} = TOL_A^{DTA} \cap \overline{TOL_A^{DTA}}$ (see Notation 1).

Hypothesis 1 *We assume that infinite timed sequences (and traces) accepted by A are non-zeno, i.e., have an infinite duration, because zeno executions do not correspond to concrete systems.*

In figures, locations are represented by nodes, a transition $\text{Tr} = \langle q; \sigma; r; CC; \{c_i, c_j, \dots\}; x \rangle$ is represented by an arrow linking q to r and labelled by $(\sigma; CC; \{c_i, c_j, \dots\})$, and the absence of CC or of clock reset is indicated by “-”. x is not represented in figures; when relevant, we mention whether a given transition is forced or not. Let us consider the DTA of Fig. 1a. We do not indicate the types of events (i.e., input, output, internal action) because they are irrelevant for the comprehension of the example. Let $\delta_{u,v}$ denote the delay between events u and v .

- The DTA is initially in location l_0 . It reaches l_1 at the occurrence of α .
- From l_1 , the DTA reaches l_2 or l_3 at the occurrence of β . l_2 (resp. l_3) is reached only if $\delta_{\alpha,\beta} < 3$ (resp. $\delta_{\alpha,\beta} > 2$). We see that there is a nondeterminism when $2 < \delta_{\alpha,\beta} < 3$.
- From l_2 , the DTA reaches l_0 at the occurrence of γ . We have $\delta_{\beta,\gamma} > 1$ and $\delta_{\alpha,\gamma} < 2$.
- From l_3 , the DTA reaches l_0 at the occurrence of ρ . We have $\delta_{\beta,\rho} > 1$ and $\delta_{\alpha,\rho} \geq 2$.

To clarify the notion of *forced transition*, let us consider the case where location l_2 is reached while $c_1 > 1$ and $c_2 < 2$, i.e., γ is enabled as soon as l_2 is reached. Then γ : either 1) will occur before $c_2 = 2$, or 2) will never occur. Case 2 is impossible if γ is in a forced transition.

The DTA of Fig. 1a is nondeterministic, and its equivalent deterministic DTA is in Fig. 1b.

3 Test Problem to Be Solved

3.1 Conformance Relation between DTA, Some Related Lemmas

In the following: I and S are two DTAs over the same alphabet Σ , and o is an output of Σ . The following conformance relation is an adaptation of **io**co of [4,3] to the real-time case.

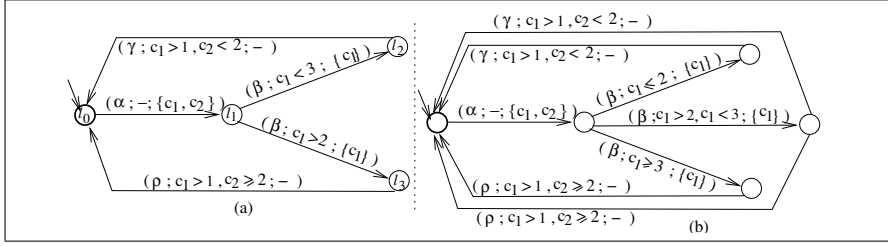


Fig. 1. Example: (a) nondeterministic DTA, and (b) equivalent deterministic DTA

Definition 1 $I \text{ ioco}_{DTA} S$ means, $\forall \lambda \in \overline{TOL}_S^{DTA}$:

- 1) $(\lambda \cdot (o, \tau) \in \overline{TOL}_I^{DTA}) \Rightarrow (\lambda \cdot (o, \tau) \in \overline{TOL}_S^{DTA})$, and
- 2) $(\lambda \in TQOL_I^{DTA}) \Rightarrow (\lambda \in TQOL_S^{DTA})$.

Let **IUT** be modelled by I . The intuition of “ $I \text{ ioco}_{DTA} S$ ” is that after an execution of **IUT** accepted by S : 1) **IUT** can generate an output o at time τ only if S accepts o at time τ , and 2) **IUT** can remain quiescent forever only if S accepts to be quiescent forever.

Definition 2 The input-completion of a DTA $A = (\mathcal{L}, \Sigma, \mathcal{C}, \mathcal{T}, l_0)$ is a DTA $\text{InpComp}(A)$, that contains all the timed traces of A , as well as all the timed traces that diverge from the timed traces of A by executing inputs not accepted by A^2 . A is said input-complete iff $A = \text{InpComp}(A)$. Intuitively, an input-complete DTA accepts every input at any time.

Hypothesis 2 We consider without loss of generality, that input transitions are always unforced, because they are under the control of the environment.

Lemma 1³ We have: $(I \text{ ioco}_{DTA} S) \Leftrightarrow (I \text{ ioco}_{DTA} \text{InpComp}(S))$.

Lemma 3 implies that we can make S input-complete before to check whether a DTA conforms to it, w.r.t. ioco_{DTA} . Hence the following hypothesis:

Hypothesis 3 With Lemma 3, when we use ioco_{DTA} we can (and will) consider, without loss of generality, that S is input-complete.

Lemma 3 and Hyp. 3 are inspired from their non-real-time versions in [5].

Lemma 2 With Hypothesis 3: $I \text{ ioco}_{DTA} S \Leftrightarrow \overline{TOL}_I^{DTA} \subseteq \overline{TOL}_S^{DTA}$.

Lemma 2 means that with Hypothesis 3, ioco_{DTA} is simplified into an inclusion of timed observable languages of DTA.

² For lack of space, we do not give a formal construction.

³ All lemmas and propositions of this article have been rigorously proved. For lack of space, the proofs are not presented.

3.2 Test Purpose, and Test Hypothesis

A *complete* DTA has all its events enabled at any time from any location. A *trap* location is a location in which, for each event $\sigma \in \Sigma$, there is a selfloop labelled $(\sigma; -; -)$.

Definition 3 A test purpose is used to select a particular functionality of **IUT** to be tested. In our study, a test purpose is modelled by a complete and deterministic DTA TP , equipped with two sets of trap locations A and R (for Accept and Refuse, respectively). All transitions of TP are unforced. Sequences to be tested are those terminating in a location A . Sequences not to be tested are those traversing a location R .

A test purpose allows to extract a relatively small part of the specification before to apply a test generation method [3]. We use the following usual *test hypothesis*:

Hypothesis 4 **IUT** can be described by a (possibly unknown) input-complete DTA $\mathcal{IUT}$.

3.3 Formalization of the Test Generation Problem

The *inputs* of the test generation problem are two DTA $Spec$ and TP , describing the specification and the test purpose, resp., over the same alphabet that consists of inputs, outputs and internal actions. The *test problem* is to *synthesize test cases* that will be executed on **IUT** to determine whether: $\mathcal{IUT} \text{ ioco}_{DTA} Spec$.

The test purpose TP means that the test system will *ignore* every execution λ of **IUT** (i.e., $\lambda \in TOL_{\mathcal{IUT}}^{DTA}$) that has a prefix μ accepted by $Spec$ (i.e., $\exists \mu \in \{\lambda\} \cap TOL_{Spec}^{DTA}$) and s.t.: a location R of TP may be reached by μ , or no location A of TP is reachable after μ by $Spec$.

By using Lemma 3 and Hyp. 3, we assume $Spec$ input-complete. Let us present, in Section 4, the SEA model and the transformation “ $SetExp: DTA \rightarrow SEA$ ” that will be used by the test method of Section 6.

4 Transformation of DTA into SEA

We present $SetExp$ that transforms a DTA A into a FSA called Set-Exp-Automaton (SEA) and denoted $SetExp(A)$. The latter uses two types of events: Set and Exp , in addition to the events of A . The DTA A and the SEA $SetExp(A)$ are *equivalent* because they specify exactly the same order and timing constraints (of events other than Set and Exp) [2].

4.1 Events Set and Exp

Event $Set(c_i, k)$ means: clock c_i is set to zero and will expire when its value is equal to k .

Event $Exp(c_i, k)$ means: clock c_i expires and its current value is k .

Therefore, $Set(c_i, k)$ is followed (after a delay k) by $Exp(c_i, k)$.

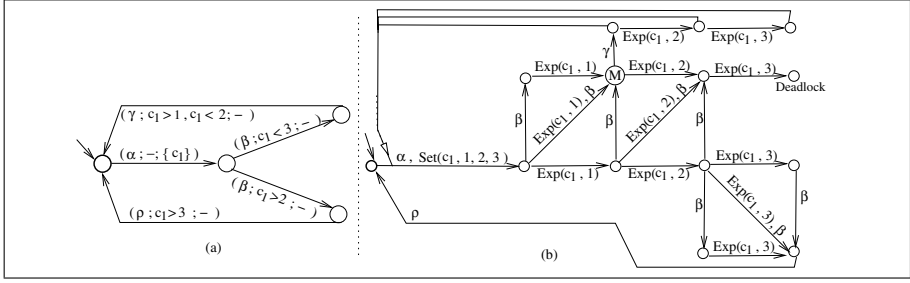


Fig. 2. Example: (a) DTA, and (b) corresponding SEA

4.2 Transitions of SEA

In the following: σ denotes an event of the alphabet of the DTA A , $\mathcal{S}$ (resp. $\mathcal{E}$) denotes a set of *Set* (resp. *Exp*) events, and *occurrence of $\mathcal{S}$* (resp. $\mathcal{E}$) means the simultaneous occurrences of all the events of $\mathcal{S}$ (resp. $\mathcal{E}$). Here are the three types of transitions of a SEA $SetExp(A)$:

Type 1: a transition labelled $(\mathcal{E})$ represents the occurrence of $\mathcal{E}$.

Type 2: a transition labelled (σ) or $(\sigma, \mathcal{S})$: (σ) represents the occurrence of σ , and $(\sigma, \mathcal{S})$ represents the simultaneous occurrences of σ and $\mathcal{S}$.

Type 3: a transition labelled $(\mathcal{E}, \sigma)$ or $(\mathcal{E}, \sigma, \mathcal{S})$: $(\mathcal{E}, \sigma)$ represents the simultaneous occurrences of $\mathcal{E}$ and σ , and $(\mathcal{E}, \sigma, \mathcal{S})$ represents the simultaneous occurrences of $\mathcal{E}$, σ and $\mathcal{S}$.

Definition 4 An *Exp-Trans* of B is any transition of type 1 or 3. i.e., whose label contains one or several *Exp* events.

4.3 Example of Transformation “*SetExp*: DTA $\rightarrow$ SEA”

For the DTA A of Fig. 2a, if all transitions are unforced, then we obtain the SEA $SetExp(A)$ of Fig. 2b. The concept of *forced transition* was not considered in [2]; let us explain its influence on the result of *SetExp*. A transition of TR (of Type 2 or 3) of $SetExp(A)$ that corresponds to a forced transition Tr of A is also called *forced transition*. If a state q of $SetExp(A)$ has an outgoing forced transition $TR1$ which becomes disabled with the occurrence of an *Exp-Trans* $TR2$, then $TR1$ must preempt $TR2$, and thus, $TR2$ never occurs. For every state q , we define the set $PreemptExp(q)$ which contains all the labels of the preempted *Exp-Trans* in q .

For the example of Fig. 2, if γ is in a forced transition, then the outgoing $Exp(c_1, 2)$ of state M is cut and we have $PreemptExp(M) = \{Exp(c_1, 2)\}$. Concretely, since $Exp(c_1, 2)$ leads to a state where (the forced) γ is disabled, such $Exp(c_1, 2)$ must be cut, which means that it is preempted by γ .

4.4 Languages and Quiescences of SEA

Let A be a DTA and $B = SetExp(A)$ be a SEA of alphabet Λ and origin state q_0 . Let us define the semantics of B by the set of traces accepted by B .

A final state q of B is a state that: - has no outgoing transitions, such q is called *deadlock state*; or - has neither outgoing Exp-Trans nor preempted Exp-Trans, such q is called *patient state*. Intuitively, B can remain forever in a final state without executing any event.

A sequence is any (finite or infinite) sequence “ $E_1 E_2 \dots E_i \dots$ ”, where $E_i \in \Lambda$.

A trace is obtained by removing all internal actions of a sequence.

Acceptance of finite sequence λ_n : λ_n is accepted by B iff it is a sequence of B that terminates in a final state. Intuitively, B can execute λ_n and become “quiescent”.

Acceptance of infinite sequence λ_∞ : λ_∞ is accepted by B iff it is a sequence of B .

Acceptance of trace μ : μ is accepted by B iff μ is obtained by removing the internal actions of a sequence accepted by B . Intuitively, B can have an execution observed as μ .

The observable language of B (OL_B^{SEA}) consists of the traces accepted by B .

The quiescent observable language of B (QOL_B^{SEA}) consists of the *finite* traces accepted by B .

Consistency condition requires that every $Set(c, k)$ and the corresponding $Exp(c, k)$ are separated by a delay k . It is implicitly respected in OL_B^{SEA} and QOL_B^{SEA} .

Lemma 3 *Hypothesis 1 implies that every (finite) $\mu \in QOL_B^{SEA}$ is obtained by removing all internal actions from a finite sequence accepted by B .*

It is clear that when we are in a final state, then B can remain quiescent forever. Lemma 3 states that the converse holds assuming Hypothesis 1. More precisely, Hypothesis 1 implies that if B is quiescent for an arbitrarily long time, then we are in a final state or we are switching between final states through internal actions. Therefore:

Definition 5 *Final states will be also called quiescent states.*

5 Test Architecture, and a Proposition

We use the test architecture of Fig. 3 proposed in [6] and consisting of the following modules:

Clock-Handler receives *Set* events and sends *Exp* events. (It respects consistency condition.)

Test-Controller sends inputs to **IUT**, receives outputs from **IUT**, sends *Set* events to Clock-Handler, and receives *Exp* events from Clock-Handler.

The following relation $\mathbf{conf}_{SEA}$ is simply an inclusion of observable languages of SEAs.

Definition 6 *Let I' and S' be two SEAs over the same alphabet: $(I' \mathbf{conf}_{SEA} S') \Leftrightarrow (OL_{I'}^{SEA} \subseteq OL_{S'}^{SEA})$.*

We have the following proposition, where **SUT** consists of **IUT** and Clock-Handler.

Proposition 1 *Let S be an input-complete DTA. Assuming that Test-Controller generates Set events only when they are accepted by $SetExp(S)$, we have:*
 $(\mathcal{IUT} \text{ ioco}_{DTA} S) \Leftrightarrow (\exists \text{ SEA } SUT \text{ accepting behaviour of } \mathbf{SUT} \text{ s.t. } SUT \text{ conf}_{SEA} SetExp(S)).$

The above proposition implies that we can check “ $SUT \text{ conf}_{SEA} SetExp(S)$ ” instead of “ $\mathcal{IUT} \text{ ioco}_{DTA} S$ ”. We have transformed the test problem into a non-real-time form, and thus, we will adapt a non-real-time method of Test Generation with Verification technology (TGV) [3].

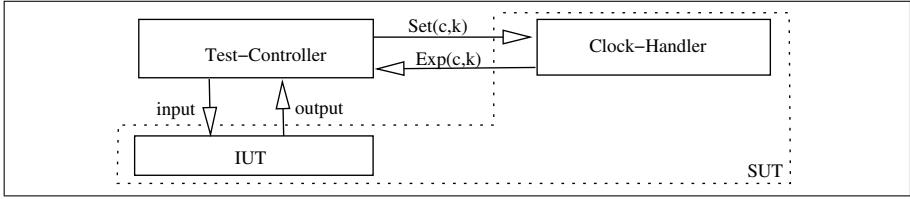


Fig. 3. Test architecture

Note that this architecture implies that Set (resp. Exp) events are inputs (resp. outputs) of **SUT**.

Remark 1 *Recall that in the DTA model, internal (i.e., unobservable) actions do not reset clocks. Interestingly, this restriction is also required by the proposed architecture. In fact, in order to generate Set events, Test-Controller must observe every event to which is associated a clock reset.*

6 Synthesis of Test Cases

Our test method consists of six steps outlined in the diagram of Fig. 4 and described in subsections 6.1 to 6.6. Its inputs are $Spec$ and TP . In a first step, we compute a DTA $SpecTP$ equivalent to $Spec$ such that locations of $SpecTP$ that correspond to locations A (resp. R) of TP are denoted A (resp. R). Then, we synthesize in five steps a set of test cases that can be used to determine whether: $\mathcal{IUT} \text{ ioco}_{DTA} SpecTP$. The indication A and R is used to ignore every execution of **IUT** that leads to a location R or does not allow to reach a location A .

The fact that TP is deterministic and complete implies that $Spec$ is input-complete iff $SpecTP$ is input-complete. By using Lemma 3 and Hyp. 3, we assume $Spec$ input-complete.

Figure 5 represents $Spec$ and TP of alphabet $\Sigma = \{\phi, \sigma, \rho, \epsilon_a, \epsilon_b\}$ used to illustrate the six steps of the test method. $\neq \sigma$ denotes any event $\in \Sigma \setminus \{\sigma\}$. Actually, $Spec$ was not initially input-complete and we represent by dotted arrows the part that has been added to make $Spec$ input-complete. The test purpose means that: we are interested to test executions of $Spec$ terminating by the first occurrence of $!\rho$ without traversing Location TL .

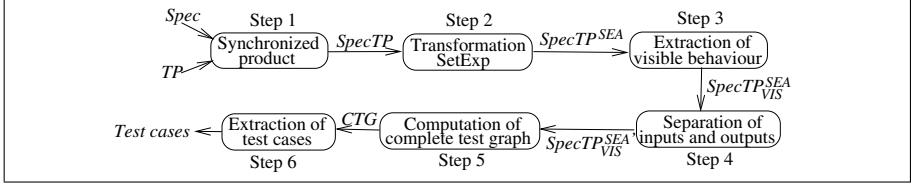


Fig. 4. Steps of the test method

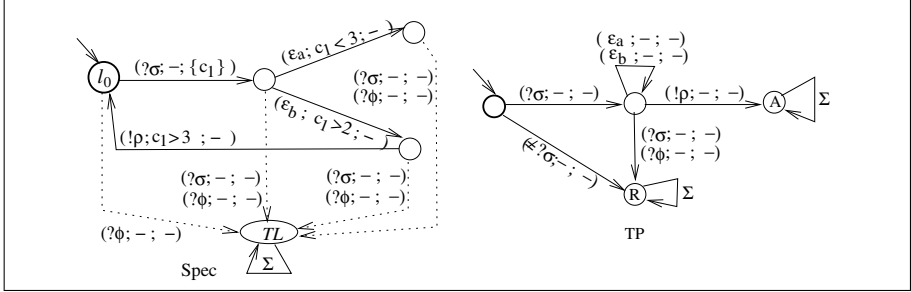


Fig. 5. Example for illustrating the test method

6.1 Step 1: Synchronous Product of *Spec* and *TP*

Definition 7 Let $A_i = (\mathcal{L}_i, \Sigma_i, \mathcal{C}_i, \mathcal{T}_i, l_{0i})$, for $i = 1, 2$, be two DTAs. The synchronous product of A_1 and A_2 , written $A_1 \otimes A_2$, is quite similar to the classical product of TA. The only difference is due to the notion of forced transitions which we process as follows: a transition of $A_1 \otimes A_2$ is unforced iff it is a “synchronization” of two unforced transitions of A_1 and A_2 ⁴.

We compute $SpecTP = Spec \otimes TP$. Locations of $SpecTP$ that correspond to locations A (resp. R) of TP are denoted A (resp. R). The fact that TP is complete implies that $Spec$ and $SpecTP$ are observationally equivalent (i.e., $TOL_{Spec}^{DTA} = TOL_{SpecTP}^{DTA}$). The effect of $Spec \otimes TP$ is to determine in $Spec$ the executions that correspond to locations A and R , respectively. For $Spec$ and TP of Fig. 5, we obtain the $SpecTP$ of Fig. 6. Locations L_0 and (l_0, A) are equivalent in the sense that $Spec$ is executable from these locations. The difference between these two locations is that only (l_0, A) corresponds to Location A of TP .

6.2 Step 2: Transforming the DTA $SpecTP$ into a SEA

We compute $SpecTP^{SEA} = SetExp(SpecTP)$. For the $SpecTP$ of Fig. 6, we obtain the $SpecTP^{SEA}$ of Fig. 7, if no transition is forced.

- Quiescent states of $SpecTP^{SEA}$ that correspond to locations A are denoted A . We consider only *quiescent* states, because sequences to be tested are those *terminating in* Location A .

⁴ For lack of space, we do not give a more detailed definition.

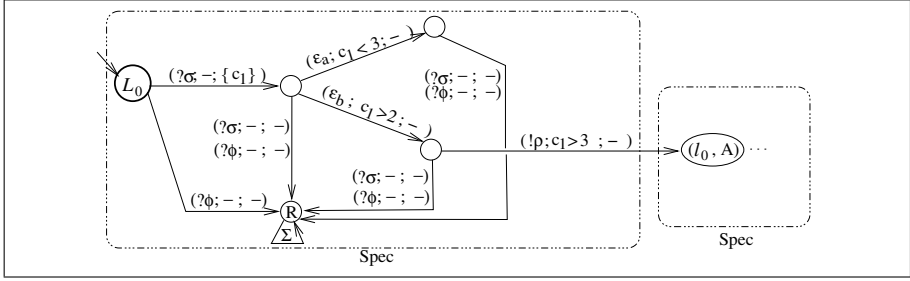


Fig. 6. Step 1: $SpecTP$ obtained from $Spec$ and TP of Fig. 5

- States of $SpecTP^{SEA}$ that correspond to locations R are denoted R . We consider *quiescent* and *non-quiescent* states, because sequences not to be tested are those *traversing* (and thus, not necessarily terminating in) Location R .

In Fig. 7: $?*$ denotes any input (i.e., $?φ$ or $?σ$); $Σ$ means any event $x \in Σ = \{?φ, ?σ, !ρ, ε_a, ε_b\}$; $!Exp_i$ denotes $!Exp(c_1, i)$ for $i = 2, 3$; $(!Exp_i, Σ)$ means the simultaneous occurrence of $!Exp_i$ and any $x \in Σ$; quiescent states are indicated by q ; nodes linked by a dotted line correspond to the same state; and State A is equivalent to the original state s_0 with the difference that s_0 does not correspond to a location A of TP .

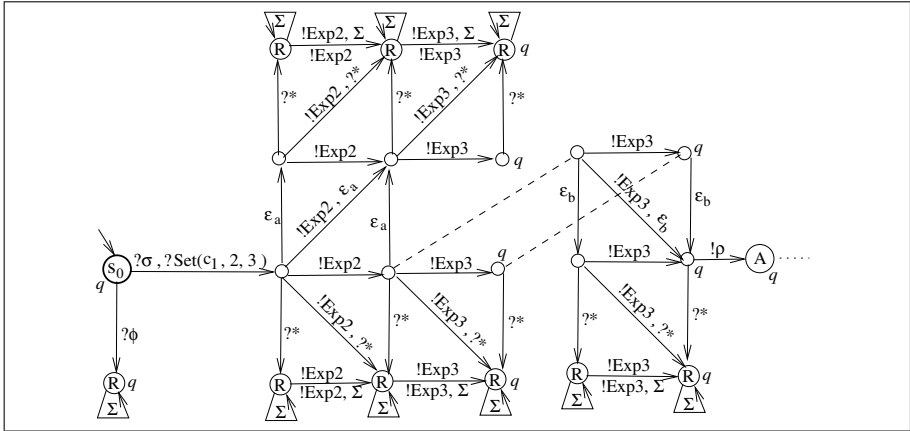


Fig. 7. Step 2: $SpecTP^{SEA}$ obtained from $SpecTP$ of Fig. 6

6.3 Step 3: Extracting the Visible Behaviour of $SpecTP^{SEA}$

By analogy with [3], we construct the visible behaviour of $SpecTP^{SEA}$ in four substeps:

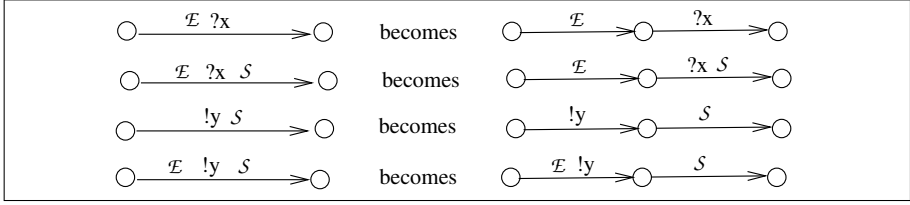


Fig. 9. Step 4: Separation of inputs and outputs

6.5 Step 5: Computing a Complete Test Graph (CTG)

By analogy with [3], we construct a Complete Test Graph (CTG) as follows:

- Let $L2A$ denote the set of states of $SpecTP_{VIS}^{SEA'}$ from which a state A is accessible.
- Let **Pass** denote the set of states A of $SpecTP_{VIS}^{SEA'}$.
- Let **Fail** = {fail} consist of a new state that is accessible by every non-specified output transition of $SpecTP_{VIS}^{SEA'}$ executable from $L2A$.
- Let **Inconc** be the set of states of $SpecTP_{VIS}^{SEA'}$ that are not in $L2A \cup \mathbf{Pass}$ and that are accessible from $L2A$ by a single output transition of $SpecTP_{VIS}^{SEA'}$.
- We then obtain CTG from $SpecTP_{VIS}^{SEA'}$ by: adding (implicitly) **Fail** and its incoming transitions, removing every state $q \notin L2A \cup \mathbf{Pass} \cup \mathbf{Inconc} \cup \mathbf{Fail}$, and removing outgoing transitions of every state $q \in \mathbf{Pass} \cup \mathbf{Inconc}$.

To synthesize test sequences executable in acceptable time, we define delays T_m and T_M such that $T_m < T_M$, and fictitious events $!\delta_m$ and $!\delta_M$ as follows: **IUT** is considered *quiescent* (resp. *quiescent forever*) if it generates no output during a period T_m (resp. T_M). The occurrence of $!\delta_m$ (resp. $!\delta_M$) means that Test-Controller has detected that no output occurs during a period T_m (resp. T_M). We proceed in CTG as follows:

- For every state** $q \notin \mathbf{Pass} \cup \mathbf{Inconc} \cup \mathbf{Fail}$ with an outgoing (possibly selfloop) $!\delta$: replace $!\delta$ by $!\delta_m$, and add an outgoing $!\delta_M$ that leads to a state $\in \mathbf{Inconc}$. Intuitively, if **IUT** is *quiescent forever* and no verdict (**Pass**, **Inconc** or **Fail**) has been produced, then the verdict **Inconc** is generated.
- For every state** $q \in \mathbf{Pass}$: insert a transition labelled $!\delta_m$ between: the incoming transitions of q , and q itself. Intuitively, the verdict **Pass** is generated only when **IUT** is *quiescent* and a state **Pass** has been reached.

Note that in CTG , every input (resp. output) must be interpreted as an output (resp. input) of the tester. For the $SpecTP_{VIS}^{SEA'}$ of Fig. 8, we obtain the CTG of Fig. 10. Transitions $!\delta_m$ and $!\delta_M$ in State 0 are irrelevant, because they can be preempted by the only possible other transition labelled $(?\sigma, ?Set(c_1, 2, 3))$, which is under the control of Test-Controller. Transition $!\delta_m$ in State 3 indicates that we are in a quiescent state. Transition $!\delta_M$ in State 3 indicates that we

are and will remain in a quiescent state; that is, $!\delta_M$ indicates that $!\rho$ will not be generated, and thus, the sequence that leads to a State **Pass** will not be completed; that is why $!\delta_M$ leads to a State **Inconc**. Transition $!\delta_m$ in State 4 indicates that after the execution of $!\rho$, we are in a quiescent state, and thus, we can generate a verdict **Pass**. For simplicity, the state **fail** and its incoming transitions are not represented; **fail** is implicitly reached by every non-specified output. Note that transition $!\delta_x$ can be easily realized by using $?Set(c_0, T_x)$ and $!Exp(c_0, T_x)$, for $x = m, M$, where c_0 is a clock not used for describing timing constraints of *Spec*.

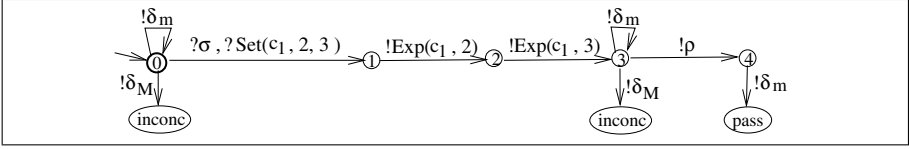


Fig. 10. Step 5: *CTG* obtained from $SpecTP_{VIS}^{SEA'}$ (the latter being obtained from $SpecTP_{VIS}^{SEA}$ of Fig. 8)

6.6 Step 6: Constructing Test Cases

Similarly to [3], the objective is to extract so-called controllable subgraphs of *CTG* as follows, *assuming that inputs can preempt outputs*: In a state of *CTG*, either one input transition is kept and all other input and output transitions are pruned, or all output transitions are kept and input transitions are pruned. Unreachable states are suppressed. Reachability to **Pass** can be preserved by a backward traversal of *CTG*. See [3] for more details on this step. For our example, the only controllable subgraph of the *CTG* of Fig. 10 is obtained by removing $!\delta_m$ and $!\delta_M$ in the initial state.

Note that by construction, all test cases are feasible. This is so, because a SEA produced by *SetExp* does not contain unfeasible paths.

7 Conclusion and Future Work

We extend the test method TGV [3] to the real-time case. Our approach is based on the use of a method that transforms a TA into an equivalent FSA, called Set-Exp-Automaton (SEA), using two additional types of events: *Set* and *Exp*. In comparison with some related work [7,8,9,10,11,12,13,14]⁵, we support nondeterminism and internal actions, define quiescent states in the real-time case, define the notion of forced transitions, and use an exact time information instead of a discrete time. And our method is relatively simple. Here are some possible future work:

- We intend to implement our method and apply it to complex examples. We have started to work on this aspect by implementing the transformation *SetExp*.

⁵ For lack of space, we omit to cite other references

- We have noted that, in the worst theoretic case, there is a state explosion problem with the obtained SEA CTG. Our conviction is that such a worst case is very rare and improbable. We intend to investigate such issue in a near future.
- Our method does not support unobservable clock resets (i.e., internal actions with clock reset). We intend to determine conditions under which our method is applicable in the presence of unobservable clock reset.
- We assume that **IUT** is *centralized*. When **IUT** is *distributed* in several sites: 1) a distributed test architecture consisting of several testers must be designed, and 2) every synthesized test case must be distributed into local test cases which are executed by the different testers, respectively. This *distributed* aspect will be studied in a near future.
- In a test sequence, the exact instants of inputs/outputs are not specified. During each test execution, the tester has to select an exact instant of each input a in the period of time when a is enabled. Therefore, the same given test sequence must be tested several times, for different instants of inputs. Since the number of instants at which any input may be applied is theoretically infinite, we will propose a method for selecting a finite number of *relevant* instants.

References

1. R. Alur. Timed automata. In *Proc. 11th Intern. Conf. on Comp. Aided Verif. (CAV)*, pages 8–22. Springer-Verlag LNCS 1633, 1999.
2. A. Khoumsi and L. Ouedraogo. A new transformation of timed automata into finite state automata. Submitted to *10th Intern. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Barcelona, Spain, April 2004.
3. C. Jard and T. Jéron. TGV: theory, principles and algorithms. In *Proc. 6th World Conf. on Integr. Design and Process Technol. (IDPT)*, Pasadena, California, USA, June 2002.
4. J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software-Concepts and Tools*, 17, 1996.
5. C. Jard, T. Jéron, L. Tanguy, and C. Viho. Remote testing can be as powerful as local testing. In *Proc. PSTV/FORTE*, Beijing, China, October 1999.
6. A. Khoumsi. A method for testing the conformance of real time systems. In *Proc. 7th Intern. Sympos. on Formal Techn. in Real-Time and Fault Toler. Systems (FTRTFT)*, Oldenburg, Germany, September 2002.
Available <http://www.gel.usherb.ca/khoumsi/Research/Public/FTRTFT02.ps>.
7. B. Nielsen. *Specification and test of real-time systems*. PhD thesis, Dept of Comput. Science, Faculty of Engin. and Sc., Aalborg University, Aalborg, Denmark, 2000.
8. R. Cardell-Oliver. Conformance testing of real-time systems with timed automata. *Formal Aspects of Computing*, 12:350–371, 2000.
9. R. Cardell-Oliver. Conformance testing of real-time systems with timed automata. In *Nordic Workshop on Programming Theory*, October 2000.
10. D. Clarke. *Testing real-time constraints*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, USA, 1996.

11. V. Braberman, M. Felder, and M. Massé. Testing timing behaviors of real time software. In *Proc. Quality Week 1997*, pages 143–155, San Francisco, USA, April-May 1997.
12. J. Peleska, P. Amthor, S. Dick, O. Meyer, M. Siegel, and C. Zahlten. Testing reactive real-time systems. In *Proc. Mater. for the School-5th Intern. School and Sympos. on Form. Technique in Real-Time and Fault-Toler. Syst. (FTRTFT)*, Lyngby, Denmark, 1998.
13. J. Springintveld, F. Vaandrager, and P. D’Argenio. Testing timed automata. Technical Report CTIT97-17, University of Twente, Amsterdam, The Netherlands, 1997.
14. A. En-Nouaary, R. Dssouli, F. Khendek, and A. Elqortobi. Timed test generation based on state characterization technique. In *Proc. 19th IEEE Real-Time Systems Sympos. (RTSS)*, Madrid, Spain, December 1998.

Property Oriented Test Case Generation

Jean-Claude Fernandez, Laurent Mounier, and Cyril Pachon

Verimag, Centre Equation, 2 avenue de Vignate, F38610 Gieres, France
{Jean-Claude.Fernandez, Laurent.Mounier, Cyril.Pachon}@imag.fr,
<http://www-verimag.imag.fr/>

Abstract. In this paper we propose an approach to automatically produce test cases allowing to check the satisfiability of a linear property on a given implementation. Linear properties can be expressed by formulas of temporal logic. An observer is built from each formula. An observer is a finite automaton on infinite sequences. Of course, testing the satisfiability of an infinite sequence is not possible. Thus, we introduce the notion of bounded properties. Test cases are generated from a (possibly partial) specification of the IUT and the property to validate is expressed by a parameterised automaton on infinite words. This approach is formally defined, and a practical test generation algorithm is sketched.

1 Introduction

Testing is certainly one of the most popular software validation techniques and it is a crucial activity in many domains such as embedded systems, critical systems, information systems, telecommunication, etc. Consequently, a lot of work was carried out during the last decade to both formalise the testing activities and to develop tools allowing to automate the production and execution of test suites.

The particular problem of testing if an implementation is “correct” with respect to its specification is referred to as *conformance testing*. This problem was mainly investigated inside the telecommunication area (as described in the ISO standard 9646 [7]), and a formal approach was outlined in [19,11,3]. These works gave birth to several (academic and commercial) tools[1,12,17,6] able to automatically generate test cases from a system specification. For instance, in [5], a technique is proposed to derive test cases from a (formal) specification and a test purpose. This technique is based on a partial exploration of a kind of product between the specification and the test purpose. An associated tool, called TGV, was developed by Irisa Rennes and Verimag Grenoble.

We first explicit a bit more the concepts of *black box testing* and *conformance testing*.

Black Box Testing. We consider here “black box” testing, meaning that the behaviour of the *IUT* (*Implementation Under Test*) is only visible by an external tester, through a restricted test interface (called PCO, for *Points of Control and Observation*). There exists two kinds of interactions between the tester and the IUT: outputs of the tester are stimuli sent in order to control the IUT,

whereas inputs of the tester are observations of the IUT's outputs. These sets of interactions are described by a *test architecture*. In black box testing the internal state of the IUT is not observable by the tester. Consequently:

- the tester cannot observe the internal non-determinism of the IUT;
- the tester should remain deterministic since it cannot backtrack the IUT to a given internal state.

A possible model to describe these sequences of interactions is Input-Output Labelled Transition System (IOLTS, see definition below).

Conformance Testing. Conformance testing is based on the following concepts:

- *IUT*: Even if the internal code of the IUT is not visible from the outside, its behaviour can be characterised by its interactions with its environment. This external behaviour can be modeled with an IOLTS. We suppose in the following that this IOLTS is *input complete*, that is, in each state the IUT cannot refuse any input proposed by the environment.
- *Test architecture*: Test architecture defines the set of interactions between the IUT and an environment, distinguishing between *controllable* and *observable* events. ISO9646 standard proposes four test methods : local test method, distributed test method, coordinated test method, and remote test method. All these methods are based on the black box testing principles and describe possible environment of the IUT. Of course, test architecture is a parameter of a test generation technique. In this paper, we consider a local test architecture.
- *Specification*: The specification represents the expected behaviour of the IUT, to be used as a reference by the tester. This expected behaviour can be also formally modelled by an IOLTS. Note that the specification not necessarily describes only the *visible* behaviour of the IUT, but it may also contain some of the internal actions performed by the implementation.
- *Conformance relation*: Defining whether an IUT is correct or not with respect to a specification is achieved in this context by introducing a formal relation between IOLTS. Several relations have been proposed so far, such as **ioco** [19]. Other relations have also been proposed on other models (such as **conf** [3]).
- *Test case*: Roughly speaking a test case is a set of interactions (input and output) sequences a tester can perform on an IUT interface. When executed, each interaction sequence delivers a *verdict* indicating whether the IUT was found conform or not on this particular execution (with respect to a given conformance relation).
- *Test purpose*: The test purpose represents a particular functionality (or sets of abstract scenarios) the user wants to test. It can also be modeled by an IOLTS, and may be used to automate the test case generation.

Although this conformance testing framework is now well established and happens to be very useful in the telecommunication area, its use in other application domains suffers, in our opinion, from two important limitations:

- first, it requires a rather *exhaustive* formal specification, since conformance is defined with respect to this specification, and any IUT exhibiting unexpected behaviours (from this specification point of view) would be rejected;
- second, the conformance relation is not very *flexible*: it is not always easy to understand what it does exactly preserve, and, more important, it is not possible to adapt it to the particular functionality one wants to test.

We propose in this work to extend this framework (and particularly what was done inside the TGV tool) to the generation of *property oriented* test cases. The general idea is to allow automatic test generation from a *partial specification* (not necessarily expressing the overall expected behaviour of the system), and with respect to a particular property (test case execution should indicate whether the IUT satisfy or not this property). This approach is outlined below.

Property Testing. The properties we consider are *linear* properties: each property defines a language (i.e., a set of sequences), and an IUT satisfies a given property if and only if all its execution sequences belong to its associated language. In this context it is a common practice to distinguish between *safety* properties, that can be checked by considering only finite execution sequences of the IUT, and *liveness* properties that need to consider also the infinite ones. Several characterisations of such properties have been proposed in the verification community, based on various specification formalisms : automata on infinite words (recognising ω -regular languages), linear-time temporal logics (or μ -calculus), boolean equation systems, etc. Automata on infinite words, like Büchi automata [4], are very interesting from an algorithmic point of view, and they are used in several decision procedures [20] implemented in model checkers. It can be shown in particular that any ω -regular language can be characterised by a Büchi automaton, or, equivalently, by a *deterministic* Rabin automaton, see for example [9,16]. Since the use of a deterministic automaton is an important issue in the test generation technique we propose in this paper, we will consider in the following that the property to be checked is expressed by a deterministic Rabin automaton. Of course, testing the satisfiability of a liveness property is not possible: it would require an infinite execution time. However, automata on infinite words can be parameterised to specify so-called *bounded liveness* properties: the automaton recognises a set of infinite execution sequences and some external parameters simply limit the “length” of the sequences to consider (this length being expressed for instance in number of interactions, or as an overall execution time).

More precisely, the test generation technique we propose can be sketched as follows:

- A (possibly partial) specification $\mathcal{S}$ is used as a “guideline” for the test case synthesis, and it is therefore supposed to be “closed enough” to the actual behaviour of the IUT. Note however that we do not require at this level any particular conformance relation between $\mathcal{S}$ and the IUT.
- A safety or bounded liveness property $\mathcal{P}$ is given through an observer $\mathcal{Obs}$ recognising sequences of $\neg\mathcal{P}$. This observer is a parameterised automaton on infinite words.

- Test cases are automatically generated by traversing the specification in order to find the “most promising” execution sequences able to show the non satisfiability of $\mathcal{P}$ by the IUT. These execution sequences are the sequences recognised by $\mathcal{O}bs$ that are the “closest” to the ones provided by the specification.

Related Work. Producing test cases from a formal specification to check the satisfiability of a given property is a rather natural idea, and consequently numerous works have been already carried out in this area, leading to various kinds of tools. They mostly differ in the nature of the specification and property they consider, and they are often based on probabilities to select the test sequences (such in [15,10,8,13]). However, an original aspect of our approach is the use of parameterized automata on infinite words to specify properties and to instantiate them only at test time. In addition, test cases we produce are IOLTS (not only sequence sets) that can be executed against non deterministic IUTs.

2 Models

This section formalises the different elements involved in the test case generation framework we propose.

2.1 Input-Outputs Labelled Transition Systems

The basic models we consider are based on Input-Output Labelled Transition Systems (IOLTS), namely Labelled Transition Systems in which input and output actions are differentiated (due to of the asymmetrical nature of the testing activity). We consider a finite alphabet of actions A , partitioned into two sets: *input actions* A_I and *output actions* A_O . A (finite) IOLTS is a quadruplet $M = (Q^M, A^M, T^M, q_{init}^M)$ where Q^M is the finite set of states, q_{init}^M is the initial state, $A^M \subseteq A$ is a finite alphabet of actions, and $T^M \subseteq Q^M \times A^M \cup \{\tau\} \times Q^M$ is the transition relation. Internal actions are denoted by the special label $\tau \notin A$. τ is assumed to be unobservable for the system’s environment whereas actions of A^M are visible actions representing the interactions either between the system and its environment, or between its internal components.

Notations. We denote by $\mathbb{N}$ the set of non negative integers. For each set X , X^* (resp. $X^\omega = [X \rightarrow \mathbb{N}]$) denotes the set of finite (resp. infinite) sequences on X . Let $\sigma \in X^*$; σ_i or $\sigma(i)$ denotes the i^{th} element of σ . We adopt the following notations and conventions: Let $\sigma \in A^*$, $\alpha \in A$, $p, q \in Q^M$. We write $p \xrightarrow{\alpha}_M q$ iff $(p, \alpha, q) \in T^M$ and $p \xrightarrow{\sigma}_M q$ iff $\exists \sigma_1, \sigma_2 \dots \sigma_n \in A$, $p_0, \dots, p_n \in Q^M$ such that $\sigma = \sigma_1 \cdot \sigma_2 \dots \sigma_n$ and $p_0 = p$, $p_i \xrightarrow{\sigma_{i+1}}_M p_{i+1}$ for $i < n$, $p_n = q$. In this case, σ is called a *trace* or *execution sequence*, and $p_0 \dots p_n$ a *run* over σ . An infinite run of M over an infinite execution sequence σ is an infinite sequence ρ of Q^M such that 1. $\rho(0) = q_{init}^M$ and 2. $\rho(i) \xrightarrow{\sigma(i)}_M \rho(i+1)$. $\mathbf{inf}(\rho)$ denotes the set of symbols from Q^M occurring infinitely often in ρ : $\mathbf{inf}(\rho) = \{q \mid \forall n. \exists i. i \geq n. \wedge \rho(i) = q\}$.

Let V a subset of the alphabet A . We define a *projection operator* $\downarrow_V: A^* \rightarrow V^*$ in the following manner: $\epsilon \downarrow_V = \epsilon$, $(a.\sigma) \downarrow_V = \sigma \downarrow_V$ if $a \notin V$, and $(a.\sigma) \downarrow_V = a.(\sigma \downarrow_V)$ if $a \in V$. This operator can be extended to a language L (and we note $L \downarrow V$) by applying it to each sequence of L . The language recognised by M is $\mathcal{L}(M) = \{w \mid \exists q \text{ such that } q_{\text{init}}^M \xrightarrow{w} q\}$.

Let $M = (Q^M, A^M, T^M, q_{\text{init}}^M)$ an IOLTS, we recall the *completeness*, *determinism* and *quiescence* notions.

Completeness. M is *complete with respect to a set of actions* $X \subseteq A$ if and only if for each state q^M of Q^M and for each action x of X , there is at least one outgoing transition of T^M from q^M labelled by $x \in X$:

$$\forall p^M \in Q^M. \forall x \in X. \exists q^M \in Q^M \text{ such that } p^M \xrightarrow{x}_M q^M.$$

Determinism. M is said *deterministic with respect to a set of actions* X if and only if it is a deterministic IOLTS containing only actions labelled by elements of X :

$$\forall p^M \in Q^M. \forall x \in X. p^M \xrightarrow{x}_M q^M \wedge p^M \xrightarrow{x}_M q'^M \Rightarrow q^M = q'^M.$$

We introduce a *determinisation operator*

$\text{det}(M, X) = (Q^{\text{det}(M, X)}, A^{\text{det}(M, X)}, T^{\text{det}(M, X)}, q_{\text{init}}^{\text{det}(M, X)})$ to compute a deterministic IOLTS with respect to X associated to M . This IOLTS is defined as follows: $Q^{\text{det}(M, X)} \subseteq 2^{Q^M}$, $A^{\text{det}(M, X)} = X$, $q_{\text{init}}^{\text{det}(M, X)} = \{q \in Q^M \mid q_{\text{init}}^M \xrightarrow{\omega}_M q \wedge \omega \in (A \setminus X)^*\}$ and $T^{\text{det}(M, X)} = \{(S_p, a, S_q) \mid \exists p \in S_p. \exists q \in S_q. p \xrightarrow{\omega, a}_M q \text{ with } a \in X \wedge \omega \in (A \setminus X)^*\}$. Note that, $\mathcal{L}(M) \downarrow X = \mathcal{L}(\text{det}(M, X))$.

Quiescence. A test should be able to observe IUT quiescence [19]. Several kinds of quiescence may happen: a state p is said *quiescent* in M either if it has no outgoing transition (deadlock), or if it belongs to a cycle of internal transitions (livelock):

$$\text{quiescent}(p) \equiv (\nexists(a, q). p \xrightarrow{a}_M q) \vee p \xrightarrow{\tau^+}_M p$$

Quiescence can be modelled at the IOLTS level by introducing an extra transition to each quiescent state labelled by a special symbol δ . δ is considered as an output (observable by the environment). In practice, the quiescence is observed by means of timers: a timeout occurs if and only if the implementation is locked inside a quiescent state. Formally, we handle quiescence by associating to LTS M its so-called “suspension automaton” $\delta(M) = (Q^M, A^M \cup \{\delta\}, T^{\delta(M)}, q_0^M)$ where $T^{\delta(M)} = T^M \cup \{(p, \delta, p) \mid p \in Q^M \wedge \text{quiescent}(p)\}$.

2.2 Specification and Implementation

The system specification is in general expressed using a dedicated language or notation (SDL, Lotos, UML, etc). The operational semantics of this language can be described in terms of IOLTS. Thus, we note the specification $S = (Q^S, A^S, T^S, q_{\text{init}}^S)$, with $A^S = A_I^S \cup A_O^S$.

The Implementation Under Test (IUT) is assumed to be a “black box” whose behaviour is known by the environment only through a restricted interface (a set of inputs and outputs). From a theoretical point of view, it is convenient to consider the IUT behaviour as an IOLTS $IUT = (Q^{IUT}, A^{IUT}, T^{IUT}, q_{init}^{IUT})$, where $A^{IUT} = A_I^{IUT} \cup A_O^{IUT}$ is the IUT interface. We assume in addition that this IUT is complete with respect to A_I (it never refuses an unexpected input), and that the specification S is a *partial IOLTS* of the IUT:

$$A^S \subseteq A^{IUT} \text{ and } \mathcal{L}(S) \subseteq \mathcal{L}(IUT) \downarrow (A^S).$$

Intuitively, a specification is partial if each trace of the specification may be executed by the IUT (but the IUT may contain unspecified behaviours).

2.3 Property and Satisfiability Relation

The objective of this work is to generate test cases allowing to check the satisfiability of some classes of *properties* on a given IUT. In particular we restrict ourselves to *linear properties*, those associated *models* are sets of IOLTS execution sequences. Two kinds of linear properties can be considered: *safety* properties, characterised by *finite* execution sequences, and *liveness* properties, characterised by *infinite* ones. Thus, an IUT will satisfy a given linear property $\mathcal{P}$ if and only if all of its execution sequences belong to the model of $\mathcal{P}$.

From the test point of view, only the (non-)existence of a finite execution sequence can be checked on a given IUT (since the test execution time has to remain bounded). This restricts in practice the test activity to the validation of safety properties. Nevertheless, an interesting sub-class of safety properties are the so-called *bounded liveness*. Such properties allow for instance to express that the IUT will exhibit a particular behaviour within a given amount of time, or before a given number of iterations has been reached. From a practical point of view, it is very useful to express such properties as liveness (i.e., in terms of infinite execution sequences, telling that the expected behaviour will *eventually* happen), and then to bound their execution only at test time. The main advantage is that the “bounds” are not part of the test generation process, and they can be chosen depending on the concrete test conditions. Therefore, we propose in this section to specify the properties of interest using a general model, allowing to express both finite and infinite execution sequences. This model is then “parameterised” to handle bounded liveness properties.

Automata on Infinite Words. Several acceptance conditions (Büchi, Muller, Streett, Rabin, etc) have been proposed to extend finite-state IOLTS to recognise infinite sequences. We recall the definition of Büchi and Rabin automata and illustrate on an example the difference between them.

Definition 1. A Büchi automaton R_b is a structure $(B, \mathcal{G}^B)$ where $B = (Q^B, A^B, T^B, q_{init}^B)$ is an IOLTS and $\mathcal{G}^B$ is a subset of Q^B . The automaton R_b accepts an infinite execution σ of A^{B^ω} if there is an infinite run ρ of B over σ such that $\inf(\rho) \cap \mathcal{G}^B \neq \emptyset$.

Definition 2. A Rabin automaton R_a is a structure $(R, \mathcal{T}^R)$ where $R = (Q^R, A^R, T^R, q_{init}^R)$ is an IOLTS and $\mathcal{T}^R = \langle (L_1^R, U_1^R), (L_2^R, U_2^R), \dots, (L_k^R, U_k^R) \rangle$ is a pairs table with $L_i^R, U_i^R \subseteq Q^R$ for $i \in \{1, 2, \dots, k\}$. The automaton R_a accepts an infinite execution σ of $A^{R\omega}$ if there is an infinite run ρ of R over σ such that for some $i \in \{1, 2, \dots, k\}$, $\inf(\rho) \cap L_i^R \neq \emptyset$ and $\inf(\rho) \cap U_i^R = \emptyset$.

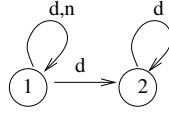


Fig. 1. Non deterministic Büchi automaton recognising $(d + n)^*d^\omega$

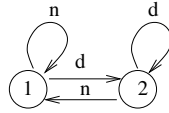


Fig. 2. Deterministic Rabin automaton recognising $(d + n)^*d^\omega$

Example. As an example, consider the following property “The system always comes back to its nominal mode (action n) after entering a degraded one (action d)”. This property can be expressed by the following (ω -regular) language: $L = (d^*n)^\omega$. The negation of this property is expressed by $\bar{L} = (d + n)^*d^\omega$ which is not recognisable by a deterministic Büchi automaton. The non deterministic Büchi automaton recognising $\bar{L}$ is given by the figure 1, with $\mathcal{G}^B = \{2\}$ and the initial state is 1.

Consider now the deterministic automaton of figure 2 as a Büchi automaton, with $\mathcal{G}^B = \{2\}$ and the initial state is 1. This automaton accepts all sequences containing infinitely often many occurrences of n or many occurrences of d , which are not in $\bar{L}$.

Now, if the automaton of figure 2 is considered as a Rabin automaton with the pair table $\{\{2\}, \{1\}\}$, then this automaton recognises exactly $\bar{L}$ (it accepts an infinite word iff it has infinitely many occurrences of d). Thus, we consider in this paper *deterministic Rabin automata* [14] since they recognise all classes of ω -regular language.

As another example, the figure 3 shows a Rabin automata with pair (L, U) equals to $(\{3\}, \emptyset)$ recognising execution sequences in which a **req** action is at some point followed by an **error** action. The δ -loop on state 3 indicates that a finite execution sequence terminating after an **error** action is recognised by this automaton. This artefact allows to deal both with finite and infinite execution sequences.

Rabin automata are a natural model to express liveness properties. However, to correctly handle bounded liveness as well, we need to “parameterise” these automata in order to limit the size of the infinite execution sequences they recognise. The (simple) solution we propose consists in associating a counter to each

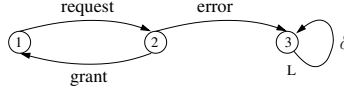


Fig. 3. Example of a safety property expressed by a Rabin automaton

state belonging to an (L_i, U_i) pair. An execution sequence σ is now recognised if and only if it visits “sufficiently often” an L_i -state, and “not too often” an U_i -state, according to the counters associated to these sets (those actual value will be instantiated at test time).

Definition 3. A *Parameterised Rabin automaton* is a tuple $PR_a = (R, \mathcal{T}^R, \mathcal{C}^R)$ where $(R, \mathcal{T}^R)$ is a Rabin automaton and $C = \{(cl_1, cu_1), \dots, (cl_k, cu_k)\}$ with $cl_i, cu_i \in \mathbb{N}$. An execution sequence σ is accepted by PR_a if and only if: there is a finite run of PR_a ρ on σ such that for some $i \in \{1, 2, \dots, k\}$

$$|\{j \mid \rho(j) \in L_i^R\}| \geq cl_i \text{ and } |\{j \mid \rho(j) \in U_i^R\}| \leq cu_i$$

Thus, the language accepted by PR_a is $\mathcal{L}(PR_a)$, the set of sequences accepted by PR_a .

Observer and Satisfiability Relation. Test case generation with respect to a linear property $\mathcal{P}$ is facilitated by considering an *observer automaton* recognising exactly the execution sequences of $\neg\mathcal{P}$. Since we want to deal with safety and bounded liveness properties we choose here to model these observers as deterministic Parameterised Rabin automaton $\mathcal{Obs} = (\mathcal{O}, \mathcal{T}^\mathcal{O}, \mathcal{C}^\mathcal{O})$. We are now able to formally define the *satisfiability relation* relation we consider between an IUT and a linear property.

Definition 4. Let IUT be an *IOLTS*, $\mathcal{P}$ a property, and $\mathcal{Obs} = (\mathcal{O}, \mathcal{T}^\mathcal{O}, \mathcal{C}^\mathcal{O})$ the observer recognising the sequences of $\neg\mathcal{P}$, where $\mathcal{O} = (Q^\mathcal{O}, A^\mathcal{O}, T^\mathcal{O}, q_{init}^\mathcal{O})$. Then, IUT satisfies $\mathcal{P}$ iff $(\mathcal{L}(IUT) \downarrow A^\mathcal{O}) \cap \mathcal{L}(\mathcal{O}) = \emptyset$. That is, none of the observable execution sequences of the IUT are recognised by the observer.

2.4 Test Architecture and Test Case

Test Architecture. At the abstract level we consider, a test architecture is simply a pair $(\mathcal{A}_c, \mathcal{A}_u)$ of actions sets, each of them being a subset of $\mathcal{A}$: the set of *controllable* actions $\mathcal{A}_c$, initiated by the tester, and the set of *observable* actions $\mathcal{A}_u$, observed by the tester. A test architecture will be said *compliant* with an observer $\mathcal{Obs}$ if it satisfies the following constraints: $A_I^\mathcal{O} \subseteq \mathcal{A}_c$ and $A_O^\mathcal{O} \subseteq \mathcal{A}_u$. In other words the tester needs at least being able to control (resp. observe) all inputs (resp. outputs) appearing in the observer.

Test Cases. For a given observer $\mathcal{Obs}$, a test architecture $(\mathcal{A}_c, \mathcal{A}_u)$ compliant with $\mathcal{O}$, an (abstract) test case is a Parameterised Rabin automaton $(TC, \mathcal{T}^{TC}, \mathcal{C}^{TC})$ with $TC = (Q^{TC}, A^{TC}, T^{TC}, q_{init}^{TC})$ and satisfying the following requirements:

1. $A^{TC} = A_I^{TC} \cup A_O^{TC}$ with $A_O^{TC} = \mathcal{A}_c$ and $A_I^{TC} \subseteq \mathcal{A}_u$. Note that $\mathcal{A}_c$ (resp. $\mathcal{A}_u$) is the set of controllable (resp. observable) actions.

2. TC is deterministic wrt A^{TC} , controllable (for each state of Q^{TC} there is at most one outgoing transition labelled by an action of A_c), and input-complete (for each state of Q^{TC} , for each element a of A_u , there exists exactly one outgoing transition labelled by a).
3. The pair table $\mathcal{T}^{\text{TC}} = \langle (L_1^{\text{TC}}, U_1^{\text{TC}}), \dots, (L_k^{\text{TC}}, U_k^{\text{TC}}) \rangle$ is defined with respect to the observer $\mathcal{O}\text{bs}$:
 $q^{\text{TC}} \in L_i^{\text{TC}}$ (resp. U_i^{TC}) iff $\exists \sigma \in A^{\text{TC}*}, q^{\text{O}} \in L_i^{\text{O}}$ (resp. U_i^{O}) such that $q_{\text{init}}^{\text{O}} \xrightarrow{\sigma} q^{\text{O}}$
 and $q_{\text{init}}^{\text{TC}} \xrightarrow{\sigma \downarrow A} q^{\text{TC}}$

The last condition expresses that there is an execution sequence of the test case starting from the initial state of the test case and leading to a state of L_i^{TC} (resp. U_i^{TC}) if there is a corresponding execution sequence of the observer starting from the initial state of the observer and leading to a state of L_i^{O} (resp. U_i^{O}).

2.5 Test Cases Execution and Verdicts

Let $\text{IUT} = (Q^{\text{IUT}}, A^{\text{IUT}}, T^{\text{IUT}}, q_{\text{init}}^{\text{IUT}})$ an implementation, $(\text{TC}, \mathcal{T}^{\text{TC}}, \mathcal{C}^{\text{TC}})$ a test case with $\text{TC} = (Q^{\text{TC}}, A^{\text{TC}}, T^{\text{TC}}, q_{\text{init}}^{\text{TC}})$, and (A_c, A_u) a test architecture. The test execution of TC on IUT can be modelled by a parallel composition between IUT and TC with synchronisations on action sets A_c and A_u . More formally this test execution can be described by an IOLTS $\mathcal{E} = (Q^{\mathcal{E}}, A^{\mathcal{E}}, T^{\mathcal{E}}, q_{\text{init}}^{\mathcal{E}})$, where $A^{\mathcal{E}} = A^{\text{TC}}$, and sets $Q^{\mathcal{E}}$ and T^{TC} are defined as follows:

- $Q^{\mathcal{E}}$ is a set of *configurations*. A *configuration* is a triplet $(p^{\text{TC}}, p^{\text{IUT}}, \lambda)$ where $p^{\text{TC}} \in Q^{\text{TC}}, p^{\text{IUT}} \in Q^{\text{IUT}}$ and λ is a partial function from Q^{TC} to $\mathbb{N}$, which counts the number of times an execution sequence visits a state belonging to L_i^{TC} or U_i^{TC} .
- $T^{\mathcal{E}}$ is the set of transitions $(p^{\text{TC}}, p^{\text{IUT}}, \lambda) \xrightarrow{a}_{\mathcal{E}} (q^{\text{TC}}, q^{\text{IUT}}, \lambda')$ such that
 - $p^{\text{TC}} \xrightarrow{a}_{\text{TC}} q^{\text{TC}}, p^{\text{IUT}} \xrightarrow{a}_{\text{IUT}} q^{\text{IUT}}$ and
 - $\lambda'(q^{\text{TC}}) = \begin{cases} \lambda(q^{\text{TC}}) & \text{if } q^{\text{TC}} \notin \bigcup_{i \in \{1, \dots, k\}} (L_i^{\text{TC}} \cup U_i^{\text{TC}}) \\ \lambda(q^{\text{TC}}) + 1 & \text{if } q^{\text{TC}} \in \bigcup_{i \in \{1, \dots, k\}} (L_i^{\text{TC}} \cup U_i^{\text{TC}}) \end{cases}$

The initial configuration $q_{\text{init}}^{\text{TC}}$ is $(q_{\text{init}}^{\text{TC}}, q_{\text{init}}^{\text{IUT}}, \lambda_{\text{init}})$, where for all q , $\lambda_{\text{init}}(q) = 0$.

$T^{\mathcal{E}}$ describes the interactions between the IUT and the test case. Each counter associated with a state of $L_i^{\text{TC}} \cup U_i^{\text{TC}}$ is incremented when an execution sequence visits this state.

Verdicts. Test execution is supposed to deliver some *verdicts* to indicate whether the IUT was found correct or not. These verdicts can be formalised as a function on runs of $\mathcal{E}$ to the set $\{\mathbf{Pass}, \mathbf{Fail}\}$. More precisely:

Fail. The execution of a run ρ of $\mathcal{E}$ on σ gives the verdict **Fail** if and only if there is an $i \in \{1, 2, \dots, k\}$ and a $l \in \mathbb{N}$ such that

1. $\rho(l) = (p_l^{\text{TC}}, p_l^{\text{IUT}}, \lambda_l)$, $p_l^{\text{TC}} \in L_i^{\text{TC}}$ and $\lambda_l(p_l^{\text{TC}}) \geq cl_i$, and
2. for each $m \in [0 \dots l]$ $\rho(m) = (q_m^{\text{TC}}, q_m^{\text{IUT}}, \lambda_m)$ satisfies $\lambda_m(q_m^{\text{TC}}) \leq cu_i$. In this case, the property is not satisfied.

Pass. Similarly, the execution of a run ρ give the verdict **Pass** iff $\forall i \in \{1, 2, \dots, k\}$ and $\forall l \in \mathbb{N}$

1. $\rho(l) = (p_l^{\text{TC}}, p_l^{\text{IUT}}, \lambda_l)$ and $p_l^{\text{TC}} \in L_i^{\text{TC}}$ implies $\lambda_l(p_l^{\text{TC}}) < cl_i$ or
2. there is a $m \in [0 \dots l]$ $\rho(m) = (q_m^{\text{TC}}, q_m^{\text{IUT}}, \lambda_m)$ and $\lambda_m(q_m^{\text{TC}}) > cu_i$.

In practice the test case execution can be performed as follows:

- At each step of the execution the controllability condition may give a choice between a controllable and an observable action. In this situation the tester can first wait for the observable action to occur (using a local timer), and then choose to execute the controllable one.
- Formal parameters $\mathcal{C}^{\text{TC}}$ are instantiated according to the actual test environment. Counters are then associated to each sets U_i^{TC} and L_i^{TC} . These counters, initialised to 0, are incremented (inside the test case) whenever a corresponding state is reached during test execution. Thus, a **Fail** verdict is issued as soon as an incorrect execution sequence is reached (according to definition above), and a **Pass** verdict is issued either if the current execution sequence visits “too many often” a state of U_i^{TC} ($\lambda_m(q_m^{\text{TC}}) > cu_i$), or if a global timer, started at the beginning of test execution, expires. This last case occurs when an execution sequence enter a loop without state belonging to L_i^{TC} or U_i^{TC} .

3 Test Generation

We propose in this section an algorithm to automate the generation of “property oriented” test cases. This algorithm takes as input a (partial) specification S_0 of a given implementation IUT, an observer (a deterministic parameterised Rabin automaton) $\mathcal{O}bs = (O, \mathcal{T}^{\mathcal{O}}, \mathcal{C}^{\mathcal{O}})$ characterising the negation of a linear property $\mathcal{P}$, and a test architecture $TA = (A_c, A_u)$. Test cases produced by this algorithm are *sound* in the sense that, when executed against the IUT, a **Fail** verdict is produced *only if* this IUT does not satisfy property $\mathcal{P}$.

The test generation algorithm we propose is based on two steps: generation of a so-called *test graph* (TG, for short) and *test cases selection* from this TG. We first describe these two steps at an abstract level, and then we discuss some implementations issues.

3.1 Test Graph

The purpose of the test graph is to gather a set of execution sequences, computed from the specification S_0 and the observer $\mathcal{O}bs$, compliant with the test architecture TA (i.e., executable by an external tester), and able to witness the non satisfiability of $\mathcal{P}$ for a given IUT. Each controllable sub-graph of this TG could then be turned into an executable test case for property $\mathcal{P}$ (as defined in the previous section).

However, even for a simple property and with a restricted test architecture, it appears that the number of sequences matching this definition is quite large: in fact it could be *any* sequence over $A_c \cup A_u$ recognised by $\mathcal{O}$. Considering such a

“complete” test graph would be of limited practical interest in this context: most of these sequences are likely to be very “far” from the actual IUT behaviour, and executing them would not provide very useful information. Consequently, the probability to extract a “relevant” controllable test case from this large set would be rather low. Therefore we need some heuristic to restrict this test graph to the most promising execution sequences.

The heuristic we propose here to compute the test graph is to exploit at best the information provided by the specification, proceeding as follows:

1. First, we transform the initial specification S_0 by computing its deterministic suspension automaton S with respect to the test architecture TA : $S = \delta(\det(S_0, A_c \cup A_u))$. This operation preserves the observable/ controllable language of the specification: $\mathcal{L}(S) = \mathcal{L}(S_0) \downarrow (A_c \cup A_u)$.
2. Then, we select the longest sequences of $\mathcal{L}(S)$ matching with a prefix of $\mathcal{L}(\mathcal{O})$. Such sequences are the most promising candidates to witness the non-satisfiability of $\mathcal{P}$ since they belong both to the specification (and then are supposed to be executable on the IUT), and to a prefix of $\mathcal{L}(\mathcal{O})$.
3. Finally, these sequences are then extended to cover complete sequences of $\mathcal{L}(\mathcal{O})$. Note that if the specification already contains a complete sequence of $\mathcal{L}(\mathcal{O})$ (and not only one of its proper prefix) this means that the specification itself does not satisfy $\mathcal{P}$.

From a more formal point of view the test graph we compute is a parameterised Rabin automaton $(TG, \mathcal{T}^{TG}, \mathcal{C}^{TG})$: the IOLTS TG gathers the set of execution sequence described above, and the pair table $\mathcal{T}^{TG}$ and counter sets $\mathcal{C}^{TG}$ are inherited from $\mathcal{T}^{\mathcal{O}}$ and $\mathcal{C}^{\mathcal{O}}$. This is described in definition 5 below, proceeding in two steps:

1. Computation of an asymmetric product $\otimes$ between S and $\mathcal{O}$. The purpose of this product is to mark each state p_S of S with a corresponding state $p_{\mathcal{O}}$ of $\mathcal{O}$, such that p_S and $p_{\mathcal{O}}$ are reachable from the initial states by “matching” execution sequences (rules R1 and R2).
2. Selection of the longest execution sequences of $S \otimes \mathcal{O}$ matching with a prefix of $\mathcal{L}(\mathcal{O})$, and extension of these sequences to obtain a complete sequence of $\mathcal{L}(\mathcal{O})$. This is performed by rule R4: a transition $(p_S, p_{\mathcal{O}}) \xrightarrow{a} (p_S, q_{\mathcal{O}})$ is added to the transition relation T^{TG} iff such a transition exists in $\mathcal{O}$ but not in $S \otimes \mathcal{O}$.

Definition 5. Let $TA = (A_c, A_u)$ a test architecture, S_0 a specification and $S = (Q^S, A^S, T^S, q_{init}^S)$ its deterministic suspension automaton with respect to TA : $S = \delta(\det(S_0, A_c \cup A_u))$. Let $(\mathcal{O}, \mathcal{T}^{\mathcal{O}}, \mathcal{C}^{\mathcal{O}})$ be an observer with

$\mathcal{O} = (Q^{\mathcal{O}}, A^{\mathcal{O}}, T^{\mathcal{O}}, q_{init}^{\mathcal{O}})$ and $\mathcal{T}^{\mathcal{O}} = \langle (L_1^{\mathcal{O}}, U_1^{\mathcal{O}}), (L_2^{\mathcal{O}}, U_2^{\mathcal{O}}), \dots, (L_k^{\mathcal{O}}, U_k^{\mathcal{O}}) \rangle$ such that TA is compliant with $\mathcal{O}$. We define the Parameterised Rabin automaton

$(TG, \mathcal{T}^{TG}, \mathcal{C}^{TG})$ where $TG = (Q^{TG}, A^{TG}, T^{TG}, q_{init}^{TG})$, such that $Q^{TG} \subseteq Q^S \times Q^{\mathcal{O}}$, $A^{TG} \subseteq A^S$, $q_0^{TG} = (q_0^S, q_0^{\mathcal{O}})$, and Q^{TG}, T^{TG} are obtained as follows:

1. Let $Q^\otimes$ and $T^\otimes$ be the smallest sets satisfying rules R0, R1 and R2 below:

$$q_0^{TG} \in Q^\otimes \text{ [R0]}$$

$$\frac{(p_S, p_O) \in Q^\otimes, p_S \xrightarrow{a}_{T^S} q_S, p_O \xrightarrow{a}_{T^O} q_O}{(q_S, q_O) \in Q^\otimes, (p_S, p_O) \xrightarrow{a}_{T^\otimes} (q_S, q_O)} \text{ [R1]}$$

$$\frac{(p_S, p_O) \in Q^\otimes, p_S \xrightarrow{a}_{T^S} q_S, \neg p_O \xrightarrow{a}_{T^O} q_O}{(q_S, p_O) \in Q^\otimes, (p_S, p_O) \xrightarrow{a}_{T^\otimes} (q_S, p_O)} \text{ [R2]}$$

2. Then, Q^{TG} and T^{TG} are the smallest sets satisfying rules R3 and R4 below:

$$Q^\otimes \subseteq Q^{TG}, T^\otimes \subseteq T^{TG} \text{ [R3]}$$

$$\left. \begin{array}{l} (p_S, p_O) \in Q^{TG}, \\ p_O \xrightarrow{a}_{T^O} q_O, \sigma \in (A^S \setminus A^O)^* \\ \nexists q_S. ((p_S, p_O) \xrightarrow{\sigma, a}_{T^\otimes} (q_S, q_O)) \\ \hline (p_S, q_O) \in Q^{TG}, (p_S, p_O), \xrightarrow{a}_{T^{TG}}, (p_S, q_O) \end{array} \right\} \text{ [R4]}$$

3. The pair table $\mathcal{T}^{TG}$ is equal to $\langle (L_1^{TG}, U_1^{TG}), (L_2^{TG}, U_2^{TG}), \dots, (L_k^{TG}, U_k^{TG}) \rangle$ where L_i^{TG} and U_i^{TG} are defined as follows:

$$L_i^{TG} = \{(p_S, p_O) \in Q^{TG} \mid q_O \in L_i^O\}$$

$$U_i^{TG} = \{(p_S, p_O) \in Q^{TG} \mid$$

4. The set of counters $\mathcal{C}^{TG}$ is directly inherited from $\mathcal{O}bs$:

$$\mathcal{C}^{TG} = \mathcal{C}^O$$

3.2 Test Cases Selection

The purpose of the test case selection is to generate a particular test case TC from the test graph TG. Roughly speaking, it consists in “extracting” a sub-graph of TG that are controllable and containing a least a sequence of $\mathcal{L}(\mathcal{O})$.

Clearly, to belong to $\mathcal{L}(\mathcal{O})$, an execution sequence of $\mathcal{O}$ has to reach a cycle containing a state belonging to some distinguished set L_i^O (for some i) of the pair table associated to $\mathcal{O}$. Conversely, any sequence of $\mathcal{O}$ not leading to a strongly connected component of $\mathcal{O}$ containing a state of L_i^O cannot belong to $\mathcal{L}(\mathcal{O})$. Therefore, we first define on TG the predicate L2L (for “leads to L”), to denote the set of states leading to such a strongly connected component:

$$\text{L2L}(q) \equiv \exists (q_1, q_2, \omega_1, \omega_2, \omega_3). (q \xRightarrow{\omega_1}_{T^{TG}} q_1 \xRightarrow{\omega_2}_{T^{TG}} q_2 \xRightarrow{\omega_3}_{T^{TG}} q_1 \text{ and } \exists i. q_2 \in L_i^O)$$

We can now define a subset of relation T^{TG} , controllable, and containing at least a sequence of $\mathcal{L}(\mathcal{O})$. This subset, computed by the function *select* below, contains all non controllable transition of T^{TG} (labelled by an element of A_u), and at most one (randomly chosen) controllable transition of T^{TG} leading to a state of L2L when several such transitions exist from a given state of TG:

$$\text{select}(T^{TG}) = \{(p, a, q) \in T^{TG} \mid a \in A_u \text{ or}$$

$$a = \text{one-of}(\{a_i \in A_c \mid p \xrightarrow{a_i}_{T^{TG}} q_i \text{ and } \text{L2L}(q_i)\})\}$$

Note that this function preserves the reachability of states belonging to $\mathcal{L}^O$.

Finally, this subset of T^{TG} remains to be extended with all (non controllable) action of a_u not explicitly appearing in T^{TG} , to ensure that the test case execution will never be stopped by reception of an unexpected event. The definition of a test case TC is then the following:

Definition 6. let $(TG, \mathcal{T}^{TG}, \mathcal{C}^{TG})$ a test graph with $TG=(Q^{TG}, A^{TG}, T^{TG}, q_{init}^{TG})$ and $TA = (A_c, A_u)$ a test architecture. A test case $(TC, \mathcal{T}^{TC}, \mathcal{C}^{TC})$ is a Parameterised Rabin automaton with $TC=(Q^{TC}, A^{TC}, T^{TC}, q_{init}^{TC})$ such that $q_0^{TC} = q_0^{TG}$, $A^{TC} = A^{TG} \cup A_u$, Q^{TC} is the subset of Q^{TG} reachable by T^{TC} from q_0^{TC} , $\mathcal{T}^{TC}$ is the restriction of $\mathcal{T}^{TG}$ over Q^{TC} , and T^{TC} is defined as follows:

$$T^{TC} = select(T^{TG}) \cup \{(p, a, p) \mid a \in A_u \text{ and } \nexists q. (p, a, q) \in T^{TG}\}$$

3.3 Implementation Issues

We briefly sketch the concrete algorithms that could be used to implement the test case generation method proposed in this section. The objective here is not to provide a detailed implementation description (beyond the scope of this paper), but rather to give some indications on its algorithmic complexity. A possible (and simple) approach to compute a test case TC from a specification S, an observer Obs and a test architecture TA is to proceed as follows:

1. computation of S (determinisation and suspension of S_0) and computation of sets $Q^\otimes$ and $T^\otimes$ introduced in definition 5. These operations can be done during a joint traversal of S and O.
2. computation of the test graph TG (sets Q^{TG} and T^{TG}) from the previous result. This can be done through a single traversal of $Q^\otimes$ and $T^\otimes$.
3. computation of the strongly connected components of TG containing a distinguished state of L_i^O , using for instance Tarjan's algorithm [18]. This operation also gives the L2L predicate.
4. test case selection (computation of function *select*) using a backward traversal of TG.

Apart the determinisation phase, all these operations remain linear in the number of transitions of the LTS considered, but the test graph has to be explicitly stored. However, some of the algorithms proposed in the TGV tool could certainly be used to perform most of these operations on an on-the-fly basis. This point has not been investigated at this time.

4 Example

4.1 System Description

We consider a control system for an automatic door, specified by the IOLTS given in figure 4. The behaviour of the controller is the following: it can receive a request for opening the door (REQOPEN), the door is then successively open (OPEN) and closed (CLOSE). It can also receive a LOCK request, those effect is to definitely lock the door, or any other requests OTHER, that are silently ignored. All these actions are supposed to belong to the test architecture. A possible specification of the controller is the IOLTS shown at the figure 4.

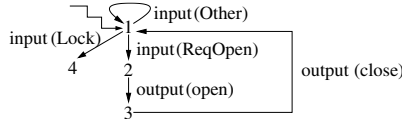


Fig. 4. Specification

The property we want to test on this system is: whenever the door is open, then it should be closed before a given amount of time (to be precised at test time). The negation of this property (the observer) is modelled by the Parameterised Rabin automaton of figure 5, where the α label denotes any observable action other than **CLOSE** (including δ). We now assume that the IUT is not quite conform to the specification. In particular it may spontaneously output an **ABORT** action and re-enter the initial state. The corresponding IOLTS is pictured on figure 8.

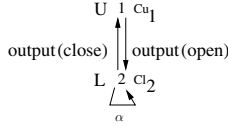


Fig. 5. Observer

4.2 Test Graph Generation

The first step consists in generating a test graph from the specification and the observer. The corresponding deterministic parameterised Rabin automaton is shown on figure 6. Note that the sets L and U are inherited from the observer. On this test graph, the execution sequences belonging to the language of the observer are the ones ending by α^ω (namely in states 32 and 42).

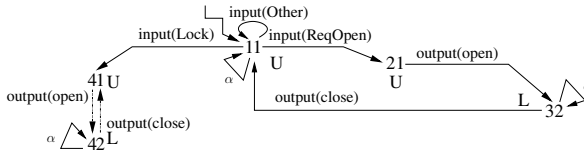


Fig. 6. Test Graph

4.3 Test Selection and Test Execution

From the test graph we can then extract some particular test cases, for instance the one pictured on figure 7.

Transitions labelled with α indicate that the test case is output complete. Executing this test case may exhibit a possible incorrect behaviour of the IUT (figure 8), in which an occurrence of the **ABORT** action in state 32 leads to a **Fail** verdict (since the IUT is deadlocked in this state).

More precisely, each time states 32, 11 or 21 are reached their respective counter are incremented. So during the test execution, the counter associated with the state 32 can overflow if an **ABORT** action occurred. Of course, this scenario is not guaranteed to appear since this incorrect behaviour is not fully controllable by the tester.

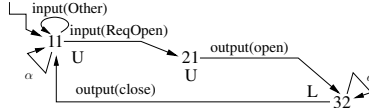


Fig. 7. Test Case

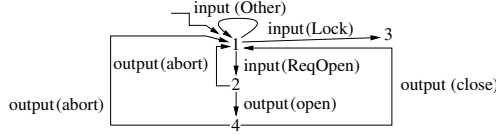


Fig. 8. IUT

5 Conclusion

In this paper, we have proposed an approach to automatically produce test cases allowing to check the satisfiability of a linear property on a given implementation. Parameterised test cases are generated from a (possibly partial) specification of the IUT, the (bounded liveness) property being expressed itself by a Parameterised Rabin automaton. The resulting test case can then be instantiated only at test time, depending on the test environment considered (for instance the target architecture, or the actual communication structure between the tester and the IUT, etc.). This approach has been formally defined, and a practical test generation algorithm has been sketched.

The objective of this work is to extend to other contexts or application domains the framework of conformance testing, already well established in the telecommunication area. We believe that a prerequisite was to make this framework more flexible, for instance allowing the use partial specifications, or allowing the validation of explicit properties. This is a first step in this direction.

This work can now be extended in several directions. First we need to prototype the algorithms we have proposed to better estimate their performances, and possible optimisations. Then, their application on various case studies will certainly allow to improve the test selection strategy (possibly using TGV-like test purposes in combination with a property). Finally, the use of static analysis techniques (for instance as presented in [2]) could also certainly improve the efficiency of the test generation algorithm by focusing on most promising parts of the specification.

Acknowledgement

We would like to thank the people of a French group working on *robustness testing*, inside a French action supported by the CNRS and gathering members of IRISA, LAAS, LABRI, LRI and VERIMAG laboratories

(<http://www.laas.fr/TSF/AS23/>).

We would like also to thank the anonymous referees.

References

1. A. Belinfante, J. Feenstra, R. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal Test Automation : a Simple Experiment. In *12th International Workshop on Testing of Communicating Systems*, G. Csopaki et S. Dibuz et K. Tarnay, 1999. Kluwer Academic Publishers.
2. M. Bozga, J.-C. Fernandez, and L. Ghirvu. Using static analysis to improve automatic test generation. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 235–250, 2000.
3. E. Brinksma. A theory for the derivation of tests. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing and Verification VIII, IFIP*, pages 63–74. Elsevier Science Publishers, B.V., North-Holland, 1988.
4. J. Büchi. On a decision method in restricted second order arithmetic. In N. et al., editor, *Logic, Methodology and Philosophy of Sciences*. Stantford Univ. Press, 1962.
5. J.-. Fernandez, C. Jard, T. Jéron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In *CAV'96*. LNCS 1102 Springer Verlag, 1996.
6. R. Groz, T. Jeron, and A. Kerbrat. Automated test generation from SDL specifications. In R. Dssouli, G. von Bochmann, and Y. Lahav, editors, *SDL'99 The Next Millenium, 9th SDL Forum, Montreal, Quebec*, pages 135–152, Elsevier, Juin 1999.
7. OSI-Open Systems Interconnection, Information Technology - Open Systems Interconnection Conformance Testing Methodology and Framework - Part 1 : General Concept - part 2 : Abstract Test Suite Specification - part 3 : The Tree and Tabular Combined Notation (TTCN). *International Standard ISO/IEC 9646-1/2/3*, 1992.
8. B. Marre and A. Arnould. Test sequences generation from lustre descriptions: Gatel. In *In Fifteenth IEEE Int. Conf. on Automated Software Engineering (ASE 2000)*, IEEE Computer Society Press, pages 229–237, Grenoble, septembre 2000.
9. M. Mukund. Finite-state automata on infinite input. Technical report, 1996.
10. I. Parissis and J. Vassy. Test des proprietes de surete. In *In Actes du colloque Modelisation de Systemes Reactifs (MSR'01)*, pages 563–578, Hermes, 2001.
11. M. Phalippou. *Relations d'implantations et Hypothèses de Test sur des automates à entrées et sorties*. Thèse de doctorat, Université de Bordeaux, France, 1994.
12. M. Phalippou. Test sequence using Estelle or SDL structure information. In *FORTE'94*, Berne, Oct. 1994.
13. P.Thevenod-Fosse. Unit and integration testing of lustre programs: a case study from the nuclear industry. In *9th European Workshop on Dependable Computing (EWDC-9)*, pages 121–124, Gdansk, Pologne, 14-16 Mai 1998.
14. M. Rabin. *Automata o, Infinite Object and Church' Problem*. Number 13 in Regional Conference series in mathematics. American Mathematical Society, 1972.
15. P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, dec 1998.
16. S. Safra. On the complexity of ω -automata, checking. pages 319–327, 1988.
17. M. Schmitt, B. Koch, J. Grabowski, and D. Hogrefe. Autolink - A Tool for Automatic and Semi-Automatic Test Generation from SDL Specifications. Technical Report A-98-05, Medical University of Lübeck, 1998.
18. R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computation*, 2(1), june 1972.

19. J. Tretmans. Test Generation with Inputs, Outputs, and Quiescence. In T. Margaria and B. Steffen, editors, *Second Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer-Verlag, 1996.
20. P. Wolper, M. Vardi, and A. Sistla. Reasoning about infinite computation paths. In *24th IEEE Symposium of Foundations of Computer Science*, 1983.

Computing Unique Input/Output Sequences Using Genetic Algorithms

Qiang Guo, Robert M. Hierons, Mark Harman, and Karnig Derderian

Department of Information System and Computing, Brunel University
Uxbridge, Middlesex, UB8 3PH, United Kingdom
{qiang.guo,rob.hierons,mark.harman,karnig.derderian}@brunel.ac.uk

Abstract. The problem of computing Unique Input/Output sequences (UIOs) is NP-hard. Genetic algorithms (GAs) have been proven to be effective in providing good solutions for some NP-hard problems. In this work, we investigated the construction of UIOs using GAs. We defined a fitness function to guide the search of potential UIOs and introduce a **DO NOT CARE** character to improve the GA's diversity. Experimental results suggest that, in a small system, the performance of the GA based approaches is no worse than that of random search while, in a more complex system, the GA based approaches outperform random search.

Keywords: FSMs, UIOs, Conformance Testing, Genetic Algorithms, Optimisation

1 Introduction

Finite state machines (FSMs) have been used for modelling systems in various areas such as sequential circuits, software and communication protocols [1,2,4,9,10,11,12]. Four test sequence generation methods are discussed and compared in [4], namely, Transition Tours (T-Method), Unique Input/Output Sequence (U-Method), Distinguishing Sequence (D-Method), and Characterizing Set (W-Method). The last three methods are known as formal methods since they not only check the transitions, but also verify the states. In terms of the fault coverage, the U-, D-, and W-Methods achieve better performance than T-Method does, while exhibiting no significant difference among themselves [4].

Among the formal methods, U-Method is popular since it benefits from the facts: (1) Not all FSMs have a Distinguishing Sequence (DS), but nearly all FSMs have UIOs for each state [6]; (2) The length of a UIO is no longer than DS; (3) While UIOs may be longer than a characterising set, in practice UIOs often lead to shorter test sequences. Unfortunately, computing UIOs is NP-hard [2]. Lee *et al.* [2] note that an adaptive distinguishing sequences and UIOs may be produced by constructing a state splitting tree. However, no rule is explicitly defined to guide the construction of input sequence. Naik [5] proposes an approach to construct UIOs by introducing a set of inference rules. Some minimal length UIOs are found. These are used to deduce some other states' UIOs. A state's UIO is produced by concatenating a sequence to another state, whose

UIO has been found, with this state's UIO sequence. Although it reduces computational complexity, the inference rule inevitably increases a UIO's length, which will consequently add more costs to the forthcoming test.

Genetic algorithms (GAs) have proven efficient in search and optimisation [7] and have shown their effectiveness in providing good solutions to NP-hard problems such as the Travelling Salesman Problem. This work investigates the use of GAs for constructing UIOs from an FSM. An initial population is produced by randomly generating input sequences. This population is used to explore potential UIOs. Based on the state splitting tree, a fitness function is defined to evaluate the quality of the input sequences. This fitness function encourages candidates to split the set of all states into more discrete units and punishes the length of the sequences. Roulette wheel selection and uniform crossover are implemented. Simulation results are also presented and discussed. During the evolutionary computation, good solutions found are stored in a database. This database can be used to preserve the information lost during the computation and to further optimise UIOs' length.

This paper is organised as follows: FSMs are briefly reviewed in section 2. A simple GA is introduced in section 3. Experiments and corresponding results are described in section 4. Finally, conclusions are drawn in section 5.

2 Preliminaries

2.1 Finite State Machines

An FSM M is defined as a quintuple $(I, O, S, \delta, \lambda)$ where I, O , and S are finite and nonempty sets of input symbols, output symbols, and states, respectively; $\delta : S \times I \longrightarrow S$ is the state transition function; and $\lambda : S \times I \longrightarrow O$ is the output function. When the machine is in a current state $s \in S$ and receives an input $a \in I$, it moves to the next state $\delta(s, a)$ and produces output $\lambda(s, a)$.

An FSM M can be viewed as a directed graph $G = (V, E)$, where the set of vertices V represents the state set S of M and the set of edges E represents the transitions. An edge has label i/o where $i \in I$ and $o \in O$ are the corresponding transition's input and output. Figure 1 illustrates an FSM represented by its corresponding directed graph. Throughout this paper where, for some state s and input x , no transition from s with input x is shown, this will be interpreted as the input of x in s leading to an error message being output and the FSM moving to a special error state.

Two states s_i and s_j are said to be *equivalent* if and only if for every input sequence $\alpha \in I^*$ the machine produces the same output sequence, $\lambda(s_i, \alpha) = \lambda(s_j, \alpha)$. Machines M_1 and M_2 are *equivalent* if and only if for every state in M_1 there is a corresponding state in M_2 , and vice versa. A machine is *minimal* (*reduced*) if and only if no two states are equivalent. It will be assumed that any FSM being considered is minimal since any (deterministic) FSM can be converted into an equivalent (deterministic) minimal FSM [3]. An FSM is *completely specified* if and only if for each state s_i and input a , there is a specified next state $s_{i+1} = \delta(s_i, a)$, and a specified output $o_i = \lambda(s_i, a)$. Otherwise, the

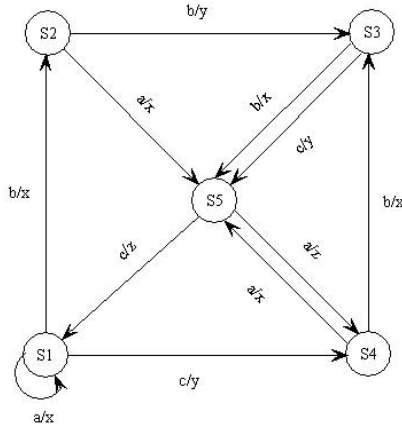


Fig. 1. A Finite State Machine

machine is *partially specified*. An FSM is *strongly connected* if, given any ordered pair of states (s_i, s_j) , there is a sequence of transition that moves the FSM from s_i to s_j .

It will be assumed throughout this article that an FSM is deterministic, minimal, completely specified, and strongly connected.

2.2 Conformance Testing

Given a specification FSM M , for which we have its complete transition diagram, and an implementation M' , for which we can only observe its I/O behaviour ("black box"), we want to test to determine whether the I/O behaviour of M' conforms to that of M . This is called *conformance testing*. A test sequence that solves this problem is called a *checking sequence*. I/O behavioral difference between specification and implementation can be caused by either an incorrect output (an output fault) or an earlier incorrect state transfer (a state transfer fault). The latter can be detected by adding final state check after a transition testing is finished. A standard test strategy is:

1. Homing: Move M' to an initial state s ;
2. Output Check: Apply an input sequence α and compare the output sequences generated by M and M' separately;
3. Tail State Verification: Using state verification techniques to check the final state.

The first step is known as homing a machine to a desired initial state. The second step checks whether M' produces the desired output sequence. The last step checks whether M' is in the expected state $s' = \delta(s, \alpha)$. Three techniques can be used for state verification:

- Distinguishing Sequence (DS)
- Unique Input/Output (UIO)
- Characterizing Set (CS)

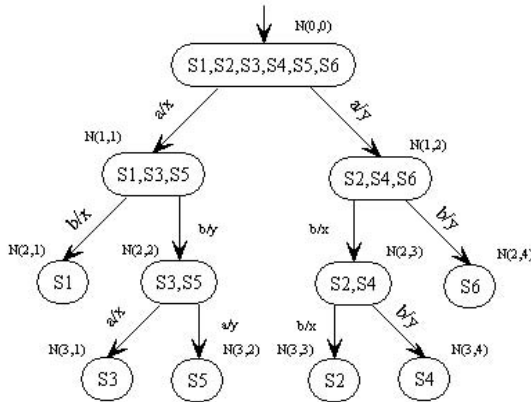


Fig. 2. A state splitting tree from an FSM

A distinguishing sequence is a sequence that produces a different output for each state. Not every FSM has a DS.

A UIO sequence of state s_i is an input/output sequence x/y , that may be observed from s_i , such that the output sequence produced by the machine in response to x from any other state is different from y , i.e. $\lambda(s_i, x) = y$ and $\lambda(s_i, x) \neq \lambda(s_j, x)$ for any $i \neq j$. A DS defines a UIO. While not every FSM has a UIO for each state, some FSMs without a DS have a UIO for each state.

A characterizing set W is a set of input sequences with the property that, for every pair of state (s_i, s_j) , $i \neq j$, there is some $w_i \in W$ such that $\lambda(s_i, w_i) \neq \lambda(s_j, w_i)$. Thus, the output sequences produced by executing each $w_i \in W$ from s_j verifies s_j . This paper will focus on the problem of generating UIOs.

2.3 State Splitting Tree

A state splitting tree is a rooted tree T that is used to construct adaptive distinguishing sequences or UIOs from an FSM. Each node in the tree has a predecessor (parent) and successors (children). The predecessor of the root node, which contains the set of all states, is null. The nodes containing discrete state have empty successor. These node are also known as terminals. A child node is connected to its parent node through an edge labelled with characters. The edge implies that the states in the child node are partitioned from states in the parent node upon receiving the labelled characters. The splitting tree is complete if the partition is a discrete partition.

An example is illustrated in Figure 2 where an FSM (different from the one shown in Figure 1) has six states, namely, $S = \{s_1, s_2, s_3, s_4, s_5, s_6\}$. The input set is $I = \{a, b\}$ while output set $O = \{x, y\}$. The root node is indicated by $N(0, 0)^1$, containing the set of all states. Suppose states $\{s_1, s_3, s_5\}$ produce x simultaneously and arrive at some states when responding to a , while $\{s_2, s_4, s_6\}$

¹ $N(i, j)$: i indicates that the node is in the i^{th} layer from the tree. j refers to the j^{th} node in the i^{th} layer.

produce y . Then $\{s_1, s_3, s_5\}$ and $\{s_2, s_4, s_6\}$ are distinguished by a . Two new nodes rooted from $N(0, 0)$ are then generated, indicated by $N(1, 1)$ and $N(1, 2)$. Continuing to input FSM with b , state initially from $\{s_1\}$ produces x while states initially from $\{s_3, s_5\}$ produce y . Then ab distinguish $\{s_1\}$ from $\{s_3, s_5\}$. Two new nodes rooted from $N(1, 1)$ are generated, denoted $N(2, 1)$ and $N(2, 2)$. The same operation can be applied to $\{s_2, s_4, s_6\}$. Repeating this process, we can get all discrete partitions as shown in Figure 2 (if there is no adaptive distinguishing sequence, this will not happen). A path from a discrete partition node to the root node forms a UIO to the state related to this node. When the splitting tree is complete, we can construct UIOs for each state.

Unfortunately, the problem of finding data to build up the state splitting tree is NP-hard. This provides the motivation for investigating the use of GAs. In the following sections, we will discuss the problem in detail.

3 Apply GA to FSM

3.1 Genetic Algorithms

A genetic algorithm (GA) is a heuristic optimisation technique that simulates natural processes, utilizing selection, crossover, mutation and fitness proportionate reproduction operators. Since Holland's seminal work (1975) [8], it has been applied to a variety of learning and optimisation problems.

A GA starts with a randomly generated population, each element (chromosomes) being a sequence of variables/parameters. Variable values can be represented in binary form, real-number, or even characters. The quality of each chromosome is determined by a fitness function that depends upon the problem considered. Those of high fitness have a greater probability of multiple reproduction while those of low fitness have a greater probability of being rejected.

Crossover and mutation are applied to produce new chromosomes. Crossover exchanges information between randomly selected parent chromosomes by exchanging parameter values to form children. Single-point crossover, multi-point crossover and uniform crossover are three major types. Mutation injects information into the genetic pool by mutating randomly selected parameters according to the preset mutation probability. Mutation prevents genetic pool from premature convergence, namely, getting stuck in local maxima/minima. A flow chart for a simple GA is presented in Figure 3.

3.2 Solution Representation

When applying a GA the first question to be considered is what representation should be used. In this work, the chromosomes in the genetic pool are strings of characters from the input set I . To preserve more information, a *DO NOT CARE* character ' $\#$ ' is also considered. We will explain the reason for using this character in the following sections. When receiving this input, the state of an FSM remains unchanged and no output is produced. Crossover operated on two parent chromosomes swaps characters. When a gene (character) is mutated according to the mutation rate, it is replaced with a character randomly selected from the rest in the input set, including ' $\#$ '.

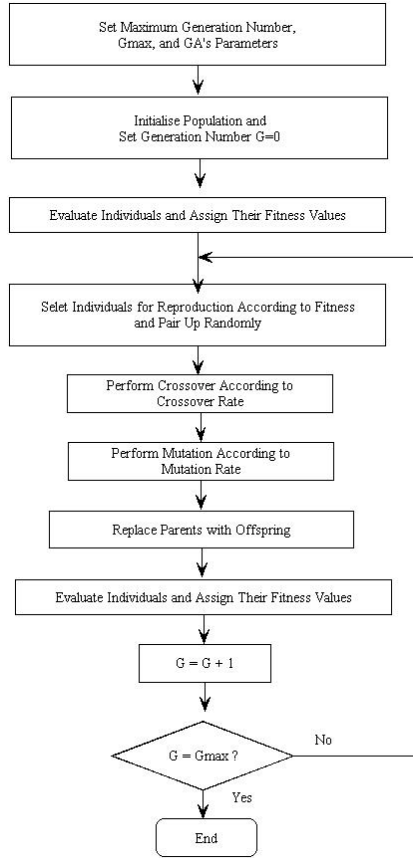


Fig. 3. Flow Chart for a Simple GA

3.3 Fitness Definition

A key issue is to define a fitness function to (efficiently) evaluate the quality of solutions. This function should embody two aspects: (1) Solutions should create as many discrete units as possible. (2) The solution should be as short as possible. The function needs to make a trade-off between these two points. This work uses a function that rewards the early occurrence of discrete partitions and punishes the chromosome's length. An alternative would be to model the number of state partitions and the length of solution as two objectives and then treat them as multi-object optimisation problems.

We define a fitness function in (2) that is derived from (1). While applying an input sequence to an FSM, at each stage of a single input, the state splitting tree constructed is evaluated by equation (1),

$$f_{(i)} = \alpha \frac{x_i e^{(x_i + \delta x_i)}}{l_i^\gamma} + \beta \frac{(y_i + \delta y_i)}{l_i}. \quad (1)$$

where i refers to the i^{th} input character. x_i denotes the number of existing discrete partitions while δx_i is the number of new discrete partitions caused by the i^{th} input. y_i is the number of existing separated groups while δy_i is the number of new groups. l_i is the length of the input sequence up to the i th element (**Do Not Care** characters are excluded). α , β and γ are constants. From the equation it can be seen that, when x_i increases, $f_{(i)}$ will exponentially increase, while, when an input sequence's length l_i increases, $f_{(i)}$ is reduced exponentially. Suppose x_i and l_i change approximately at the same rate, that is $\delta x_i \approx \delta l_i$, as long as $e^{(x_i + \delta x_i)}$ has faster dynamics than l_i^γ , $\frac{x_i e^{(x_i + \delta x_i)}}{l_i^\gamma}$ will still increase exponentially. If no discrete partition is found with the input sequence length increases, the fitness function will decrease dramatically. $\frac{x_i e^{(x_i + \delta x_i)}}{l_i^\gamma}$ thus performs two actions: encouraging the early occurrence of discrete partitions and punishing the increment of an input sequence's length. $\frac{(y_i + \delta y_i)}{l_i}$ will also affect $f_{(i)}$ in a linear way. Comparing to $\frac{x_i e^{(x_i + \delta x_i)}}{l_i^\gamma}$, it plays a less important role. This term rewards partitioning even when discrete classes have not been produced.

After all input characters have been examined, the final fitness value for this input candidate is defined as the average of (1)

$$F = \frac{1}{N} \sum_{i=1}^N f_{(i)}. \quad (2)$$

where N is the sequence's length.

3.4 Tracking Historical Records

Mutation prevents a GA from getting stuck in a local maxima/minima but might also force a GA to jump out of the global maxima/minima when it happens to be there. Solutions provided at the end of evolutionary computation could be good, but need not be the best found during the process. It is therefore useful to keep track of those candidates that have produced good or partially good solutions, and store them for the purpose to further optimise the final solutions.

Consider an example shown in Figure 4. Suppose that a GA produces a UIO sequence U_t for state s_t , forming a path shown in thin solid arrow lines. During the computation, another solution U'_t for s_t has been found, forming a path shown in dotted arrows. The two lines visit a common node at N_4 . U_t has a shorter path than U'_t before N_4 while has a longer path after N_4 . The solution recombined from U_t and U'_t (indicated in figure by thick arrow lines), taking their shorter parts, is better than either of them.

In this work, a database is created to track all candidates that result in the occurrence of discrete partitions. This database is then used to further optimise the final solutions through recombination. Solutions for a state, which are of the same length, are multi-UIOs of this state which can be used in test generation [9,10].

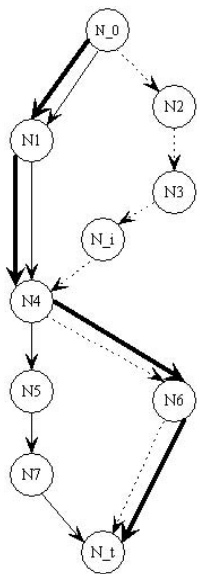


Fig. 4. Solution Recombination

4 Experiments

A set of experiments was devised to test the GA’s performance. The first FSM studied is the one shown in Figure 1. All minimum-length UIOs of all states are presented in Table 1. Roulette Wheel Selection (RWS) and Uniform Crossover (UC) are implemented.

Table 1. UIO Sequences for Figure 1

State	UIOs
s_1	aa/xx, ab/xx, ac/xy, ba/xx bb/xy, ca/yx, cb/yx
s_2	b/y
s_3	ba/xz, bc/xz, ca/yz, cc/yz
s_4	bb/xx, bc/xy
s_5	a/z, c/z

In the first experiment, the input space is $I = \{a, b, c\}$. The parameters are set to²: $ChrLen = 10$, $XRate = 0.75$, $MRate = 0.05$, $PSize = 30$, $MGen = 50$, $\alpha = 0.1$, $\beta = 1.5$, and $\gamma = 5$.

At the end of computation, by looking at the average fitness values (Figure 5), we found that the genetic pool converges quite quick. The curve is comparatively smooth. However, by examining all individuals (Table 2), we found that the

² ChrLen:Chromosome Length; XRate:Crossover Rate; MRate:Mutate Rate; PSize: Population Size; MGen:Max Generation.

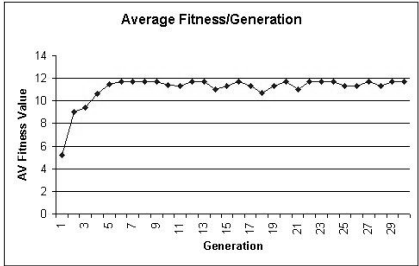


Fig. 5. Input Space $\{a,b,c\}$

Table 2. Final sequences from input space $\{a,b,c\}$

ID	Sequence	ID	Sequence
1	<i>bccaabcacc</i>	16	<i>bbcaabcacc</i>
2	<i>bbbaabaacc</i>	17	<i>bbbaabaaca</i>
3	<i>bbbaabcacc</i>	18	<i>bbcaababcc</i>
4	<i>bcccabaacb</i>	19	<i>bbbcbabbcc</i>
5	<i>bbcbabcacb</i>	20	<i>bbcaabaacc</i>
6	<i>bbbaabcacc</i>	21	<i>bccaabaacc</i>
7	<i>bbcbabcacb</i>	22	<i>bbbaabaacc</i>
8	<i>bbccabcacb</i>	23	<i>bbbcbabbcc</i>
9	<i>bbcbabaacb</i>	24	<i>bbcbabcacb</i>
10	<i>bcbbaabaacb</i>	25	<i>bbbcbaccacc</i>
11	<i>bbcbabcabb</i>	26	<i>bbcaabcacb</i>
12	<i>bbcaabaacc</i>	27	<i>bbbababcc</i>
13	<i>bccaabaacb</i>	28	<i>bbcaabcacb</i>
14	<i>bbbbaabcacc</i>	29	<i>bbccabbacc</i>
15	<i>bbcbabcacb</i>	30	<i>bbbaabbacc</i>

whole population tends to move to the individuals that start with *bb* or *bc*. The population loses its diversity and converges prematurely. Consequently, only a few UIOs have been found $\{b, bb, bc\}$.

This is not what we expected. To keep the genetic pool diverse, we introduced a ***DO NOT CARE*** character ' $\#$ '. When receiving this character, the state of an FSM remains unchanged. The input space is then $\{a, b, c, \#\}$. We keep the same values for all other parameters. The average fitness chart is presented in Figure 6. It can be seen that Figure 6 is not as smooth as Figure 5, but still shows a general tendency to increase. After examining the genetic pool, we found that eleven UIOs, $\{a, b, c, ab, ac, ba, bb, bc, ca, cb, cc\}$, were found (Table 3³). By retrieving the historical records, we also found $\{aa\}$ (Table 4). The GA thus performs well in this experiment.

The reason that crossover operator can be used to explore information is that it forces genes to move among chromosomes. Through recombination of genes, unknown information can be uncovered by new chromosomes. However, the gene movement exerted by crossover operator can only happen among different chro-

³ *Sequence*: candidate sequence. *VS*: minimum-length UIO.

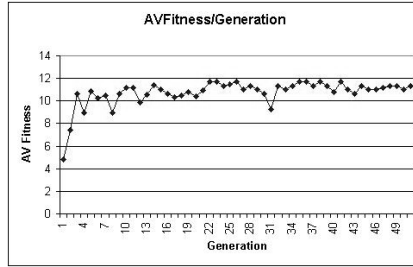


Fig. 6. Input Space $\{a,b,c,\#\}$

Table 3. Final sequences from input space $\{a,b,c,\#\}$

ID	Sequence	VS	ID	Sequence	VS
1	bbbbcbb#c	bb	16	#bcbabcabb	bc
2	ab#bbc#bbc	ab	17	#bcbac#cbcb	cb
3	#bc#caaaba	bc	18	#bb#bbaca	bb
4	#bcbcb#c	bc	19	bc#bb#cb#	bc
5	b#bcaca#cb	bb	20	cbcb#c#ca	cb
6	bb#bca#bb	bb	21	#bc#aabc#c	bc
7	cbacc#ac#b	cb	22	c#a#cacc#c	ca
8	acabaaaacc	ac	23	c#accac#a	ca
9	ba#a#aaa#b	ba	24	#b#bcabbc#	bb
10	#ccaab#acc	cc	25	bb#bba#a#a	bb
11	#cbc#a#ab	cb	26	bacb#c#b#b	ba
12	#bbcca#a#a	bb	27	bb#bb#b#c	bb
13	cccb#####c	cc	28	cb#babcb#b	cb
14	aca#bbaa#b	ac	29	cccc#cc#bb	cc
15	cbb#c#cb#c	cb	30	b#cca#b#c	bc

mosomes. We call it vertical movement. By using a *DO NOT CARE* character, some spaces can be added in a chromosome, which makes it possible for genes to move horizontally. Therefore, *DO NOT CARE* makes the exploration more flexible, and, consequently, can help to keep the genetic pool diverse.

We organised eleven experiments with the same parameters. By examining the solutions obtained in the final genetic pool (historical records are excluded), we evaluated the average performance. Table 5 shows that, in the worst case, 8 out of 12 UIOs are found, which accounts for 66.7%. The best case is 100%. The average is 86.4%.

After examining the solutions from different experiments, we found that *aa* is the hardest UIO to be found while *bb* and *bc* are most frequent ones that occur in the final solutions. By checking Table 1., we found a very interesting fact: a majority of UIOs initially start with *b* or *c*. If individuals happen to be initialised with $ba \times \times \times \times \times \times$, they will distinguish s_1 , s_2 and s_5 in the first two steps, and so achieve high fitness. These individuals are likely to be selected for the next generation. Individuals initialised with $aa \times \times \times \times \times \times$ can distinguish only s_1 and s_5 in the first two steps, and achieve lower fitness values. They are

Table 4. Historical Records

ID	Sequence	VS	Fitness
1	<i>aa#caab#c</i>	<i>aa</i>	6.2784
2	<i>a#aba#bc#c</i>	<i>aa</i>	4.2605

Table 5. Different Experiment Results

Exp.	UIOs Found	Total	Percent(%)
1	8	12	66.7
2	8	12	66.7
3	11	12	91.7
4	11	12	91.7
5	10	12	83.3
6	11	12	91.7
7	11	12	91.7
8	11	12	91.7
9	10	12	83.3
10	12	12	100
11	11	12	91.7
Avg	10.36	12	86.4

less likely to be selected for reproduction. This fact seems to imply that there exist multiple modals in the search space. Most individuals are likely to crowd on the highest peak. Only a very few individuals switch to the lower modals. To overcome this problem, *sharing techniques* might help. The application of such approaches will be left for future work.

We then turn to compare the performance between GA and random search (RS). RS is defined as randomly perturbing one bit in an input sequence. 30 input sequences of ten input characters were randomly generated. We repeated this experiment several times. The results shown in Table 6 are the best ones. From the table it can be seen that 11 out of 12 UIOs (a, b, c, aa, ab, ac, bb, bc, ca, cb, cc) are found over these experiments. Only one is missed (*ba*).

Since the FSM is comparatively simple, and the UIOs are short, it is not difficult to find all UIOs through RS. Thus the GA does not show significant advantages over RS. A more complicated system, shown in Figure 7, is therefore designed to further test GA’s performance. Unfortunately, no existing UIOs are available, which means that we can never be sure that a complete set of UIOs has been found. Hence, we will compare the numbers of UIOs found by using RS and the GA separately.

A total of 50 candidates were used in the experiment. All UIOs found, whether minimum-length or not, are listed to make a comparison. Experiments on both RS and the GA were repeated several times. The solutions presented in Table 7 and Table 8 are the best ones. Table 7 lists the UIOs obtained through RS while Table 8 shows the solutions found by GA. After comparing these two tables, we find that the GA finds many more UIOs than RS does. Both RS and GA easily find the short UIOs. However, for other UIOs the performance

Table 6. Solutions by random search

ID	Sequence	VS	ID	Sequence	VS
1	<i>cabbbcbcac</i>	<i>ca</i>	16	<i>cacacacccc</i>	<i>ca</i>
2	<i>ccbcbbbcbe</i>	<i>cc</i>	17	<i>bcbbabcbca</i>	<i>bc</i>
3	<i>bcccaaabcb</i>	<i>bc</i>	18	<i>cccbbcccb</i>	<i>cc</i>
4	<i>cccccbcbac</i>	<i>cc</i>	19	<i>accbbccacb</i>	<i>ac</i>
5	<i>ccbbaebccc</i>	<i>cc</i>	20	<i>cbecaccccb</i>	<i>cb</i>
6	<i>bccccabcaac</i>	<i>bc</i>	21	<i>acbbcbbbb</i>	<i>ac</i>
7	<i>bcbcbbbcbe</i>	<i>bc</i>	22	<i>cbccccbcb</i>	<i>cb</i>
8	<i>ccbccaabc</i>	<i>cc</i>	23	<i>ccabacbeca</i>	<i>cc</i>
9	<i>bbacccccba</i>	<i>bb</i>	24	<i>accacbabcc</i>	<i>ac</i>
10	<i>cbbbbaccbb</i>	<i>cb</i>	25	<i>aabcbacbbb</i>	<i>aa</i>
11	<i>bcaacccccb</i>	<i>bc</i>	26	<i>cabbacacbc</i>	<i>ca</i>
12	<i>ccbcbbbcbe</i>	<i>cc</i>	27	<i>bcbcccaaac</i>	<i>bc</i>
13	<i>abcbccccab</i>	<i>ab</i>	28	<i>aabcbccbca</i>	<i>aa</i>
14	<i>bcccccbccc</i>	<i>bc</i>	29	<i>bcccaabbbb</i>	<i>bc</i>
15	<i>bbcbcbcbba</i>	<i>bb</i>	30	<i>acaccbaaba</i>	<i>ac</i>

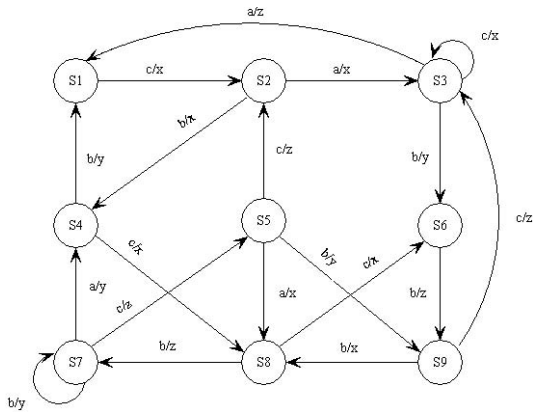


Fig. 7. A more complicated FSM

of GA appears to be much better than that of RS. For example, the GA finds *bcbcc/xxxxx bcbcb/xzzyz* while RS does not.

We also measure the frequency of hitting UIOs. RS is redefined by initialising population routinely. Experimental result show that both methods hit UIOs with the length of 3 or less frequently. However, on hitting those with the length of 4, RS is roughly the half times of GA, while, for those with the length of 5, in the first 30 iterations, RS hits 10 times while GA 27. All these results suggest that, in simple systems, it is possible to obtain good solutions through random search. However, in more complicated systems, especially in those with large input and state spaces, finding UIOs with random search is likely to be infeasible. By contrast, GA seems to be more flexible. In future work, we will apply GA to some examples with high input dimensions, or some real applications such as the *IEEE 802.2 Logical Level Control (LLC) Protocol* (48 inputs & 65 outputs).

Table 7. UIO Sequences By Random Search

State	UIOs
s_1	ca/xx, cb/xx
s_2	aa/xz, ab/xy, acc/xxx, bb/xy bcc/xxx, bcba/xxzy
s_3	a/z, bb/yz, ca/xz, cb/xy, ccb/xy, ccc/xxx
s_4	bc/yx, cba/xzy, ccb/xxz cbca/xzzx, cbcc/xzzz
s_5	ab/xz, acb/xxz, bb/yx bcc/yzx, cacc/zxxx, cb/zx
s_6	bb/zx, bcc/zzx
s_7	a/y, bb/yy, bcc/yzz, bcbcyzyz, cbc/zyz, cc/zz
s_8	bb/zy, bcc/zzz, bcbcyzyz, cbca/xzzz, cbcc/xzzx
s_9	bb/xz, ca/zz, cc/zx

Table 8. UIO Sequences Found by GA

State	UIOs
s_1	ca/xx, cb/xx
s_2	aa/xz, ab/xy, aca/xxz, acb/xy acc/xxx, bb/xy, bcc/xxx, bcbcc/xxzzz
s_3	a/z, bb/yz, ca/xz, cb/xy, ccb/xy, ccc/xxx
s_4	cbbcb/zxxyx, bc/yx, cba/xzy, cbb/xzy, ccb/xxz, cbca/xzzx, cbcc/xzzz, cbcbc/xzzyz
s_5	ab/xz, acb/xxz, bb/yx, bca/yzz bcc/yzx, cab/zxy, cb/zx
s_6	bb/zx, bca/zzz, bcc/zzx
s_7	a/y, ba/yy, bb/yy, bca/yzx bcc/yzz, cab/zxz, cbb/zyx, cc/zz cbc/zyz, cc/zz, bcbcyzyz
s_8	ba/zy, bb/zy, bca/zzx, bcc/zzz, cbb/xzx, cbca/xzzz, cbcc/xzzx, cbbcb/xzzyz
s_9	bb/xz, ca/zz, cbb/zyz, cc/zx

5 Conclusion

In this paper, we investigated GA's performance in computing UIOs from an FSM. We showed that the fitness function can guide the candidates to explore potential UIOs by encouraging the early occurrence of discrete partitions while punishing length. We showed that using *DO NOT CARE* character can help to improve the diversity in GAs. Consequently, more UIOs can be explored.

The simulation results in a small system showed that, in the worst case, 67% of the minimum-length UIOs have been found while, in the best case, 100%. On the average, more than 85% minimum-length UIOs were found from the model under the test. In a more complicated system, GA found many more UIOs than random search. The GAs was much better, than random search, at finding the longer UIOs. These experiments and figures suggest that GAs can provide good solutions on computing UIOs.

We also found that some UIOs were missed with high probability. This may be caused by their lower probability distribution in the search space. Future work will consider this problem.

References

1. A.V.Aho, A.T.Dahbura, D.Lee and M.U.Uyar, "An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours" *IEEE Trans, Communications*, vol.39, No.3, pp.1604-1615, 1991.
2. D.Lee and M.Yannakakis, "Testing Finite State Machines: State Identification and Verification" *IEEE Trans, Computers*, vol.43, No.3, pp.306-320, 1994.
3. D.Lee and M.Yannakakis, "Principles and Methods of Testing Finite State Machines - A Survey" *Proceedings of IEEE*, vol.84, No.8, pp.1090-1122, 1996.
4. D.P.Sidhu, and T.K.Leung, "Formal Methods for Protocol Testing: A Detailed Study" *IEEE Transactions on Software Engineering*, Vol.15, No.4, April 1989.
5. Kshirasagar Naik, "Efficient Computation of Unique Input/Output Sequences in Finite-State Machines" *IEEE/ACM Transactions on Networking*, vol.5, No.4, pp.585-599, Aug., 1997.
6. Z.Kohavi, "Switching and Finite Automata Theory", *New York: McGraw-Hill*, 1978.
7. D.E.Goldberg, "Genetic Algorithms in Search, Optimization, and Machine Learning". Reading, MA: Addison-Wesley, 1989.
8. J.H.Holland, "Adaptation in Natural and Artificial Systems". Ann Arbor, MI, University of Michigan Press, 1975.
9. Y.N.Shen, F.Lombardi, and A.T.Dahbura, "Protocol Conformance Testing Using Multiple UIO Sequences" *Proc, Ninth IFIP WG6.1 Int. Symp. on Protocol Specif., Test., and Verif.*, 1989.
10. Bo Yang and Hasan Ural, "Protocol Conformance Test Generation Using Multiple UIO Sequences with Overlapping" *ACM SIGCOMM 90: Communications, Architectures, and Protocols*, Twente, The Netherlands, Sep.24-27, pp118-125. North-Holland, The Netherlands.
11. R.E.Miller and S.Paul, "On the Generation of Minimal-Length Conformance Tests for Communication Protocols" *IEEE/ACM Transactions on Networking*, Vol.1, No.1, Feb, 1993.
12. Xiaojun Shen and Guogang Li, "A new protocol conformance test generation method and experimental results" *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied computing*, pp.75-84, Kansas City, Missouri, United States, 1992.

Automatic Generation of Test Purposes for Testing Distributed Systems

Olaf Henniger¹, Miao Lu², and Hasan Ural²

¹ Fraunhofer Institute Secure Telecooperation,
Rheinstr. 75, 64295 Darmstadt, Germany
henniger@sit.fraunhofer.de

² School of Information Technology and Engineering, University of Ottawa,
Ottawa, Ontario, K1N 6N5, Canada
{mlu, ural}@site.uottawa.ca

Abstract. In this paper, we present an algorithm for generating test purpose descriptions in form of MSC's from a given labeled event structure that represents the behavior of a system of asynchronously communicating extended finite state machines. The labeled event structure is a non-interleaving behavior model describing the behavior of a system in terms of the partial ordering of events.

1 Introduction

For testing whether the behavior of an implementation conforms to its designated behavior, test cases are to be generated from the specification describing the designated behavior. The behavior of a distributed system can be specified e.g. using a system of asynchronously communicating state machines. This model forms the basis e.g. of the standardized formal description technique SDL [1]. A system of communicating state machines implicitly describes all, possibly non-deterministic, sequences of inputs and outputs that constitute the designated behavior. Since the number and length of these sequences are infinite in general, it is impossible to test each and every possible behavior and we face the problem to select a set of meaningful test cases, i.e. a test suite, that allows to discover as many implementation errors as possible at an acceptable cost. This forms the main problem in generating conformance test suites.

Each test case in a test suite specifies the actions required to achieve a specific test purpose. The test purpose in each case is to check a particular requirement implied by the given specification [2]. A test purpose can be expressed e.g. by prose text or by a message sequence chart (MSC) describing the behavior to be checked. MSC's are a standardized description technique for the graphical representation of the temporal ordering of interactions between components of a distributed system [3].

The existing methods for test generation from formal specifications can be roughly classified into methods with explicit test purposes and methods with implicit test purposes. Methods with explicit test purposes require information about the test purposes

as input in addition to the specification. These methods offer much flexibility to the test designer and ensure that only executable test cases are generated. However, they require considerable manual efforts to define appropriate test purposes and do not guarantee systematic test coverage. Methods with implicit test purposes provide test cases for test purposes that they tacitly assume. These methods generally guarantee a complete test coverage w.r.t. the implicit test purposes. However, most of them are applicable only to restricted classes of specifications, e.g. to specifications containing a single state machine, and they may result in very large test suites.

Since practically relevant system specifications may be voluminous and complicated, a manual generation and maintenance of test purposes and test cases is too time-consuming and error-prone. It is therefore highly desirable to have test generation methods with implicit test purposes or at least methods for the automatic generation of test purposes. Only few test generation tools, like Autolink [4] in the Telelogic Tau toolset and TestComposer [5] in the Telelogic ObjectGeode toolset are applicable to complex multi-process SDL specifications of a realistic size. These two tools are based on interleaving models for the behavior of the specified system. This entails that the same behavior may be represented by different paths of the reachability graph, which differ only in the order of execution of concurrent actions.

Our approach uses a non-interleaving model (labeled event structure) to alleviate the state-space explosion problem. In [6], an algorithm for transforming a system of asynchronously communicating state machines into a labeled event structure is given and a method with implicit test purposes for generating test cases in Concurrent TTCN from a labeled event structure is proposed. To combine the advantages of methods with implicit test purposes with those of methods with explicit test purposes, this paper aims at the automatic generation of test purposes from labeled event structures. From a labeled event structure, test purpose descriptions are generated in form of MSC's by interpreting the parallel paths of the labeled event structure as MSC's. These MSC's can serve as input for test generation tools with explicit test purposes, preferably if those support test generation for distributed testers, as proposed in [7, 8].

The rest of this paper is organized as follows. Section 2 introduces the prerequisites necessary for the proposed approach. Section 3 deals with the generation of test purposes from a labeled event structure. Throughout the paper, a simple sliding-window protocol serves as an example. Section 4 gives a summary and outlook.

2 Preliminaries

2.1 Communicating State Machines

A system of asynchronously communicating state machines is an obvious semantic model for specifications in SDL. Therefore, they form the starting point for our approach.

A *system of asynchronously communicating state machines* is composed of a set of state machines and a set of perfect (i.e. without loss or reordering of messages) FIFO queues that connect the state machines with each other and with their environment.

We consider each state machine as an extended finite state machine (EFSM) without enabling conditions for transitions. In general, an EFSM is a finite state machine extended by additional variables that may be used in enabling conditions for transitions, in calculations to be carried out during the execution of transitions, or for representing message parameters. An EFSM with enabling conditions can be transformed into an equivalent one without enabling conditions if the variables influencing the executability of transitions take on only a finite number of discrete values. An algorithm for this transformation is given in [9, 6]. This condition is not unduly restricting the class of specifications for which the algorithm for generating test purposes is applicable since it is a common practice for a test designer to determine the context by assigning values to control variables and to parameters of input messages.

We do not require that the EFSM's form a closed system, but allow open interfaces to the environment. To limit the complexity imposed by the environment, the following assumption is made. The environment is assumed to put a message into a queue if and only if the associated EFSM is ready to consume it. Hence, a transition with a trigger input (excited by a message from the environment) is assumed always to be enabled as soon as the EFSM reaches the start state of that transition. This assumption is common practice in test generation for conformance testing, which is, in contrast to robustness testing, confined to the behavior foreseen in the specification.

Let $m = \langle M, Q \rangle$ be a system of asynchronously communicating EFSM's composed of a set of EFSM's $M = \{m_1, \dots, m_n\}$ and a set of message queues $Q = \{q_1, \dots, q_r\}$. A *global state* of m is an $(n+r)$ -tuple $g = (s_{m_1}, \dots, s_{m_n}, c_{q_1}, \dots, c_{q_r})$ consisting of the states $s_{m_1}, \dots, s_{m_n}$ of the EFSM's $m_1, \dots, m_n$ and the contents $c_{q_1}, \dots, c_{q_r}$ of the queues $q_1, \dots, q_r$.

Fig. 1 shows a system of asynchronously communicating EFSM's modeling a simple sliding-window protocol. The EFSM's t , r , and m model the transmitter and receiver protocol entities and the transmission medium, respectively. To facilitate de-

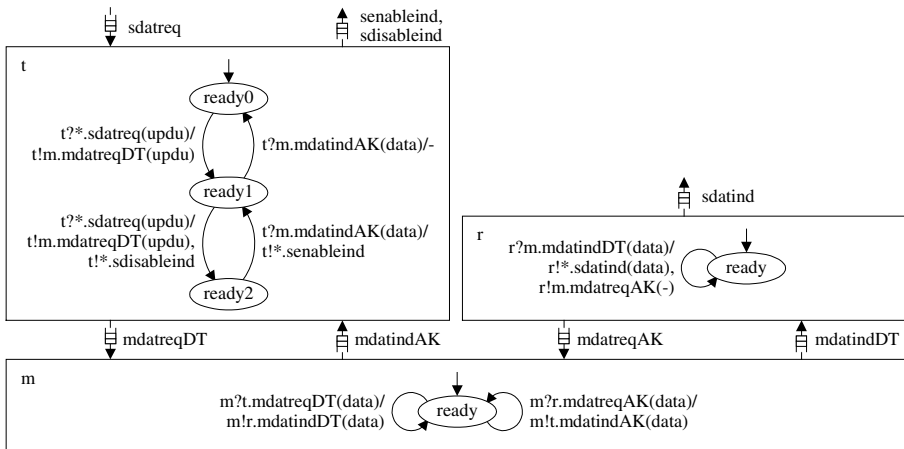


Fig. 1. Example of a system of asynchronously communicating EFSM's

nominating the location of actions, we denote input and output actions in the form loc (“?” | “!”) rem “.” msg [“(” par {“,” par } “)”] where loc denotes the EFSM where the action is located, “?” indicates an input action (receiving msg), “!” indicates an output action (sending msg), rem is the name of the remote EFSM sending msg (in case of an input action) or receiving msg (in case of an output action), msg is a message, and par is a message parameter. “*” stands for the environment.

The example protocol provides the service to transmit data from a user on the transmitter side to a user on the receiver side while protecting the receiver against overload by attending to acknowledgements. If the number of messages for which the acknowledgement is outstanding (the window size) reaches its maximum (2 for simplicity), the protocol entity on the transmitter side indicates to the user that no more messages can be transmitted for the time being. When the protocol entity on the transmitter side receives an acknowledgement, then the number of messages for which the acknowledgement is outstanding is decremented and new messages can be transmitted again. The transmission medium is reliable and does not lose, corrupt, add, or reorder messages.

2.2 Labeled Event Structures

Definitions. For generating test purposes, we would like to have a model that explicitly describes the behavior of a distributed system in terms of the order of events. A labeled event structure fulfils this requirement. A system of asynchronously communicating EFSM’s can be “unfolded” into a labeled event structure. In a labeled event structure concurrent events are not linearized as in a reachability tree, but lined up side by side without order relation. Event structures were introduced in [10] as being like acyclic Petri nets without backward branching and with the places removed.

A basic element of labeled event structures are actions. The same action can occur various times in a system run, each time forming a new, distinguishable event. The actions in the labeled event structures correspond to actions in the underlying systems of asynchronously communicating EFSM’s: they model the inputs and outputs, calculations in the context variables, and the setting, resetting and expiration of timers.

A labeled event structure over a set of actions A is a quadruple $\langle E, \preceq, \#, l \rangle$ where

- E is a finite set of events;
- $\preceq \subseteq E \times E$ is a partial order relation in E , called *causality relation*, such that for all $e \in E$ the set $\{e' \in E \mid e' \preceq e\}$ is finite (i.e., the number of causal predecessors of any event is finite);
- $\# \subseteq E \times E$ is an irreflexive and symmetric relation in E , called *conflict relation*, such that $\forall e, e', e'' \in E ((e \# e' \wedge e' \preceq e'') \Rightarrow e \# e'')$ (i.e., conflicts are inherited: if an event e is in conflict to some event e' , then it is also in conflict to all causal successors of e');
- $l : E \rightarrow A$ is a *labeling function* assigning an action to each event.

$e \preceq e'$ means that if the events e and e' both happen, then e must happen before e' . $e \# e'$ means that the events e and e' cannot happen both in a single run of the

system. If two events are neither causally related nor in conflict, then they are *concurrent* to each other and both can occur in any order: either e before e' , e and e' at the same time or e' before e . All events occurring in the same EFSM are either causally related or in conflict, but not concurrent to each other.

A labeled event structure is interpreted informally as follows: An event can occur if all its causal predecessors have occurred and no conflicting event has occurred yet.

Let $m_{les} = \langle E, \preceq, \#, l \rangle$ be a labeled event structure and $C \subseteq E$ be a subset of events of m_{les} . C is *causally closed* if $\forall e \in C \forall e' \in E (e' \preceq e \Rightarrow e' \in C)$. C is *conflict-free* if $\forall e, e' \in C (\neg(e \# e'))$. C is a *configuration* of m_{les} if it is causally closed and conflict-free. That means, a configuration is a set of events that have occurred by some stage in executing a labeled event structure. The *necessary configuration* $[e]$ of an event $e \in E$ of a labeled event structure m_{les} is the subset of events that includes e and all causal predecessors of e , but not any other events, i.e. $[e] = \{e' \in E \mid e' \preceq e\}$. All events that have to occur prior to an event e belong to the necessary configuration of e . Events that are concurrent to e do not belong to the necessary configuration of e .

Each configuration of a labeled event structure constructed from a system of asynchronously communicating EFSM's corresponds to a global state of the system. The *final state* $gs(C)$ of a configuration of a labeled event structure constructed from a system of asynchronously communicating EFSM's m is the global state of m reached after all events $e \in C$, but no other events have occurred.

The construction of a labeled event structure from a system of asynchronously communicating EFSM's can be cut off at different points, leading to different event structures. The labeled event structure obtained by unfolding a system of asynchronously communicating EFSM's as much as possible is referred to as *the* labeled event structure of the system. Only a complete prefix of the labeled event structure of a system of asynchronously communicating EFSM's is constructed in our approach. A *prefix* of the labeled event structure $\langle E, \preceq, \#, l \rangle$ is a labeled event structure $\langle E', \preceq', \#, l' \rangle$ induced by a causally closed subset of events $E' \subseteq E$. A prefix of the labeled event structure of a system of asynchronously communicating EFSM's is *complete* if it contains a configuration C for each reachable global state g of the system such that

- $g = gs(C)$, i.e., g is represented by C , and
- for each transition $g \xrightarrow{\mu/\omega} g'$ enabled in g with $\omega = v_1 \dots v_p$, the prefix contains a configuration $C' = C \cup \{e, e_1, \dots, e_p\}$ with $e, e_1, \dots, e_p \notin C$ and $l(e) = \mu$, $l(e_1) = v_1, \dots, l(e_p) = v_p$.

A *maximal configuration* is a configuration to which no more events of the complete prefix of the labeled event structure can be added. An event e is a *maximal event* of a configuration C if there does not exist any $e' \in C$ with $e \preceq e'$.

Graphical Representation. A labeled event structure is represented as a graph where vertices represent events, directed edges lead to the immediate causal successors of an event, and undirected dashed edges connect events in immediate conflict. Next to an event e its label $l(e)$ is indicated. The graph of a labeled event structure is cycle-free. The set of events occurring in the same EFSM induces a subgraph that is a directed tree. We draw the subgraphs for the parallel EFSM's with their edges in parallel.

Fig. 2 shows a complete prefix of the labeled event structure of the system of asynchronously communicating EFSM's in Fig. 1. The complete prefix is annotated with the global states at cut-off points and at recursion points. The prefix may be expanded by appending the sub-structures starting with the corresponding global states to the cut-off points.

Construction of a Labeled Event Structure. The algorithm for unfolding systems of asynchronously communicating state machines into labeled event structures resembles the reduced reachability analysis from [11, 12], yet the results are taken down in the form of event structures. These reduced reachability algorithms aim at alleviating the state explosion problem and yield reduced reachability trees whose nodes represent only certain reachable global states and whose directed edges represent sets of transitions concurrently executable in a certain global state. Intermediate global states reached while executing a set of concurrent transitions are not explicitly represented.

For finding cut-off points suitable for a complete prefix of the labeled event structure, [6] takes up an approach for coping with the state explosion problem in analyzing Petri nets with finite state space [13, 14]. The main idea can be outlined as fol-

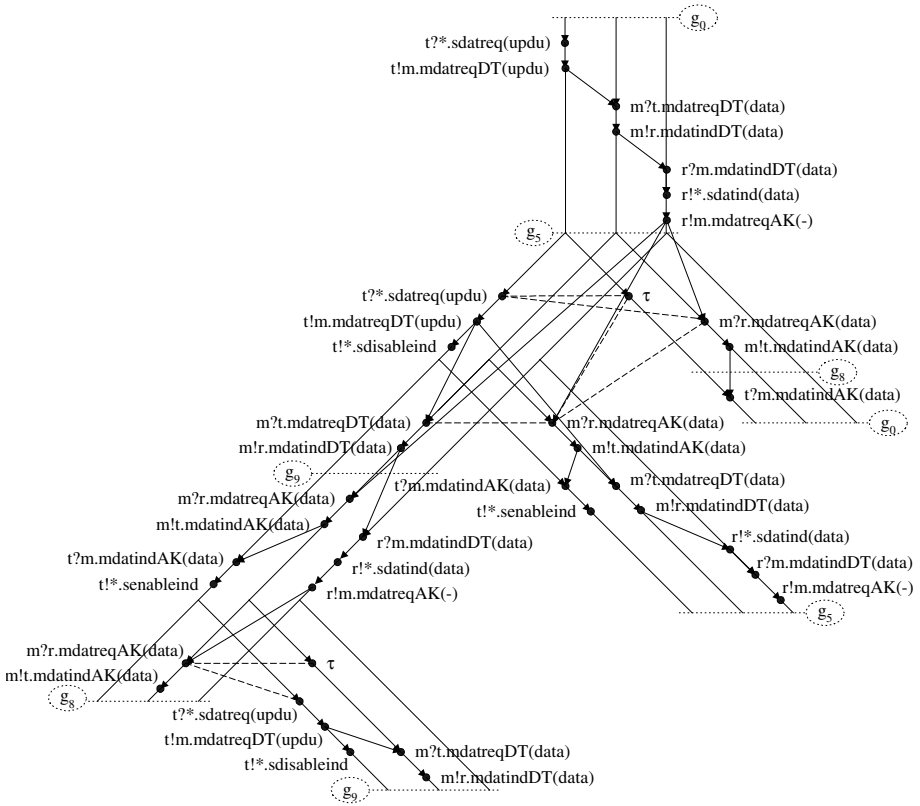


Fig. 2. A complete prefix of the labeled event structure for the example from Fig. 1

lows: An event is a cut-off event if its necessary configuration has the same final state as the necessary configuration of another event already contained in the unfolding. The unfolding can be cut off after these events since all events appended after the cut-off events would lead to states already covered by the prefix. [13] presents an algorithm for constructing a finite prefix of the unfolding of a Petri net. The prefix is complete with respect to the reachable markings of the Petri net. As the complete prefix of the unfolding constructed after [13] is sometimes larger than necessary, [14] improves the algorithm such that a complete prefix is constructed that is minimal in a certain sense. The algorithm in [14] is applicable to n -safe Petri nets with $n \geq 1$.

How a testing equivalent labeled event structure or its complete prefix can be constructed from a given system of asynchronously communicating state machines is treated in detail in [6]. The approach is applicable if all state machines of the system have a finite number of states and all queues of the system are bounded. This is not an undue restriction as in many cases an unbounded growth of the state space can be avoided by appropriate design criteria.

3 Test Generation Approach

3.1 Starting Point

Starting point for the generation of test purposes is a complete prefix m_{les} of the labeled event structure constructed from a system of asynchronously communicating EFSM's m . It forms a semantic model of a given specification of the implementation under test (IUT) embedded in a test context and hence models the behavior perceivable at the system boundaries during black box testing. The events that involve an interaction with the environment represent events occurring at points of control and observation (PCO's), i.e. at points where a test system may interact with IUT and test context.

As illustrated in Fig. 2, cut-off points and recursion points of m_{les} are labeled with the corresponding global states of the system m in order to characterize the possible continuations of the behavior.

3.2 Implicit Test Purposes and Test Coverage

As each maximal configuration of a complete prefix of the labeled event structure represents a significant behavior, it is desirable that a test suite tries to execute each maximal configuration of the complete prefix at least once. We also regard it as sufficient to execute each maximal configuration of the complete prefix once. This limits the size of the test suite. At the cut-off points of the complete prefix, behavior that has been encountered before is repeated anyway. By generating a larger test suite covering more than the complete prefix, one attains a higher test coverage and a higher degree of confidence that the IUT will operate free of error when actions are executed repeatedly. In principle, if the IUT is regarded as a black box, it remains uncertain whether or not it will operate free of error when the same actions are executed next

time. Based on knowledge about the inner structure of an implementation (e.g. about the reliability of the operating system, about the programming language used, etc.), however, often it is inferred that an implementation will work free of error any number of times if it does so at least once.

For each maximal configuration of the complete prefix of the labeled event structure of a system of asynchronously communicating EFSM's a test case is to be generated. Its test purpose is to check the behavior described by the corresponding maximal configuration.

By covering each maximal configuration of the complete prefix, we achieve all-nodes coverage (or all-events coverage) w.r.t. the complete prefix. We do not necessarily achieve all-transition coverage w.r.t. the underlying system of asynchronously communicating EFSM's due to the fact that the EFSM's may contain transitions that are never triggered in normal interaction with the other EFSM's of the system.

3.3 Algorithm for Generating Test Purposes

Overview. The goal is to construct a set of test purpose descriptions in form of MSC's from the complete prefix m_{les} of the labeled event structure of a system of asynchronously communicating EFSM's. The generation of test purposes is carried out in the following steps, which are implemented as a prototype tool [15]:

1. Identify all maximal configurations of the complete prefix;
2. Restrict the maximal configurations to events occurring at the PCO's;
3. For each restricted maximal configuration, check whether it is included in another maximal configuration, and if so, eliminate it from the set of maximal configurations;
4. Format the maximal configurations as MSC's.

Identification of Maximal Configurations. In order to obtain the set of events belonging to a maximal configuration, we start from the cut-off points and follow the causality relation backwards to the roots. First, all the maximal events at a cut-off point are put into an initially empty event queue and into an initially empty event set. Loop while the event queue is not empty, get the first event from the queue and put all its predecessors that have not been put into the event set yet into the event queue and into the event set. When the loop terminates, all the events belonging to the maximal configuration have been put into the event set. After a maximal configuration is obtained, it is added to the set of maximal configurations.

The identification of all maximal configurations is described in pseudo-code below. $mconf_i$ denotes a maximal configuration from the set of all maximal configurations $MCONF$. $cutoff_i$ denotes the set of maximal events at a cut-off point. $m_{les}.CUTOFF$ denotes the set of all cut-off points of m_{les} . $pred_queue$ is the queue data structure for processing the predecessor events.

```

MCONF := ∅;
for all  $cutoff_i \in m_{les}.CUTOFF$  do
   $mconf_i := \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$ ;
   $pred\_queue := \emptyset$ ;

```

```

for all  $e_j \in \text{cutoff}_i$  do
   $mconf_i.E := mconf_i.E \cup e_j$ ;
   $\text{put}(\text{pred\_queue}, e_j)$ ;
endfor;
while not  $\text{empty}(\text{pred\_queue})$  do
   $ev := \text{get}(\text{pred\_queue})$ ;
  for all  $e_j \in ev.\text{predecessors}$  do
    if  $e_j \notin mconf_i.E$  then
       $mconf_i.E := mconf_i.E \cup e_j$ ;
       $\text{put}(\text{pred\_queue}, e_j)$ ;
    endif;
  endfor;
endwhile;
 $MCONF := MCONF \cup mconf_i$ ;
endfor;

```

Restriction to Events at PCO's. The restriction has to be done because only the events occurring at the system boundaries can be controlled or observed during black box testing.

The process of restricting a maximal configuration to events occurring at the PCO's consists of checking all events in the maximal configuration and omitting the events for which the remote communication partner is not the environment. In restricting the maximal configurations, the transitivity of the causality relation has to be preserved.

Below, the restriction to events occurring at PCO's is described in pseudo-code.

```

for all  $mconf_i \in MCONF$  do
  for all  $e_j \in mconf_i.E$  do
    if  $(l(e_j).\text{rem} \neq \text{"*"})$  then
       $mconf_i.E := mconf_i.E \setminus \{e_j\}$ ;
    endif;
  endfor;
endfor;

```

Inclusion Checking. In order to get a minimal set of maximal configurations, each configuration is checked, after restricting it to the events occurring at the PCO's, whether it is included in another configuration in the obtained set of restricted maximal configurations. If so, it is removed from the set.

Below is pseudo-code for the inclusion checking.

```

for all  $mconf_i \in MCONF$  do
  for all  $mconf_j \in MCONF$  ( $i \neq j$ ) do
    if  $mconf_i.E \subseteq mconf_j.E$  then
       $MCONF := MCONF \setminus mconf_i$ ;
    endif;
  endfor;
endfor;

```

Formatting Maximal Configurations as MSC's. The test purpose descriptions can be laid out as process-level MSC's or as system-level MSC's.

A maximal configuration of a complete prefix of the labeled event structure of a system of asynchronously communicating EFSM's can be straightforwardly interpreted as a process-level MSC with one instance for every EFSM associated with a PCO and one instance for every PCO. This way, the concurrency of different EFSM's remains unhidden. Fig. 3 shows a test purpose description for the example in Fig. 1 in form of a process-level MSC. The interfaces to the environment on transmitter and receiver side are referred to as PCO_t and PCO_r , respectively.

On the other hand, test purpose descriptions for Autolink are stored as system-level MSC's containing only one instance for the whole system under test and one instance for every PCO [4]. This way, the concurrency of different components of the system is hidden. To make the output of our tool applicable as input to Autolink, our tool also generates system-level MSC's.

To generate system-level MSC's, we have to linearize the maximal configurations. A linearization of a partially ordered event set is a total order on this event set that contains the partial order. A linearization can be derived from a configuration by adding arbitrary ordering constraints to the partial order of the configuration.

In order to get a linearization for a maximal configuration restricted to the PCO's, first, all the events in the maximal configuration are put into an initially empty event queue. Loop while the event queue is not empty, get the first event from the queue, check whether all its predecessor events are already included in the linearization. If so, add the event to the linearization. The first event added to the linearization will be an initial event without any predecessor. If not yet all predecessor events are in the linearization, put the event again into the event queue.

The linearization of maximal configurations is described in pseudo-code below. $mconf_i.seq$ denotes the linearization of a maximal configuration. e_queue is the queue data structure for processing the events.

```

for all  $mconf_i \in MCONF$  do
   $mconf_i.seq := \emptyset$ ;
   $e\_queue := \emptyset$ ;
  for all  $e \in mconf_i.E$  do
     $put(e\_queue, e)$ ;
  endfor;
  while not  $empty(e\_queue)$  do
     $ev := get(e\_queue)$ ;
    if  $(ev.predecessors \subseteq mconf_i.seq)$  then
       $mconf_i.seq := concatenate(mconf_i.seq, ev)$ ;
    else  $put(e\_queue, ev)$ ;
    endif;
  endwhile;
endfor;

```

Fig. 4 shows a test purpose description for the example in Fig. 1 in form of a system-level MSC.

3.4 Data Flow Aspects

The complete prefix of the labeled event structure constructed from a set of asynchronously communicating EFSM's without enabling conditions for transitions may contain variables that are used for representing message parameters, for buffering values, or for calculations to be carried out during the execution of transitions. It does not contain enabling conditions for the occurrence of events. Therefore, the occurrence of each configuration in the complete prefix is feasible.

Some data flow oriented test selection criteria that have been introduced for specifications represented by directed graphs can be transferred to labeled event structures. These criteria establish associations between definitions and uses of variables. Such associations are identified by tracking variables through the specification, following them as they are modified, until they are ultimately used in outputs or to compute values for other variables. The criteria require that each of these associations be examined at least once during testing. The intuition behind the selection of tests based on the coverage of data flow associations is that faults in a system may lead to incorrect values and, as a result of propagation through computations, an error may show up at the system's output.

The all-uses coverage criterion is satisfied w.r.t. the complete prefix of the labeled event structure if for each variable defined in the complete prefix each subsequent use of that variable (i.e., each def-use pair) is covered by at least one test. Even if there are no definitions without subsequent use within the underlying system of asynchronously communicating EFSM's, not necessarily all variables defined within the complete prefix of the labeled event structure are used within the complete prefix. To achieve full all-uses coverage, our tool appends sub-structures of the complete prefix to the cut-off points whenever necessary and possible for covering definitions without use within the complete prefix.

4 Summary and Outlook

The approach introduced in this paper generates test purpose descriptions in form of MSC's from a non-interleaving model, viz. from a complete prefix of the labeled event structure constructed from a system of asynchronously communicating EFSM's.

This model alleviates the state-explosion problem and preserves true concurrency. The size of the resulting test suite is restricted in a suitable way. The approach is applicable to a large class of specifications. The executability of the test cases is ensured.

A prototype tool implementing the approach described in this paper is available [15]. Its input is generated by the prototype tool for constructing a complete prefix of the labeled event structure from a generalized model of asynchronously communicating state machines [6]. Together with the corresponding system specification, the output of the test purpose tool is intended as input for test generation tools that take explicit descriptions of test purposes as input.

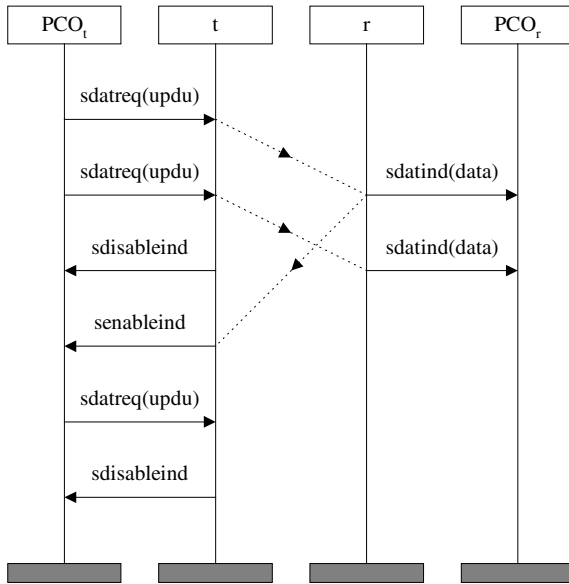


Fig. 3. A test purpose description as process-level MSC

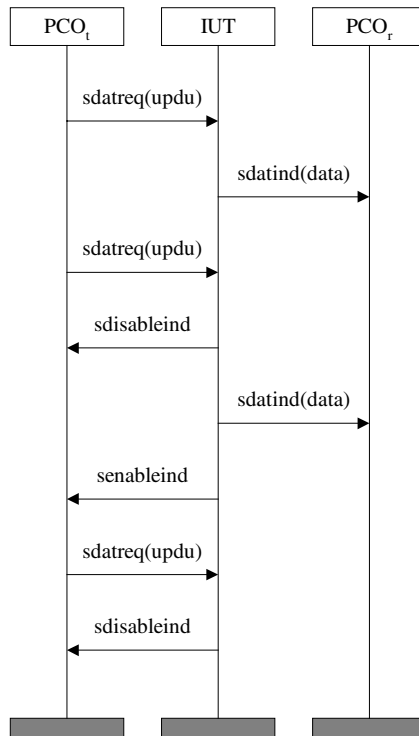


Fig. 4. A test purpose description as system-level MSC

As an alternative to the textual and tabular presentation formats, the new version of TTCN (TTCN-3) [16] allows describing tests in a graphical presentation format based on a subset of MSC's. The MSC's generated by our tool describe only the desired behavior to be checked in a test case. Therefore, the generated MSC's are used as test purpose descriptions. MSC's for defining test cases have to describe the behavior of the test components interacting with IUT and test context via the PCO's and to cover possible behavior alternatives, which would lead to inconclusive or fail verdicts. The verdicts have to be included in a test case as well. The direct generation of MSC test cases from a labeled event structure is an area of future work.

Acknowledgements

This work is supported in part by the Natural Sciences and Engineering Research Council of Canada.

References

1. Specification and Description Language (SDL-2000), ITU-T Recommendation Z.100, 1999
2. Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 1: General concepts, International Standard ISO/IEC 9646-1, 2nd edition, 1994
3. Message Sequence Chart (MSC), ITU-T Recommendation Z.120, 1999
4. B. Koch, J. Grabowski, D. Hogrefe, M. Schmitt, "Autolink – a tool for automatic test generation from SDL specifications", in Proc. of the 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques, Boca Raton, Florida, USA, 1998
5. Kerbrat, T. Jéron, and R. Groz, "Automated test generation from SDL specifications", in [17], pp. 135–151
6. O. Henniger, "Test generation from specifications in Estelle and SDL", Ph.D. thesis (in German), Brandenburg Technical University of Cottbus, Germany, 2002, Shaker Verlag
7. J. Grabowski, B. Koch, M. Schmitt, D. Hogrefe, "SDL and MSC based test generation for distributed test architectures", in [17], pp. 389–404
8. C. Jard, "Synthesis of distributed testers from true-concurrency models of reactive systems", *Information and Software Technology* 45, pp. 805–814, 2003
9. O. Henniger, A. Ulrich, H. König, "Transformation of Estelle modules aiming at test case generation", in A. Cavalli, and S. Budkowski (eds.), Proc. of IWPTS'95, Evry, France, 1995, pp. 36–51
10. M. Nielsen, G. Plotkin, and G. Winskel, "Petri nets, event structures and domains, Part I", *Theoretical Computer Science* 13, 1981, pp. 85–108
11. M. Itoh and H. Ichikawa, "Protocol verification algorithm using reduced reachability analysis", *Transactions of the IECE of Japan* E 66 (2), pp. 88–93, 1983
12. N. Arakawa and T. Soneoka, "A test case generation method for concurrent programs", in J. Kroon, R.J. Heijink, E. Brinksma (eds.), Proc. of IWPTS'91, Leidschendam, The Netherlands, 1991, pp. 95–106

13. K.L. McMillan, “A technique of state space search based on unfolding”, *Formal Methods in System Design* 6 (1), 1995
14. J. Esparza, S. Römer, and W. Vogler, “An improvement of McMillan’s unfolding algorithm”, *Formal Methods in System Design* 20 (3), pp. 285–310, 2002
15. M. Lu, “Generation of tests from labeled event structures”, M.Sc. thesis, University of Ottawa, Canada, May 2003
16. Methods for Testing and Specification; The Testing and Test Control Notation Version 3; Part 3: TTCN-3 Graphical Presentation Format (GFT), ETSI Standard ETSI ES 201 873-3, February 2003
17. R. Dssouli, G.v. Bochmann, and Y. Lahav (eds.), Proc. of the 9th SDL Forum, Montréal, Québec, Canada, 1999

Interaction Testing in an Embedded System Using Hardware Fault Injection and Program Mutation*

Ahyoung Sung and Byoungju Choi**

Department of Computer Science & Engg, Ewha Woman's University, Seoul, Korea
{aysung, bjchoi}@ewha.ac.kr

Abstract. An embedded system is a combination of hardware and software subsystems. Interaction between these two subsystems may lead to unexpected behavior when faults are present in either. An effective technique is required to detect the presence of such “interaction faults” in an embedded system. We propose a test data selection technique for interaction testing in the embedded system using hardware fault injection and mutation test criteria. The proposed technique simulates hardware faults as software faults and uses these to mutate the software component. The mutants so created are then used as a means to select test data that differentiates the original program from the mutants. An experimental evaluation of the proposed technique is also presented.

1 Introduction

An embedded system [1] is comprised of hardware and software components. Examples include nuclear power plant system, medical devices, electric home appliances, and practically most devices in common use. The importance of testing embedded systems has grown with the rapid increase in their complexity. Especially, for safety-critical embedded systems such as nuclear power plants and medical devices, testing the entire system involves high cost and risk. Even when there no faults are detected when testing either hardware or software alone, the combined system combining can lead to an unexpected situation that requires using testing technique directed at the detection of interaction faults.

We propose a test data selection technique as an application of the well-known mutation based test data selection techniques [2,3] to detect faults due to the interaction between the hardware and software components. Program mutation is a technique to select input test data that differentiates the original program from its mutants. In the proposed technique, we first identify parts of hardware, software and their interaction from the system requirements. Next, a simulator (a program) is prepared to simulate the behavior of the embedded system. Hardware faults are simulated as software faults in the simulator. These software faults are then injected into the simulator to generate the mutated programs. The input data, which differentiates the simulator

* This work was partially supported by grant No.R04-2003-000-10139-0 from the Basic Research Program of the Korea Science & Engineering Foundation. This work was partially supported by the Ministry of Science and Technology under the KNICS Projects. This work was partially supported by University IT Research Center Project.

** Also Department of Computer Sciences, Purdue University W.Lafayette, IN47906, USA.

program from its mutant, is selected as the test data that we expect to be capable of detecting the interaction faults in the embedded system. We apply our technique to a safety-critical embedded system, namely DPPS (Digital Plant Protection System) [4] in order to select test data to detect interaction faults and to show the effectiveness of the test data selected.

The remainder of this paper is organized as follows. In Section 2, we provide an overview of DPPS. In Section 3 we explain the test data selection technique to detect interaction faults in the embedded system. In Section 4 we illustrate the application of our technique to DPPS, and analyze its effectiveness technique in Section 5. Our conclusions, summary and questions of further interest are presented in Section 6.

2 Digital Plant Protection System

An overview of DPPS is provided in this section DPPS is a reactor protection system, which is organized into four independent channels. When a problem occurs, DPPS is the protection system that emits a trip signal to halt the system and bring it into a safe state. As shown in Figure 1, DPPS consists of a bistable processor, a coincidence processor, and an initiation processor.

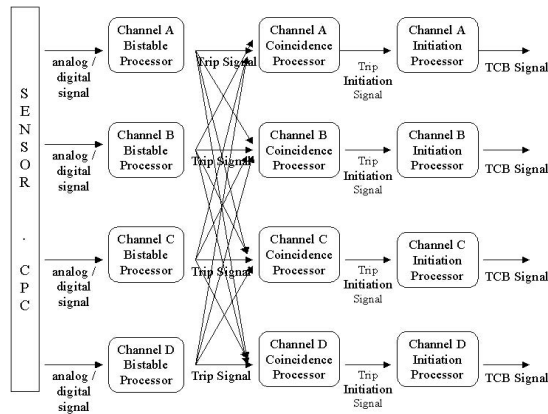


Fig. 1. Structure of DPPS

The bistable processor receives an analog signal from the sensor to monitor the status of nuclear plant and a digital signal from CPC. The bistable processor compares the two signals entered to the set point. If either signal is higher or lower than the set point, it sends out the trip signal to the coincidence processor. The coincidence processor verifies the output of the bistable processor, and sends out the trip initiation signal to the initiation processor when having the trip signal in more than 2 out of 4 channels. After receiving the trip initiation signal, the trip initiation processor sends out the TCB signal to command to stop the system. Since DPPS is composed of 4 identical channels. We use one channel, namely channel A as an example in this paper.

3 Test Data Selection Technique

Figure 2 shows the proposed test data selection procedure. As shown, we analyze the specification for the target system to identify parts of hardware, software, and their interactions. Also generated is Program S to simulate the behavior of the system under study (also referred to as the “target system”). Having identified a hardware fault f_H that can occur in the target system, f_H transformed into an equivalent software fault. We generate Program Pf_H by injecting the software fault into Program S. Next we construct test data that can differentiate between Programs S and Pf_H .

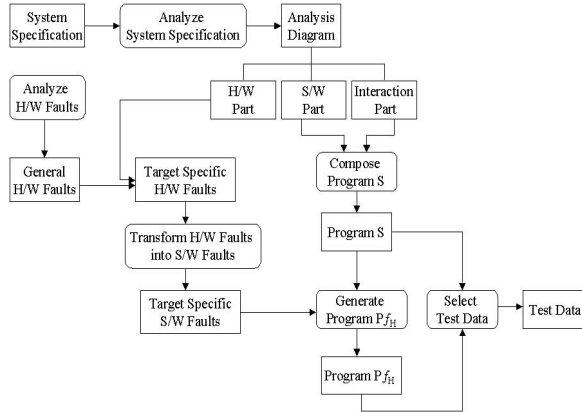


Fig. 2. Procedure of test data selection

3.1 Analysis of System Specification

An analysis diagram describes the target system as hardware, software with the target system’s behavior implemented, and the interactions between them. As shown in Figure 3, an analysis diagram illustrates the operational cycle of the embedded system.

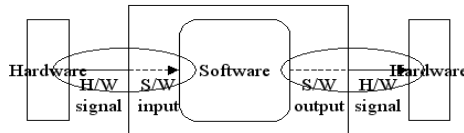


Fig. 3. Analysis diagram

Upon entry into the target system, the hardware signal is transformed into the software input. The software component in the target system sends out output, which is transformed into a hardware signal that is the output of the target system. In Figure 3, the hardware component in the target system is drawn as a rectangle, software to be embedded in hardware is shown as a rounded-rectangle, the hardware signal as a solid line, the software I/O as a dotted line, and the interaction is as an ellipse.

3.2 Generation of Program S

The program S that simulates the behavior of target system is described in C programming language, where S consists of not only the behavior of target system but also interaction part between hardware and software.

3.3 Generation of Program Pf_h

After a hardware fault is transformed into an equivalent software fault, we generate program Pf_h by injecting the software fault into program S. To do this, first it is necessary to classify the kinds of hardware faults. As shown in Table 1, the hardware faults are classified as Stuck-at 0 [5,6,7], Stuck-at 1 [5,6,7], Bridging fault [5,8], Open fault [5], Bit-flip fault [5,7], Power surge fault [5,9,10], and Spurious current fault [5,6,9,10,11]. In Table 1, each classified hardware fault, description of each hardware fault, and the impact of the hardware fault are listed.

Table 1. Hardware faults

Fault Type	Description	Impact
Stuck-at 0	The result value is fixed to 0.	The result value always comes out to be 0.
Stuck-at 1	The result value is fixed to 1.	The result value always comes out to be 1.
Bridging	When there are more than two crossing lines, the number of lines crossed varies.	The number of lines crossed is verified.
Open	Resistance on either the line or the block occurs due to the bad connection.	The value associated to the line or the block is modified to different value.
Bit-flip	The bit flips.	The variable based on the modified bit is verified.
Power surge	Inconsistent power is supplied.	The problem not solely lies on the value of the specific location, but almost the entire system is affected.
Spurious current	Exposures to heavy ion.	The problem not solely lies on the value of the specific location, but almost the entire system is affected.

The hardware fault can be modeled as a software fault somewhere in Program S. Two steps are necessary to transform the hardware fault present in the embedded system into an equivalent software fault. The first step is to identify where in Program S the hardware fault affects and to determine the location where to inject an equivalent software fault. The second step is to decide how to transform the hardware fault injection target into software fault.

Determination of Hardware Fault Injection Target. The part where hardware fault affects Program S, the software component, is defined as a variable at the very last location of in S; this is the location where we expect the impact of the hardware fault.

If there is a hardware fault, it is more likely for an unexpected value to propagate to the “last variable.” Here, by “last variable” of S we refer to the last location in S where each variable is defined. To detect the interaction faults, we propose a test data selection technique based on the hardware fault injection and mutation test criteria. Here the Fault Injection Target (FIT) becomes the last location in S to be defined for the variable affected by hardware fault. For example, because the location having impacts from the fault is known for the Open fault, Bridging fault, Bit-flip fault, Stuck-at 0 fault, and Stuck-at 1 fault as shown in table 1, it is possible to expect how the fault affects S . Therefore the variable located at the last end of S is decided as FIT. Since the Spurious current fault, and Power surge fault affect the entire system, the last output variable becomes FIT.

Generation of Pf_H Program by Transforming the Hardware Fault into Software Fault. In order to transform the hardware faults into software faults, the hardware faults identified in Table 1 are transformed into the software faults through the code patch. The code patch means to add the program code to modify the FIT variable located in the last variable in S , which is affected by hardware faults. The code patch method to transform the hardware faults into the software faults are following:

[Code Patch Method]

- If the variable of FIT has the digital signal value, negate the variable value to change.
- If the variable of FIT has the analog signal value, change the variable value with random value.

3.4 Selection of Test Data

The objective of this paper is to select test data capable detecting faults that are possible to occur due to the interaction of hardware and software. The test data T of program S is defined as input data, which makes the program S output different from the Pf_H output.

$$T = \{t \mid S(t) \neq Pf_{Hi}(t), 1 \leq i \leq n, n = \text{number of FIT} \}$$

4 An Application Example to DPPS

In this Section, an example of applying the proposed test data selection technique to the DPPS channel A is described here.

4.1 Analysis of DPPS Specification

Analyzing the DPPS Specification to express in Figure 3 notation, hardware consists of the board with Intel 80c196 processor and I/O devices, where the software embedded to the board was implemented with features of a bistable processor, coincidence processor, and an initiation processor. Identifying DPPS' hardware, software, and the interaction part from Figure 4 is as shown in Table 2.

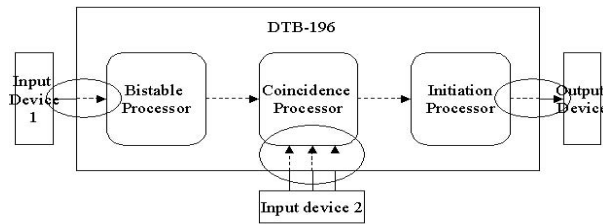


Fig. 4. Analysis diagram of DPPS channel A

Table 2. Hardware, Software, and Interaction Parts of DPPS channel A

	Hardware	Software	Interaction
H1	Input device1	N/A	N/A
H2	Bistable processor	Implementation of bistable processor	H1→H2
H3	Input device 2	N/A	N/A
H4	Coincidence processor	Implementation of coincidence processor	H3→H4
H5	Initiation processor	Implementation of initiation processor	H5→H6
H6	Output device	N/A	N/A

4.2 Generation of Program S

Implement the simulation Program S in C language program for the DPPS channel A's behavior of a bistable processor, coincidence processor, initiation processor, and the interaction part.

In case of DPPS, change the hardware and software interaction part for the hardware signal input to software input value, which can be processed in a bistable or a coincidence processor. Moreover, to send out the result value processed in an initiation processor to output device, the software output is changed to hardware signal. If the interaction part identified in Table 2 were described in program S, it would be as shown in Table 3. In case of the part H1→H2, which the hardware signal entered by the input device is transformed into the software input value, is to be expressed in software, it becomes `analog[i].measure`, `digital[i].trip`.

Table 3. Interaction Parts in DPPS channel A's software

Interaction part	Interaction parts in DPPS software
H1→H2	<code>analog[i].measure</code> , <code>digital[i].trip</code>
H3→H4	<code>analog[i].trip_bistable[B]</code> , <code>digital[i].trip_bistable[B]</code>
	<code>analog[i].trip_bistable[C]</code> , <code>digital[i].trip_bistable[C]</code>
	<code>analog[i].trip_bistable[D]</code> , <code>digital[i].trip_bistable[D]</code>
H5→H6	<code>analog[i].initiation</code> , <code>digital[i].initiation</code>

4.3 Generation of Program Pf_H

Determination of the Hardware Fault Injection Target. The hardware faults expected to occur in DPPS and the location of the FIT when these faults are transformed into software are as shown in Table 4. Table 4 identifies the hardware fault occurred in DPPS channel A according to the hardware fault classification taxonomy in Table 1.

The hardware location of each fault occurrence (the assigned number in Table 2) and the fault injection target when these faults are injected in software are shown in Table 4. In case of DPPS channel A, there were total of 76 FITs identified.

Table 4. FITs in DPPS channel A's software

H/W Fault	Hardware	FIT in DPPS software
Stuck-at 0	H1, H2	f_{H1} :analog[i].measure, f_{H2} :digital[i].trip
	H2	f_{H3} :analog[i].trip_set, f_{H4} :digital[i].trip_set f_{H5} :analog[i].opby_set, f_{H6} :digital[i].opby_set
	H3, H4	f_{H7} :analog[i].trip_bistable[B], f_{H8} :digital[i].trip_bistable[B] f_{H9} :analog[i].trip_bistable[C], f_{H10} :digital[i].trip_bistable[C], f_{H11} :analog[i].trip_bistable[D], f_{H12} :digital[i].trip_bistable[D]
	H4	f_{H13} :analog[i].trip_coincidence, f_{H14} :digital[i].trip_coincidence
	H5, H6	f_{H15} :analog[i].trip_initiation, f_{H16} :digital[i].trip_initiation
Stuck-at 1	H1, H2	f_{H17} :analog[i].measure, f_{H18} :digital[i].trip
	H2	f_{H19} :analog[i].trip_set, f_{H20} :digital[i].trip_set f_{H21} :analog[i].opby_set, f_{H22} :digital[i].opby_set
	H3, H4	f_{H23} :analog[i].trip_bistable[B], f_{H24} :digital[i].trip_bistable[B] f_{H25} :analog[i].trip_bistable[C], f_{H26} :digital[i].trip_bistable[C] f_{H27} :analog[i].trip_bistable[D], f_{H28} :digital[i].trip_bistable[D]
	H4	f_{H29} :analog[i].trip_coincidence, f_{H30} :digital[i].trip_coincidence
	H5, H6	f_{H31} :analog[i].trip_initiation, f_{H32} :digital[i].trip_initiation
Bridging	H2, H4	f_{H33} :analog[i].measure, f_{H34} :digital[i].trip
		f_{H35} :analog[i].trip_bistable[B], f_{H36} :digital[i].trip_bistable[B]
		f_{H37} :analog[i].trip_bistable[C], f_{H38} :digital[i].trip_bistable[C]
		f_{H39} :analog[i].trip_bistable[D], f_{H40} :digital[i].trip_bistable[D]
Open	H1, H2	f_{H41} :analog[i].measure, f_{H42} :digital[i].trip
	H2	f_{H43} :analog[i].trip_set, f_{H44} :digital[i].trip_set f_{H45} :analog[i].opby_set, f_{H46} :digital[i].opby_set
	H3, H4	f_{H47} :analog[i].trip_bistable[B], f_{H48} :digital[i].trip_bistable[B] f_{H49} :analog[i].trip_bistable[C], f_{H50} :digital[i].trip_bistable[C] f_{H51} :analog[i].trip_bistable[D], f_{H52} :digital[i].trip_bistable[D]
	H4	f_{H53} :analog[i].trip_coincidence, f_{H54} :digital[i].trip_coincidence
	H5, H6	f_{H55} :analog[i].trip_initiation, f_{H56} :digital[i].trip_initiation
Bit-flip	H1, H2	f_{H57} :analog[i].measure, f_{H58} :digital[i].trip
	H2	f_{H59} :analog[i].trip_set, f_{H60} :digital[i].trip_set f_{H61} :analog[i].opby_set, f_{H62} :digital[i].opby_set
	H3, H4	f_{H63} :analog[i].trip_bistable[B], f_{H64} :digital[i].trip_bistable[B] f_{H65} :analog[i].trip_bistable[C], f_{H66} :digital[i].trip_bistable[C] f_{H67} :analog[i].trip_bistable[D], f_{H68} :digital[i].trip_bistable[D]
	H4	f_{H69} :analog[i].trip_coincidence, f_{H70} :digital[i].trip_coincidence
	H5, H6	f_{H71} :analog[i].trip_initiation, f_{H72} :digital[i].trip_initiation
Power Surge	H5, H6	f_{H73} :analog[i].trip_initiation, f_{H74} :digital[i].trip_initiation
Spurious current	H5, H6	f_{H75} :analog[i].trip_initiation, f_{H76} :digital[i].trip_initiation

Generation of the Program Pf_H by Transforming the Hardware Fault into Software Fault. The 76 FITs identified in Table 4 are transformed into software faults by utilizing the code patch. Figure 5 shows the Program S and Pf_{H3} . The Pf_{H3} is generated by patching code f_{H3} to ‘analog[0].trip_set=0;’ in S. The code for Program S and Pf_{H3} , except for the part of FIT with the code patched, are the same.

<pre> ... void set_point(analog_struct *analog, digital_struct *digital){... analog[0].trip_set = 1127; analog[1].trip_set = 500; ...} ... void input(){... for(i=0; i<D_MAX; i++){ ... for(j=1; j<CH_MAX; j++){ clear_screen(); lcdMoveCursor(0,0); lcdPutString("Enter the"); ...} } digital[0].trip = 0; /* Interaction Fault*/ } ... </pre> S	<pre> ... void set_point(analog_struct *analog, digital_struct *digital){... analog[0].trip_set = 1127; analog[0].trip_set = 0; /* Stuck at 0*/ analog[1].trip_set = 500; ...} ... void input(){... for(i=0; i<D_MAX; i++){ ... for(j=1; j<CH_MAX; j++){ clear_screen(); lcdMoveCursor(0,0); lcdPutString("Enter the"); ...} } digital[0].trip = 0; /* Interaction Fault*/ } ... </pre> Pf_{H3}
--	--

Fig. 5. S and Pf_{H3}

4.4 Selection of DPPS Test Data

DPPS channel A in Program S consists of 5 input variables including 3 analog variables and 2 digital variables, and output of it is a trip signal. The test data T is selected as the data of input variable, which makes difference in the outputs of Program S and Pf_H . In Table 5, let t be one of the test data in T, then the expected output of t becomes O(t), the t applied to S is S(t), and the result of t applied to Pf_{H3} is $Pf_{H3}(t)$. In Table 5, NT denotes that there occurs No Trip, and T implies the occurrence of Trip.

Table 5. O(t), S(t), $Pf_{H3}(t)$ for test data t

Input variable	Analog_0	Analog_1	Analog_2	Digital_0	Digital_1
t	0	600	200	1	0
Output variable	Trip_A0	Trip_A1	Trip_A2	Trip_D0	Trip_D1
S(t)	NT	T	NT	NT	NT
$Pf_{H3}(t)$	T	T	NT	NT	NT
O(t)	NT	T	NT	T	NT

For example, when the test data $t = (0, 600, 200, 1, 0)$ is applied to S and Pf_{H3} , there are different outputs as shown below.

- S(t) = (NT, T, NT, NT, NT), where trip occurred only in Analog_1.
- $Pf_{H3}(t)$ = (T, T, NT, NT, NT), where trips occurred in Analog_0, and Analog_1.

Therefore t is the input data capable of differentiating the S output from the Pf_{H3} output, which is one of the test data selectable by our proposed technique. As mentioned in Table 5, the expected output O(t) is (NT, T, NT, T, NT), which is different from S(t) (NT, T, NT, NT, NT). The meaning of t is that it can be the data to detect

the interaction faults between hardware and software in DPPS channel A. In case of the correct program, there should not be any ‘digital[0].trip=0;’ but there is the hardware and software interaction fault, ‘digital[0].trip=0;’ in the bold letter part of Program S in Figure 5. For ‘digital[0].trip=0;’ the hardware signal read from the input device needs to be transformed into software signal, and because it is also the part where there is interaction between hardware and software, it is possible for the hardware and software interaction fault occurrence in DPPS. Therefore it is recognizable that the selected test data t is the test data capable of detecting the interaction fault, ‘digital[0].trip=0;’.

Therefore the goal of our proposed technique is to select test data capable of detecting the interaction faults between hardware, and the predominance of our technique is described in Section 5 based on the result of the experiment.

5 Empirical Studies and Analysis

5.1 Environments and Procedures of Empirical Studies

Environments. The DPPS channel A example mentioned in Section 4 is simulated by 4 different testers, where the test is executed by embedding Program S1, S2, S3, and S4 written in C programming language into the hardware board that uses an Intel 80c196 processor.

The environment in Figure 6 includes the input entered by the user being sent to the board through the PC and board input cables, and the results of board calculation being displayed to the user as output. Here the power for the board is supplied by the PC power taken from the power cable.

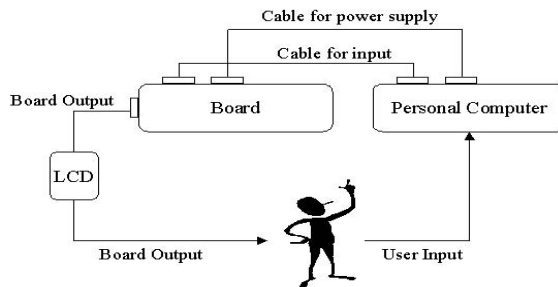


Fig. 6. Experimental Environments.

Procedure. Let S be the C language program for DPPS channel A without faults. Program S_i has different interaction fault as shown in Table 6, and the procedure for each $Expi$ ($1 \leq i \leq 4$) is as following:

Step1. Collect test data for the Program S_i by applying white-box test technique. Generate the test data set T_g , which satisfies the criteria for all-node, decision, c-use, p-use, and all-use in this experiment by using Automatic Test Analysis for C (ATAC)[12,13].

- Step2.* Generate the Program Pf_H by inserting hardware fault, which is transformed into software fault for the identified FIT in the Program Si shown in Table 2.
- Step3.* Execute the startup function related to hardware configuration to operate the board.
- Step4.* Add the master code to the Program S , the Program Si , and the Program Pf_H .
- Step5.* Compile the code with the Cross Compiler.
- Step6.* Mount the hex formatted code on the target board.
- Step7.* Select the test data set T as the set Tg collected from Step1, which differentiate the Program Si and the Program Pf_H .

5.2 Analysis

To prove the predominance of our proposed technique, we compared the number of test data and the fault detection rate to cover 100% of our technique. As shown in Table 6, the data for Exp1, Exp2, Exp3, and Exp4 are indicated with interaction faults present in Si , the number of test data, the number of test data with finding fault, and the fault detection rate for the random selection, existing technique and the proposed technique. In Table 6, 'Random' means select test data Randomly, 'Existing' means existing technique is selecting test data by ATAC, and 'Proposed' means the Proposed technique. TD means the number of Test Data consumed, FTD is the number of Test Data with Fault detected, and ATD indicates the number of Average Test Data.

Table 6. Experimental data

Expi	Interaction faults in Si	Item	Experimental Data (Fault Detection Rate)						
			Random	Existing	Proposed				
Exp1	Fault: Line # 591 analog[1].trip_initiation=1; There should not be a correct code.	TD	32	22	2	3	4	5	6
					13	50	89	26	2
		FTD	6	6	12	32	72	18	0
		ATD	32 (0.19)	22 (0.27)	3.74 (0.75)				
Exp2	Fault: Line # 424 analog[0].measure=0; There should not be a correct code.	TD	32	20	2	3	4	5	6
					9	54	97	32	0
		FTD	5	5	9	34	78	18	0
		ATD	32 (0.16)	20 (0.25)	3.79 (0.72)				
Exp3	Fault: Line # 456 digital[0].trip_bistable[1]=1; There should not be a correct code.	TD	32	23	2	3	4	5	6
					5	49	105	27	0
		FTD	6	6	5	38	92	22	0
		ATD	32 (0.19)	23 (0.26)	4.01 (0.84)				
Exp4	Fault: Line # 443 digital[0].trip=0; There should not be a correct code.	TD	32	20	2	3	4	5	6
					0	33	58	26	0
		FTD	7	7	0	15	43	22	0
		ATD	32 (0.22)	20 (0.35)	3.94 (0.68)				

Comparison of Number of Test Data. The randomly selected number of test data, the number of test data for our proposed technique is shown in Table 6 and Figure 7 bar graph. The randomly selected number of test data, the number of test data based on the existing technique, and the number of test data for our proposed technique are shown in Table 6 and Figure 7 bar graph.

As shown in Table 6 Exp1, the random number of test data is 32, the number of test data Tg to cover 100% of the existing technique is 22, and the data which differ-

entiate S1 and all the Pf_H generated from S1 that is to cover 100% of our proposed technique can be analyzed in detail as follows: in case of covering 100% of our proposed technique with 2 test data involves 13 kinds, with 3 test data involves 50 kinds, with 4 test data involves 89 kinds, with 5 test data involves 26 kinds, and with 6 test data involves 2 kinds, to have the average number of test data covering 100% of our proposed technique as 3.79, which requires relatively small number of test data compared to the random and existing test techniques. Like Exp1, the average number of test data for Exp2, Exp3, and Exp4 are 3.79, 4.01, and 3.94 respectively, which indicates very low for our proposed technique compared to the random and existing test data selection techniques.

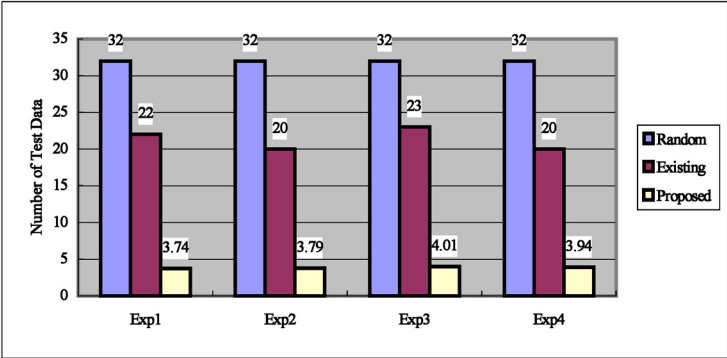


Fig. 7. Average number of test data

Comparison of Fault Detection Rate. The fault detection rate was measured with targeting S1, S2, S3, and S4 that were generated by artificially inserting the interaction fault into Program S. The fault detection rate is shown in the small parenthesis of Table 6 and in the bar graph of Figure 8. The fault detection rate is defined as following:

$$\text{Fault Detection Rate} = \frac{\text{Number of Test Data Found with Faults}}{\text{Total Number of Test Data Generated}}$$

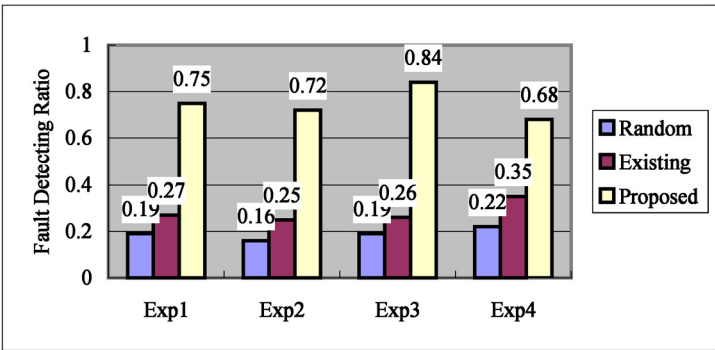


Fig. 8. Fault Detection Rate

In case of Exp1, the random test data selection technique had 6 out of 32 test data detected faults, and therefore the fault detection rate is 0.19. The existing technique had 6 out of 22 test data detected faults to lead 0.27 for the fault detection rate. Our proposed technique on the other hand had 135 out of total of 180 generated test data founded faults to derive 0.75 fault detection rate. Like Exp1, the fault detection rate for Exp2, Exp3, and Exp4 indicated as 0.72, 0.84, and 0.68 respectively, which is illustrated by experiment in the bar graph of Fig 8, to realize the higher fault detection rate of our proposed technique compared to the random and existing techniques.

6 Summary and Future Research

Increasing complexity of embedded system has raised the need for extensive testing of embedded systems. Testing an embedded system is expensive. In addition making any modification to an embedded system for the purpose of testing, such as required by program mutation, is difficult. This is especially true of the system used in the DPPS example mentioned in this paper.

In this paper, we proposed a test data selection technique to detect hardware and software interaction faults in an embedded system by utilizing the hardware fault injection and program mutation and applied our proposed technique to DPPS channel A. In order to select the test data capable of detecting the DPPS interaction faults, we analyzed the different parts of DPPS including hardware, software, and the interaction between them, and also generated the Program S in C programming language to simulate the behaviors of target system. After extracting the hardware faults from the analyzed hardware component, the location for FIT was determined, where the extracted faults were to be inserted in program S. From the code patch, hardware faults were inserted into program S by transforming it into software faults, and the programs Pf_H were generated. Next, the input data capable of differentiating the output of Programs S and Pf_H were constructed to detect the faults caused by the interaction of hardware and software.

“Good” test data means is one that has a high fault detection rate and is small in size. We conducted a case study to investigate the effectiveness of test data generated by the proposed technique. In this study, we used an Intel 80c196 prototyping board. The results from this study indicate that proposed has a high fault detection rate with the small number of test data.

In the future, we plan to implement other proposed techniques into a tool, and thus automate the generation of test data to detect faults caused due to hardware-software interactions in an embedded. In addition, instead of limiting the application to DPPS for the case study as in this paper, we plan to apply the proposed tool to various embedded systems and conduct experiments to investigate the performance of our technique.

Acknowledgements

The authors thank Professor Aditya Mathur at Purdue University for his assistance in reviewing an early version of this paper and making valuable suggestions.

References

1. E. A. Lee, *What's Ahead for Embedded Software?*, IEEE Computer, pp 18-26, September, 2000.
2. Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward, *Hints on test data selection: Help for the practicing programmer*. IEEE Computer, 11(4): 34-41, April 1978.
3. M. E. Delamaro, J. C. Maldonado, and A. P. Mathur., "Integration Testing Using Interface Mutation", In Proceedings of International Symposium on Software Reliability Engineering (ISSRE '96), pages 112--121, April 1996.
4. UCN 3&4 Final Safety Analysis Report, Volume 11, Korea Electric Power Corporations.
5. Mei-Chen Hsueh, T.K. Tsai and R.K. Iyer, *Fault Injection Techniques and Tools*, Computer, Apr, pp75-82.1997.
6. J.V.Carreira, D. Costa and J.G. Silva, *Fault Injection spot-checks computer system dependability*, IEEE Spectrum, Aug, 1999.
7. J.V. Carreira, H. Madeira and J.G. Silva, *Xeption : A technique for the Experimental Evaluation of Dependability in modern computers*, IEEE Transaction on Software Engineering, VOL 23, No.2, Feb, 1998.
8. M.Dalpasso, M. Favalli, P. Olivo and B.Ricco, *Fault Simulation of parametric Bridging Faults in CMOS ICs*, IEEE Transaction on computer aided design of integrated circuits and systems, VOL:12. no. 9, Sept, 1993.
9. J.A. Clark, and D.K. Pradhan, *Fault injection: a method for validating computer-system dependability*, IEEE Computer, VOL: 28 Issue: 6, pp47-56, June 1995.
10. G. Miremadi and J. Torin, *Evaluating processor-behavior and three error-detection mechanisms using physical fault-injection*, IEEE Transaction on Reliability, VOL:44 Issue 3, pp441-454, Sept,1995.
11. J.Karsson, P. Liden, P. Dahlgren, R. Johansson and U. Gunneflo, *Using Heavy-ion Radiation to validate fault-handling mechanisms*, IEEE micro, pp8-23,1994.
12. J.R. Horgan and S. London, "ATAC: A data flow coverage testing tool for C", in Proceedings of Symposium on Assessment of Quality Software Development Tools, pp2-10, New Orleans, LA, May, 1992.
13. M.R Lyu, J.R. Horgan and S. London, "A coverage analysis tool for the effectiveness of software testing", in Proceeding of International Symposium on Software Reliability Engineering, 1993.

Automatic Conformance Testing of Internet Applications

Harm M.A. van Beek and Sjouke Mauw

Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands
{harm, sjouke}@win.tue.nl

Abstract. We adapt and extend the theories used in the general framework of automated software testing in such a way that they become suitable for black-box conformance testing of thin client Internet applications. That is, we automatically test whether a running Internet application conforms to its formal specification. The actual implementation of the application is not taken into account, only its externally observable behaviour. In this paper, we show how to formally model this behaviour and how such formal specifications can serve as a basis for the automatic conformance testing of Internet applications.

1 Introduction

Since activity on the Internet is growing very fast, systems that are based on communication via the Internet appear more and more. To give an example, in the United States only, 45 billion dollar of products has been sold via the Internet in 2002 [1]. This is an increase of 38% compared to the on-line sales in 2001. Apart from the number of so-called Internet applications, the complexity of these applications increases too. This increasing complexity leads to a growing amount of errors in Internet applications, of which examples can be found at The Risks Digest [2], amongst others. This increasing number of errors asks for better testing of the applications and, preferably, this testing should be automated.

Research has been done in the field of automated testing of applications that are not based on Internet communication [3]. In this paper, we adapt and extend the theories used in the general framework of automated software testing in such a way that they become suitable for the testing of Internet applications.

We focus on black-box conformance testing of thin client Internet applications. That is, given a running application and a (formal) specification, our goal is to automatically test whether the implementation of the application conforms to the specification. Black box testing means that the actual implementation of the application is not taken into account but only its externally observable behaviour: We test *what* the application does, *not how* it is done. Interaction with the application takes place using the interface that is available to normal users of the application. In this case, the interface is based on communication via the Internet using the HTTP protocol [4].

As a start, in Section 2 the distinction between Web based and Window based applications is drawn. Next, in Section 3 we introduce how we plan to automatically test Internet applications. In Section 4, we describe the formalism we make use of for this automatically testing. To show the usefulness of the framework, we give a practical example in Section 5. We discuss related work in Section 6 and draw some final conclusions in Section 7.

2 Web Based versus Window Based Applications

In general, Web based applications, or Internet applications, behave like window based applications. They both communicate via a user interface with one or more clients. However, there are some major differences.

The Internet applications we focus on, are based on client-server communication via the Internet. The application runs on a server which is connected to the Internet. Via this connection, clients who are also connected to the Internet can interact with the application using prescribed protocols. Clients send requests over the Internet to the server on which the application runs. The server receives the requests and returns calculated responses.

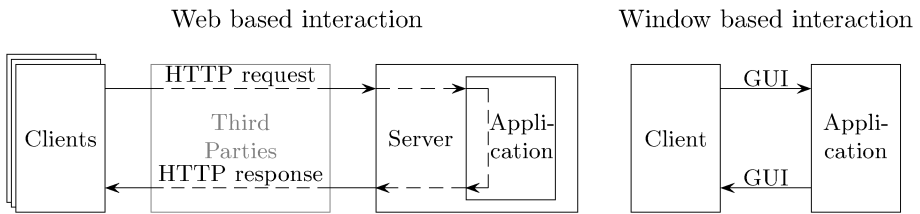


Fig. 1. Internet interaction versus stand-alone interaction.

In Figure 1 a schematic overview of the communication with Internet applications and window based applications is given. Clients interacting with window based applications are using a (graphical) user interface which is directly connected to the application. When interacting with Internet applications, the client sends an HTTP request [4] via the Internet, i.e. via some third parties, to the server. The server receives the request which subsequently is sent to the application. After receiving the request, the application calculates a response which is sent back to the requesting client. As can be seen in Figure 1, when testing an Internet application we have to take into account five entities, viz. clients, communication protocols, third parties, web servers and the application itself.

Clients. The clients we focus on are so-called *thin clients*. This means that they have reduced or no possibility to do calculations. They make use of a centralised resource to operate. In the context of Internet applications, thin clients are usually web browsers. In general, more than one client can simultaneously access an Internet application. Unlike stand-alone applications, clients can fail, i.e. they can “disappear”: a browser can simply be closed without notifying the application.

Dependency on Third Parties. Since interaction takes place via the Internet, communication depends on third parties. First of all, packages transmitted go via routers which control the Internet traffic. It is not known which route on the world wide web is taken to get from the client to the server and back.

Apart from transmitting the requests and responses, there are more dependencies, like DNS servers for translating domain names into IP addresses, trusted third parties for verifying certificates and e-mail servers for both the sending and receiving of e-mail messages.

Stand-alone applications usually do not depend on any of these parties.

Communication via the Internet. Most of the communication with Internet applications we focus on is based on the HyperText Transfer Protocol (HTTP) [4]. This protocol is request-response based. A web server is waiting for requests from clients. As soon as a request comes in, the request is processed by an application running on the server. It produces a response which is sent back. Since the communication takes place via the Internet, delay times are unknown and communication can fail. Therefore, messages can overtake other messages.

Web Servers. A web server is a piece of hardware connected to the Internet. In contrast to stand-alone machines running a stand-alone application, a client might try to access a web server which is down or overtaxed, causing the interaction to fail.

Internet Applications. The Internet application itself is running on a web server. The applications we focus on, are based on request-response interaction with multiple clients. Since more than one client can interact with the application simultaneously, there might be a notion of who is communicating with the application. By keeping track of the interacting parties, requests and corresponding responses can be grouped into so-called *sessions*.

Main differences between Internet based and window based applications are the failing of clients and web servers, the failing of communication and overtaking of messages between clients and the application and the dependency on third parties. Furthermore, Internet applications are request-response based where window based applications interact with the clients using a (graphical) user interface. Finally, most Internet applications focus on parallel communication with more than one client. Since multiple clients can share a common state space, testing Internet applications is basically different from testing window based applications. Window based applications are mostly based on single user interaction. More differences between Web based and Window based systems can be found in e.g. [5].

3 Testing Internet Applications

Now that we have a notion of what Internet applications look like, we informally show how implementations of these applications can be tested.

We focus on black-box testing, restricting ourselves to *dynamic testing*. This means that the testing consists of really executing the implemented system. We do this by simulating real-life interaction with the applications, i.e. by simulating the clients that interact with the application. The simulated clients interact in a similar way as real-life clients would do. In this way, the application cannot distinguish between a real-life

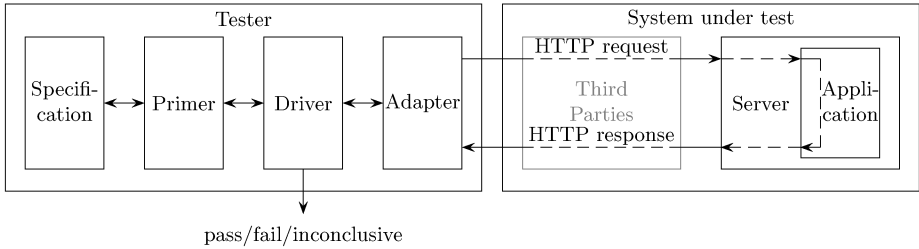


Fig. 2. Automatic testing of Internet applications.

client and a simulated one. See Figure 2 for a schematic overview of the test environment.

We make use of a tester which generates requests and receives responses. This is called *test execution*. By observing the responses, the tester can determine whether they are expected responses in the specification. If so, the implementation passes the test, if not, it fails.

The tester itself consists of four components, based on [6]:

Specification. The specification is the formal description of how the application under test is expected to behave.

Primer. The primer determines the requests to send by inspecting the specification and the current state the test is in. So the primer interacts with the specification and keeps track of the test's state. Furthermore, the primer checks whether responses received by the tester are expected responses in the specification at the state the test is in.

Driver. The driver is the central unit, controlling the execution of the tests. This component determines what actions to execute. Furthermore, the verdict whether the application passes the test is also computed by the driver.

Adapter. The adapter is used for encoding abstract representations of requests into HTTP requests and for decoding HTTP responses into abstract representations of these responses.

While executing a test, the driver determines if a request is sent or a response is checked. If the choice is made to send a request, the driver asks the primer for a correct request, based on the specification. The request is encoded using the adapter and sent to the application under test. If the driver determines to check a response, a response is decoded by the adapter. Next, the primer is asked whether the response is expected in the specification. Depending on the results, a verdict can be given on the conformance of the implementation to its specification.

As mentioned in Section 2, clients, web servers, their mutual communication and third parties can fail. In such a case, no verdict can be given on the correctness of the implementation of the Internet application. However, depending on the failure, it might be possible to determine the failing entity.

4 Conformance Testing of Internet Applications

As a basis for conformance testing of Internet applications, we take the formal framework as introduced in [7–9]. Given a specification, the goal is to check, by means of testing, whether an implemented system satisfies its specification. To be able to formally test applications, there is a need for implementations and formal specifications. Then, *conformance* can be expressed as a relation on these two sets.

Implementations under test are real objects which are treated as black boxes exhibiting behaviour and interacting with their environment. They are not amenable to formal reasoning, which makes it harder to formally specify the conformance relation. Therefore, we make the assumption that any implementation can be modelled by a formal object. This assumption is referred to as the *test hypothesis* [10] and allows us to handle implementations as formal objects. We can express conformance by a formal relation between a model of an implementation and a specification, a so-called *implementation relation*.

An implementation is tested by performing experiments on it and observing its reactions to these experiments. The specification of such an experiment is called a *test case*, a set of test cases a *test suite*. Applying a test to an implementation is called *test execution* and results in a verdict. If the implementation passes or fails the test case, the verdict will be *pass* or *fail*, respectively. If no verdict can be given, the verdict will be *inconclusive*.

In the remainder of this section, we will instantiate the ingredients of the framework as sketched above. We give a formalism for both modelling implementations of Internet applications and for giving formal specifications used for test generation. Furthermore, we give an implementation relation. By doing this, we are able to test whether a (model of an) implementation conforms to its specification. Apart from that, we give an algorithm for generating test suites from specifications of Internet applications.

4.1 Modelling Internet Applications

To be able to formally test Internet applications, we need to formally model their behaviour. Since we focus on conformance testing, we are mainly interested in the communication between the application and its users. We do not focus on the representation of data. Furthermore, we focus on black-box testing, which means that the internal state of applications is not known in the model. Finally, we focus on thin client Internet applications that communicate using the HyperText Transfer Protocol (HTTP) [4]. As a result, the applications show a request/response behaviour.

These observations lead to modelling Internet applications using labelled transition systems. Each transition in the model represents a communication action between the application and a client. The precise model is dictated by the interacting behaviour of the HTTP protocol.

In general, an HTTP interaction is initiated by a client, sending a request for some information to an application. A request can be extended with parameters. These parameters can be used by the application. After calculating a response, it is sent back to the requesting client. Normally, successive requests are not grouped. However, the grouping can be done by adding parameters to the requests and responses. In such a way, alternating sequences of requests and responses are turned into sessions.

Note that we test the interaction behaviour of Internet applications communicating via HTTP. We do *not* model the client-side tools to interact with Internet applications, i.e., we do not model the behaviour of the application when using browser buttons like *stop*, *back*, *forward* and *refresh*. Main reason for not including this behaviour is that different client implementations cause distinct interaction behaviour.

Furthermore, we do not add (failure of) components in the system under test other than the application to the specification. This means that failure of any of these components leads to tests in which the result will be *inconclusive*. If all components in the system under test operate without failure, verdicts will be pass or fail.

The tester should behave like a set of thin clients. The only requests sent to the application are the initial request which models the typing in of a URL in the browser's address bar and requests that result from clicking on links or submitting forms which are contained in preceding responses.

Since we focus on HTTP based Internet applications, and thus on sessions of alternating request-response communication with applications, we make use of so-called *multi request-response transition systems* (MRRTSs) for both modelling implementations of Internet applications and giving formal specifications used for test generation. An MRRTS is a labelled transition system having extra structure. In the remainder of this section we explain MRRTSs in more detail and show how they relate to labelled transition systems and *request-response transition systems* (RRTSs).

Labelled Transition Systems. The formalism of *labelled transition systems* is widely used for describing the behaviour of processes. We will provide the relevant definitions.

Definition 1. A labelled transition system is a 4-tuple $\langle S, L, \rightarrow, s_0 \rangle$ where

- S is a countable, non-empty set of states;
- L is a countable set of labels;
- $\rightarrow \subseteq S \times L \times S$ is the transition relation;
- $s_0 \in S$ is the initial state.

Definition 2. Let s_i ($i \in \mathbb{N}$) be states and a_i ($i \in \mathbb{N}$) be labels. A (finite) composition of transitions

$$s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \dots s_n \xrightarrow{a_n} s_{n+1}$$

is then called a computation. The sequence of actions of a computation, $a_1 \cdot a_2 \cdot \dots \cdot a_n$, is called a trace. The empty trace is denoted by ε . If L is a set of labels, the set of all finite traces over L is denoted by L^* .

Definition 3. Let $p = \langle S, L, \rightarrow, s_0 \rangle$, $s, s' \in S$, $S' \subseteq S$, $a_i \in L$ and $\sigma \in L^*$. Then,

$$\begin{aligned} s \xrightarrow{a_1 \dots a_n} s' &=_{\text{def}} \exists s_1, \dots, s_{n-1} \ s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots s_{n-1} \xrightarrow{a_n} s' \\ s \xrightarrow{a_1 \dots a_n} &=_{\text{def}} \exists s' \ s \xrightarrow{a_1 \dots a_n} s' \\ \text{init}(s) &=_{\text{def}} \{a \in L \mid s \xrightarrow{a}\} \\ \text{traces}(s) &=_{\text{def}} \{\sigma \in L^* \mid s \xrightarrow{\sigma}\} \\ \text{traces}(S') &=_{\text{def}} \bigcup s' \in S' \ \text{traces}(s') \\ s \text{ after } \sigma &=_{\text{def}} \{s' \in S \mid s \xrightarrow{\sigma} s'\} \\ S' \text{ after } \sigma &=_{\text{def}} \bigcup s' \in S' \ s' \text{ after } \sigma \end{aligned}$$

A labelled transition system $p = \langle S, L, \rightarrow, s_0 \rangle$ will be identified by its initial state s_0 . So, e.g., we can write $\text{traces}(p)$ instead of $\text{traces}(s_0)$ and p after σ instead of s_0 after σ .

We aim at modelling the behaviour of the HTTP protocol using labelled transition systems. Therefore, we need to add restrictions on the traces in the labelled transition system used for modelling this behaviour. One of these restrictions is that traces in the LTSs should answer the alternating request/response behaviour.

Definition 4. Let A, B be sets of labels. Then $\text{alt}(A, B)$ is the (infinite) set of traces having alternating structure with respect to elements in A and B , starting with an element in A . Formally, $\text{alt}(A, B)$ is the smallest set such that

$$\varepsilon \in \text{alt}(A, B) \wedge \forall \sigma \in \text{alt}(B, A) \forall a \in A \ a\sigma \in \text{alt}(A, B) .$$

As mentioned before, interactions with an Internet application can be grouped into sessions. To be able to specify the behaviour within each session, we make use of a projection function. This function will be used for determining all interactions contained within one session.

Definition 5. Let σ be a trace and A be a set of labels. Then $\sigma|_A$, the projection of σ to A , is defined by

$$\begin{aligned} \varepsilon|_A &=_{\text{def}} \varepsilon \\ (a \cdot \sigma)|_A &=_{\text{def}} \begin{cases} a \cdot (\sigma|_A) & \text{if } a \in A \\ \sigma|_A & \text{if } a \notin A . \end{cases} \end{aligned}$$

Definition 6. A partitioning S of a set A is a collection of mutually disjoint subsets of A such that their union exactly equals A :

$$\bigcup S = A \quad \wedge \quad \forall B, C \in S \ B \neq C \Rightarrow B \cap C = \emptyset$$

Request-Response Transition Systems. We give a formal definition of a request-response transition system, denoted by RRTS. RRTSs can be compared to input-output transitions systems (IOTSs) [11]. As in IOTSs, we differentiate between two sets of labels, called *request labels* and *response labels*, respectively. RRTSs are based on pure request/response alternation.

Definition 7. Let L be a countable set of labels and $\{L_?, L_!\}$ be a partitioning of L . Then, a request-response transition system $\langle S, L_?, L_!, \rightarrow, s_0 \rangle$ is a labelled transition system $\langle S, L, \rightarrow, s_0 \rangle$ such that

$$\forall \sigma \in \text{traces}(s_0) \quad \sigma \in \text{alt}(L_?, L_!) .$$

Elements in $L_?$ are called request labels, elements in $L_!$ response labels.

RRTSs resemble the notion of Mealy machines, however, it turns out to be technically adhered to start from the notion of RRTSs.

Multi Request-Response Transition Systems. IOTSSs can be used as a basis for multi input-output transition systems (MIOTSSs) [12]. Similarly, in a multi request-response transition system (MRRTS), multiple request-response transition systems are combined into one. All subsystems behave like an RRTS, however interleaving between the subsystems is possible.

Definition 8. Let L be a countable set of labels. Let $\mathbb{L} \subseteq \mathcal{P}(L) \times \mathcal{P}(L)$ be a countable set of tuples such that $\{A, B \mid (A, B) \in \mathbb{L}\}$ is a partitioning of L . Then, a multi request-response transition system $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$ is a labelled transition system $\langle S, L, \rightarrow, s_0 \rangle$ such that

$$\forall (A, B) \in \mathbb{L} \forall \sigma \in \text{traces}(s_0) \quad \sigma|_{A \cup B} \in \text{alt}(A, B) .$$

The set of all possible request labels, $\mathbb{L}_?$, is defined by

$$\mathbb{L}_? =_{\text{def}} \bigcup_{(A, B) \in \mathbb{L}} A .$$

The set of all possible response labels, $\mathbb{L}_!$, is defined by

$$\mathbb{L}_! =_{\text{def}} \bigcup_{(A, B) \in \mathbb{L}} B .$$

Note that an RRTS $\langle S, L_?, L_!, \rightarrow, s_0 \rangle$ can be interpreted as MRRTS $\langle S, \{(L_?, L_!)\}, \rightarrow, s_0 \rangle$, i.e., each MRRTS having singleton $\mathbb{L}$ is an RRTS.

We introduce some extra functions on the sets of tuples as introduced in Definition 8.

Definition 9. Let $\mathbb{L} \subseteq \mathcal{P}(L) \times \mathcal{P}(L)$ be a countable set of tuples, where each tuple contains a set of request labels and a set of response labels. We define functions for determining corresponding requests or responses given either a request label or response label. For $x \in L$, we define functions $\text{req}, \text{resp} : L \rightarrow \mathcal{P}(L)$, such that

$$(\text{req}(x), \text{resp}(x)) \in \mathbb{L} \quad \text{and} \quad x \in \text{req}(x) \cup \text{resp}(x) .$$

4.2 Relating Multi Request-Response Transition Systems

An implementation conforms to a specification if an implementation relation exists between the model of the implementation and its specification. We model both the implementation and the specification as multi request-response transition systems, so conformance can be defined by a relation on MRRTSSs.

While testing Internet applications, we examine the responses sent by the application and check whether they are expected responses by looking at the specification. So we focus on testing whether the implementation does what it is expected to do, not what it is not allowed to do.

Given a specification, we make use of function exp to determine the set of expected responses in a state in the specification.

Definition 10. Let p be a multi request-response transition system $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$. For each state $s \in S$ and for each set of states $S' \subseteq S$, the set of expected responses in s and S' is defined as

$$\begin{aligned} \text{exp}(s) &=_{\text{def}} \text{init}(s) \cap \mathbb{L}_! \\ \text{exp}(S') &=_{\text{def}} \bigcup_{s' \in S'} \text{exp}(s'). \end{aligned}$$

If a model of an implementation i conforms to a specification s , the possible responses in all reachable states in i should be contained in the set of possible responses in the corresponding states in s . Corresponding states are determined by executing corresponding traces in both i and s .

Definition 11. Let MRRTS i be the model of an implementation and MRRTS s be a specification. Then i conforms to s with respect to request-response behaviour, $i \text{ rrconf } s$, if and only if all responses of i are expected responses in s :

$$i \text{ rrconf } s \quad =_{\text{def}} \quad \forall \sigma \in \text{traces}(s) \quad \text{exp}(i \text{ after } \sigma) \subseteq \text{exp}(s \text{ after } \sigma).$$

Relation **rrconf** on MRRTSs is analogous to relation **conf** on LTSs as formalised in [13].

4.3 Test Derivation

An implementation is tested by performing experiments on it and observing its reactions to these experiments. The specification of such an experiment is called a *test case*. Applying a test to an implementation is called *test execution*. By now we have all elements for deriving such test cases.

Since the specification is modelled by an MRRTS, a test case consists of request and response actions as well. However, we have some more restrictions on test cases. First of all, test cases should have finite behaviour to guarantee that tests terminate. Apart from that, unnecessary nondeterminism should be avoided, i.e., within one test case the choice between multiple requests or between requests and responses should be left out.

In this way, a test case is a labelled transitions system where each state is either a terminating state, a state in which a request is sent to the implementation under test, or a state in which a response is received from the implementation. The terminating states are labelled with a verdict which is a **pass** or **fail**.

Definition 12. A test case t is an LTS $\langle S, \mathbb{L}_? \cup \mathbb{L}_!, \rightarrow, s_0 \rangle$ such that

- t is deterministic and has finite behaviour;
- S contains terminal states **pass** and **fail** with $\text{init}(\text{pass}) = \text{init}(\text{fail}) = \emptyset$;
- for all $s \in S \setminus \{\text{pass}, \text{fail}\}$, $\text{init}(s) = \{a\}$ for $a \in \mathbb{L}_?$ or $\text{init}(s) = \mathbb{L}_!$.

We denote this subset of LTSs by TESTS. A set of test cases $T \subseteq \text{TESTS}$ is called a test suite.

We do not include the possibility for reaching *inconclusive* states in test cases. Such verdicts are given if a component in the system under test, other than the application, fails. The tester (as described in Section 3) is able to identify errors caused by the application and lead to a **fail** state. Other errors result in an inconclusive verdict.

As mentioned, we call a set of test cases a *test suite*. Such a test suite is used for determining whether an implementation conforms to a specification. A test suite T is said to be *sound* if and only if all implementations that conform to the specification pass all test cases in T . If all implementations that do not conform to the specification fail a test case in T , T is called *exhaustive*. Test suite that are both sound and exhaustive are said to be *complete* [9].

Definition 13. Let MRRTS i be an implementation and T be a test suite. Then, implementation i passes test suite T if no traces in i lead to a fail state:

$$i \text{ passes } T \quad =_{\text{def}} \quad \neg \exists t \in T \exists \sigma \in \text{traces}(i) \quad \sigma \cdot \mathbf{fail} \in \text{traces}(t)$$

We use the notation $\sigma \cdot \mathbf{fail}$ to represent trace σ leading to a fail state, i.e., $\sigma \cdot \mathbf{fail} \in \text{traces}(t) =_{\text{def}} t \xrightarrow{\sigma} \mathbf{fail}$.

Definition 14. Let s be a specification and T be a test suite. Then for relation **rrconf**:

$$\begin{aligned} T \text{ is sound} &=_{\text{def}} \quad \forall i \quad i \text{ rrconf } s \implies i \text{ passes } T \\ T \text{ is exhaustive} &=_{\text{def}} \quad \forall i \quad i \text{ rrconf } s \longleftarrow i \text{ passes } T \\ T \text{ is complete} &=_{\text{def}} \quad \forall i \quad i \text{ rrconf } s \iff i \text{ passes } T \end{aligned}$$

In practice, however, such a complete test suite will often be infinitely large, and therefore not suitable. So, we have to restrict ourselves to test suites for detecting non-conformance instead of test suites for giving a verdict on the conformance of the implementation. Such test suites are called *sound*.

To test conformance with respect to request-response behaviour, we have to check for all possible traces in the specification that the responses generated by the implementation are expected responses in the specification. This can be done by having the implementation execute traces from the specification. The responses of the implementation are observed and compared with the responses expected in the specification. Expected responses pass the test, unexpected responses fail the test. The algorithm given is based on the algorithm for generating test suites as defined in [14].

Algorithm 1. Let s be MRRTS $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$. Let C be a non-empty set containing all possible states of the specification in which the implementation can be at the current stage of the test. Initially $C = \{s_0\}$. We then define the collection of nondeterministic recursive algorithms gentest^n ($n \in \mathbb{N}$) for deriving test cases as follows:

$$\begin{aligned} \text{gentest}^n &: \mathcal{P}(S) \rightarrow \text{TESTS} \\ \text{gentest}^n(C) &=_{\text{def}} \left[\begin{array}{l} \text{return } \mathbf{pass} \\ \parallel n > 0 \wedge a \in \mathbb{L}_? \wedge C \text{ after } a \neq \emptyset \quad \rightarrow \\ \quad \text{return } a \cdot \text{gentest}^{n-1}(C \text{ after } a) \\ \parallel n > 0 \quad \rightarrow \\ \quad \text{return } \sum \{b \cdot \mathbf{fail} \mid b \in \mathbb{L}_! \setminus \text{exp}(C)\} \\ \quad + \sum \{b \cdot \text{gentest}^{n-1}(C \text{ after } b) \mid b \in \text{exp}(C)\} \\ \end{array} \right] \end{aligned}$$

The $\cdot$ infix notation is used for sequential composition. So, e.g., $a \cdot b$ relates to transitions $s \xrightarrow{a} s' \xrightarrow{b} s''$. As mentioned, notation $a \cdot \mathbf{pass}$ and $a \cdot \mathbf{fail}$ is used for representing transitions $s \xrightarrow{a} \mathbf{pass}$ and $s \xrightarrow{a} \mathbf{fail}$, respectively. We use Σ -notation to indicate that it is not known which of the responses is returned by the implementation. So, e.g. $a + b$ relates to transitions $s \xrightarrow{a} s'$ and $s \xrightarrow{b} s''$. Depending on whether the response is expected, the algorithm might either continue or terminate in a **fail** state.

Although a choice for the first option can be made in each step, we added a parameter to the algorithm, $n \in \mathbb{N}$, to force termination. As mentioned, we want all test cases to be finite, since otherwise no verdict might take place.

The set of derivable test cases from $\text{gentest}^n(C)$ is denoted by $\overline{\text{gentest}^n(C)}$. So $\text{gentest}^n(C)$ is the set of all possible test cases of at most n transitions starting in states C of the specification. Although our goal is to generate sound test suites, we will prove that in the limit, as n reaches infinity, test suite $\bigcup_{n>0} \overline{\text{gentest}^n(\{s_0\})}$ is complete for specification $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$. To prove this, we make use of some lemmas.

Lemma 1. *Let s be a specification $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$ and $\sigma_0, \sigma_1 \in L^*$, $\sigma_1 \neq \varepsilon$. Then*

$$\sigma_0 \sigma_1 \in \text{traces}(\overline{\text{gentest}^n(\{s_0\})}) \iff \sigma_1 \in \text{traces}(\overline{\text{gentest}^{n-|\sigma_0|}(s_0 \text{ after } \sigma_0)})$$

where $|\sigma|$ is the length of trace σ .

Sketch of proof. This lemma can be proved by using induction on the structure of σ_0 .

Lemma 2. *Let s be a specification $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$, $\sigma_0 \in L^*$ and $n > 0$. Then*

$$\sigma \cdot \mathbf{fail} \in \text{traces}(\overline{\text{gentest}^n(\{s_0\})}) \implies \exists \sigma' \in L^* \exists b \in \mathbb{L}_! \sigma = \sigma' b.$$

Proof. This can be easily seen by looking at the definition of the gentest algorithm: State **fail** can only be reached after execution of a $b \in \mathbb{L}_!$. □

Theorem 1. *Let s be a specification $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$.*

Then test suite $\bigcup_{n>0} \overline{\text{gentest}^n(\{s_0\})}$ is complete.

Proof. Let s be $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$ and T be $\bigcup_{n>0} \overline{\text{gentest}^n(\{s_0\})}$. Then,

$$\begin{aligned} & T \text{ is complete} \\ \equiv & \{ \text{definition of complete test suites} \} \\ & \forall i \quad i \text{ rrconf } s \Leftrightarrow i \text{ passes } T \\ \equiv & \{ \text{definition of rrconf and passes} \} \\ & \forall i \quad \forall \sigma \in \text{traces}(s) \exp(i \text{ after } \sigma) \subseteq \exp(s \text{ after } \sigma) \\ & \iff \\ & \neg \exists t \in T \exists \sigma \in \text{traces}(i) \sigma \cdot \mathbf{fail} \in \text{traces}(t) \end{aligned}$$

We prove this by proving exhaustiveness ($\Leftarrow$) and soundness ($\Rightarrow$) separately.

– Exhaustiveness.

$$\begin{aligned} & \forall i \quad \forall \sigma \in \text{traces}(s) \exp(i \text{ after } \sigma) \subseteq \exp(s \text{ after } \sigma) \\ & \iff \\ & \neg \exists t \in T \exists \sigma \in \text{traces}(i) \sigma \cdot \mathbf{fail} \in \text{traces}(t) \end{aligned}$$

We prove exhaustiveness by contradiction:

Let $\sigma \in \text{traces}(s)$ and $b \in \exp(i \text{ after } \sigma)$ such that $b \notin \exp(s \text{ after } \sigma)$. Then, we prove that $\exists t \in T \exists \sigma' \in \text{traces}(i) \sigma' \cdot \text{fail} \in \text{traces}(t)$.

$$\begin{aligned}
& \exists t \in T \exists \sigma' \in \text{traces}(i) \sigma' \cdot \text{fail} \in \text{traces}(t) \\
\Leftarrow & \{ b \in \exp(i \text{ after } \sigma) \Rightarrow \sigma b \in \text{traces}(i), \text{ Let } \sigma' = \sigma \cdot b \} \\
& \exists t \in T \sigma \cdot b \cdot \text{fail} \in \text{traces}(t) \\
\Leftarrow & \{ \text{Definition of } T \} \\
& \exists n > 0 \sigma \cdot b \cdot \text{fail} \in \text{traces}(\overline{\text{gentest}^n(\{s_0\})}) \\
\equiv & \{ \text{Lemma 1} \} \\
& \exists n > 0 b \cdot \text{fail} \in \text{traces}(\overline{\text{gentest}^{n-|\sigma|}(\{s_0 \text{ after } \sigma\})}) \\
\Leftarrow & \left\{ \begin{array}{l} \text{gentest (third option), let } n > |\sigma|, \\ b \notin \exp(s_0 \text{ after } \sigma) \Rightarrow b \in \mathbb{L}_! \setminus \exp(s_0 \text{ after } \sigma) \end{array} \right\} \\
& \text{true}
\end{aligned}$$

– Soundness.

$$\begin{aligned}
& \forall i \quad \forall \sigma \in \text{traces}(s) \exp(i \text{ after } \sigma) \subseteq \exp(s \text{ after } \sigma) \\
& \implies \\
& \neg \exists t \in T \exists \sigma \in \text{traces}(i) \sigma \cdot \text{fail} \in \text{traces}(t)
\end{aligned}$$

Soundness is also proved by contradiction:

Let $t \in T$ and $\sigma \in \text{traces}(i)$ such that $\sigma \cdot \text{fail} \in \text{traces}(t)$. Then, by definition of T , $\exists n > 0 \sigma \cdot \text{fail} \in \text{traces}(\overline{\text{gentest}^n(\{s_0\})})$. Let $m > 0$ such that $\sigma \cdot \text{fail} \in \text{traces}(\overline{\text{gentest}^m(\{s_0\})})$. We prove that $\exists \sigma' \in \text{traces}(s) \exists b \in \exp(i \text{ after } \sigma') b \notin \exp(s \text{ after } \sigma')$.

Let $\sigma'' \in \text{traces}(s)$ and $b'' \in \exp(i \text{ after } \sigma'')$. Then, we prove that $b'' \notin \exp(s \text{ after } \sigma'')$. Since $\sigma \cdot \text{fail} \in \text{traces}(\overline{\text{gentest}^m(\{s_0\})})$, using Lemma 2, $\exists \sigma' \in \text{traces}(s) \exists b \in \mathbb{L}_! \sigma = \sigma' \cdot b$. Let $\sigma = \sigma'' \cdot b''$. Then,

$$\begin{aligned}
& \sigma \cdot \text{fail} \in \text{traces}(\overline{\text{gentest}^m(\{s_0\})}) \\
\equiv & \{ \sigma = \sigma'' \cdot b'' \} \\
& \sigma'' \cdot b'' \cdot \text{fail} \in \text{traces}(\overline{\text{gentest}^m(\{s_0\})}) \\
\equiv & \{ \text{Lemma 1} \} \\
& b'' \cdot \text{fail} \in \text{traces}(\overline{\text{gentest}^{m-|\sigma''|}(s_0 \text{ after } \sigma'')}) \\
\Rightarrow & \{ \text{Definition of algorithm gentest (third option)} \} \\
& b'' \in \mathbb{L}_! \setminus \exp(s_0 \text{ after } \sigma'') \\
\Rightarrow & \{ \text{Set theory} \} \\
& b'' \notin \exp(s_0 \text{ after } \sigma'')
\end{aligned}$$

□

Algorithm 1 can easily be optimised. As can be seen by looking at the algorithm, each choice for inspecting a response of the implementation leads to $|\mathbb{L}_!|$ new branches in the generated test case. However, many of these branches will never take place as a result of the alternating request-response behaviour of the implementation: the implementation can only send responses on requests sent by the tester. It can be proved that by adding only this restricted set of responses to the test cases, such optimised generated test suites are still complete.

5 Example: Internet Vote

We show how the theory introduced in former sections can be used for testing real-life Internet applications. As an example, we take a voting protocol. All members of a group of voters are asked whether they are for or against a proposition. They are able to visit a web site where they can either vote or have a look at the current score. They can vote at most once and they can check the score as often as they want to.

We start by giving an MRRTS that formally specifies the application. Let V be the set of voters and $\mathbb{P} = \{\text{for}, \text{against}\}$. Then,

- $\mathbb{L}$, the set of tuples of transition labels is defined as

$$\mathbb{L} = \{ (\{ \text{vote}(v, p)_{?s} \mid p \in \mathbb{P} \}, \{ \text{ok}_{!s}, \neg \text{ok}_{!s} \}) \mid v \in V, s \in \mathbb{N} \} \\ \cup \{ (\{ \text{score}_{?s} \}, \{ \text{score}(f, a)_{!s} \}) \mid f, a, s \in \mathbb{N} \} .$$

The first part specifies the interactions where voter v sends a request to vote for or against the proposition (p). The response is a confirmation (ok) or a denial ($\neg \text{ok}$), depending on whether the voter had voted before. The second part specifies the requests for the score which are responded by the number of votes for (f) and against (a) the proposition. All labels are extended with an identifier (s) for uniquely identifying the sessions.

- The set of states S is defined as $S = \mathcal{P}(\mathbb{L}_?) \times \mathcal{P}(V) \times \mathcal{P}(V) \times \mathcal{P}(\mathbb{N} \times \mathbb{N} \times \mathbb{N})$.

For $\langle R, F, A, C \rangle \in S$,

- $R \subseteq \mathbb{L}_?$ is the set of requests on which no response has been sent yet;
- $F \subseteq V$ is the set of voters who voted for the proposition;
- $A \subseteq V$ is the set of voters who voted against the proposition;
- $C \subseteq \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ is the score at the moment that a request for the score is sent.

We need to keep track of this score for determining the possible results that can be responded: The scores returned should be at least the scores at the time of the sending of the request.

For $\langle s, f, a, \rangle \in C$,

- * $s \in \mathbb{N}$ is the session identifier;
- * $f \in \mathbb{N}$ is the number of voters who voted for the proposition;
- * $a \in \mathbb{N}$ is the number of voters who voted against the proposition.

- Let $s \in \mathbb{N}$, $v \in V$ and $p \in \mathbb{P}$. Then, transition relation $\rightarrow$ is defined by the following derivation rules.

If no session exists with identifier s , a session can be started by sending a request to vote for or against the proposition or by sending a request for the current score:

$$\frac{\text{score}_{?s} \notin R, \neg \exists_{w \in V} \exists_{q \in \mathbb{P}} \text{vote}(w, q)_{?s} \in R}{\langle R, F, A, C \rangle \xrightarrow{\text{vote}(v, p)_{?s}} \langle R \cup \{ \text{vote}(v, p)_{?s} \}, F, A, C \rangle}$$

$$\frac{\text{score}_{?s} \notin R, \neg \exists_{w \in V} \exists_{q \in \mathbb{P}} \text{vote}(w, q)_{?s} \in R}{\langle R, F, A, C \rangle \xrightarrow{\text{score}_{?s}} \langle R \cup \{ \text{score}_{?s} \}, F, A, C \cup \{ \langle s, |F|, |A| \rangle \} \rangle}$$

If a request to vote for or against the proposition has been sent and the voter has not voted before, the vote can be processed and confirmed:

$$\frac{\text{vote}(v, \text{for})_{?s} \in R, v \notin F \cup A}{\langle R, F, A, C \rangle \xrightarrow{\text{ok}_{!s}} \langle R \setminus \{\text{vote}(v, \text{for})_{?s}\}, F \cup \{v\}, A, C \rangle}$$

$$\frac{\text{vote}(v, \text{against})_{?s} \in R, v \notin F \cup A}{\langle R, F, A, C \rangle \xrightarrow{\text{ok}_{!s}} \langle R \setminus \{\text{vote}(v, \text{against})_{?s}\}, F, A \cup \{v\}, C \rangle}$$

If a request to vote has been sent and the voter has already voted before or the voter is concurrently sending a request to vote in another session, the vote can be rejected:

$$\frac{\text{vote}(v, p)_{?s} \in R, \neg \exists_{t \in \mathbb{N} \setminus \{s\}} \exists_{q \in \mathbb{P}} \text{vote}(v, q)_{?t} \in R \vee v \in F \cup A}{\langle R, F, A, C \rangle \xrightarrow{\neg \text{ok}_{!s}} \langle R \setminus \{\text{vote}(v, p)_{?s}\}, F, A, C \rangle}$$

If a request for the score has been sent, the scores can be sent to the requesting client. Since interactions can overtake each other, the result can be any of the scores between the sending of the request and the receiving of the response. So, the score must be at least the score at the moment of requesting the score and at most the number of processed votes plus the number of correct votes, sent in between requesting for the score and receiving the score:

$$\frac{\begin{array}{l} \text{score}_{?s} \in R, \langle s, f, a \rangle \in C, \\ f' \leq |F| + (\#_{v \in V \exists t \in \mathbb{N}} \text{vote}(v, \text{for})_{?t} \in R \wedge v \notin F \cup A), \\ a \leq a' \leq |A| + (\#_{v \in V \exists t \in \mathbb{N}} \text{vote}(v, \text{against})_{?t} \in R \wedge v \notin F \cup A) \end{array}}{\langle R, F, A, C \rangle \xrightarrow{\text{score}(f', a')_{!s}} \langle R \setminus \{\text{score}_{?s}\}, F, A, C \setminus \{\langle s, f, a \rangle\} \rangle}$$

- Initial state $s_0 = \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$: no requests have been sent yet, no one has voted for and no one has voted against the proposition.

Labelled transition systems suit nicely for giving a theoretical framework for automatic conformance testing. However, as expected, using LTSs for giving specifications of Internet applications is not convenient. To make this framework useful in practice, we need a formalism for easier specifying these applications. Therefore, we are currently developing *DiCons* [15, 16], which is a formal specification language dedicated to the domain of Internet applications. We will not give the actual *DiCons* specification here since this goes beyond the scope of this paper. However, to give an example, the Internet vote described above can be specified in *DiCons* in five lines of code.

As a proof of concept, we implemented an on-the-fly version of Algorithm 1. We used this algorithm to test eleven implementations of the Internet vote application: one correct and ten incorrect implementations. We tested by executing 26.000 test cases per implementation. This took approximately half a day per implementation. We tested

Table 1. Test results

implementation	% failures	verdict
1. correct implementation	0.00	pass
2. no synchr.: first calculate results, then remove voter	33.30	fail
3. no synchr.: first remove voter, then calculate results	32.12	fail
4. votes are incorrectly initialised	91.09	fail
5. votes for and against are mixed up	87.45	fail
6. votes by voter 0 are not counted	32.94	fail
7. voter 0 cannot vote	91.81	fail
8. unknown voter can vote	0.00	pass
9. voters can vote more than once	68.75	fail
10. voter 0 is allowed to vote twice	16.07	fail
11. last vote is counted twice	8.82	fail

using different lengths of test traces and different numbers of voters. The test results are summarised in Table 1. The left column describes the error in the implementation. In the second column, the percentage of test cases that ended in a fail state is given.

As can be seen, in nine out of ten incorrect implementations, errors are detected. In all test cases, only requests are sent that are part of the specification, i.e., only requests for votes by known voters are sent. Because we did not specify that unknown voters are forbidden to vote, errors in the implementation that allow other persons to vote are not detected: the implementation conforms to the specification.

The percentages in Table 1 strongly depend on the numbers of voters and lengths of the test traces. Some errors can easily be detected by examining the scores, e.g. incorrect initialisation. This error can be detected by traces of length 2: request for the score and inspect the corresponding response. Other errors, however, depend on the number of voters. If the last vote is counted twice, all voters have to vote first, after which the scores have to be inspected. This error can only be detected by executing test traces with at least a length of two times the number of voters plus two.

6 Related Work

Automatic test derivation and execution based on a formal model has been an active topic of research for more than a decade. This research led to the development of a number of general purpose black box test engines. However, the domain of Internet applications implies some extra structure on the interacting behaviour of the implementation which enforces the adaptation of some of the key definitions involved. Therefore, our work can be seen as an extension to and adaptation of the formal testing framework as introduced in [7–9]. The major difference stems from our choice to model an Internet application as a multi request-response transition system. We expect that existing tools (such as TorX [6]) can be easily adapted to this new setting. The reader may want to consult [3] for an overview of other formal approaches and testing techniques.

Approaching the problem of testing Internet applications from another angle, one encounters methodologies and tools based on capture/replay (see e.g. [17, 18]). In the case of capture/replay testing, test cases are produced manually and recorded once, after which they can be applied to (various) implementations. These tools prove very beneficial for instance for regression testing. However, automatic generation of test cases

has several advantages. In general it proves to be a more flexible approach, yielding test suites that are better maintainable and more complete and test suites can be generated quicker (and thus cheaper). The main disadvantage of automatic black box testing is that it requires a formal model of the implementation under test.

A methodology that comes very close to ours is developed by Ricca and Tonella [19]. The starting point of their semi-automatic test strategy is a UML specification of a web application. This specification is manually crafted, possibly supported by re-engineering tools that help in modelling existing applications. Phrased in our terms, Ricca and Tonella consider RRTSs as their input format (which they call *path expressions*). We perform black-box testing, whereas they consider white-box testing. This implies that their approach considers implementation details (such as cookies), while we only look at the observable behaviour. White-box testing implies a focus on test criteria, instead of a complete testing algorithm. Finally, we mention the difference in user involvement. In our approach the user has two tasks, viz. building an abstract specification and instantiating the test adapter which relates abstract test events to concrete HTTP-events. In their approach the user makes a UML model, produces tests and interprets the output of the implementation. For all of this, appropriate tool support is developed, but the process is not automatic. In this way derivation and execution of a test suite consisting of a few dozens of tests takes a full day, whereas our on-the-fly approach supports many thousands of test cases being generated, executed and interpreted in less time.

Jia and Liu[20] propose a testing methodology which resembles Ricca and Tonella's in many respects, so the differences with our work are roughly the same. Their focus is on the specification of test cases (by hand), while our approach consists of the generation of test cases from a specification of the intended application's behaviour. Their approach does not support on-the-fly test generation and execution. Like Ricca and Tonella, their model is equivalent to RRTSs which makes it impossible to test parallel sessions (or users) that share data.

Wu and Offutt [21] introduce a model for describing the behaviour of Web applications, which can be compared to the *DiCons* language. In contrast to the model presented in this paper, their model supports the usage of special buttons that are available in most Web browsers. The main difference with our model is that they focus on stateless applications, i.e., responses only depend on the preceding request. We model stateful applications which are based on parallelly executed sessions.

Another functional testing methodology is presented by Niese, Margaria and Steffen in [22]. Where we focus on modelling Internet applications only, they model other subsystems in the system under test as well. In their approach, test cases are not generated automatically, but designed by hand using dedicated tools. Test execution takes place automatically via a set of cooperating subsystem-specific test tools, controlled by a so-called test coordinator.

Our research focuses on conformance testing only. Many other properties are important for the correct functioning of web applications, such as performance, user interaction and link correctness [23]. Testing such properties is essentially different from conformance testing. They focus on *how well* applications behave instead of *what* they do. Plenty of tools are available for performance testing, e.g., [24, 25].

7 Conclusion

The research reported on in this paper is conducted in the context of the *DiCons* project (see [15, 16]). The goal of this project is the application of formal methods (especially process algebra) to the application of dependable Internet applications. One of the results of this project is the development of the *DiCons* language, which is targeted to the specification of the interaction behaviour of Internet applications. The *DiCons* compiler allows for the generation of stand-alone Internet applications.

Due to the focus of *DiCons* on interaction, rather than on presentation, it is likely that developers will prefer to use a less formal approach that supports the need for a nice user interface. However, our current research shows that development of a formal interaction model, like in *DiCons*, still has benefits. Our research shows that there is a point in making a formal model, even if it is not used to generate Internet applications, since a formal model can be used for (automated) conformance testing of the application.

The input of the testing process described in this paper is a multi request-response transition system which is a theoretically simple model, but which is very hard to use in practice for the specification of real applications. Since *DiCons* is targeted to specify Internet applications and since its operational semantics is an MRRTS, we plan to connect the *DiCons* execution engine to our prototype testing tool.

As the development of a formal model of an Internet application is quite an investment, we expect that only in cases where it is vital that the application shows the correct interaction behaviour automated formal testing will be applied. However, there will be a huge gain in reliability and maintainability of the application (e.g. because of automated regression testing), compared with e.g. capture and replay techniques.

Although we have only built a simple prototype, we can already conclude that the proposed testing approach works in practice, since it quickly revealed (planted) errors in erroneous implementations. Interestingly enough, playing with the prototype made it clear that the response times in the HTTP-protocol are much slower than in traditional window based applications, resulting in less test runs per time unit. We cannot foresee if the unreliability of an Internet connection will prevent us from executing lengthy test runs over the Internet.

An interesting point is that the actual HTTP-response of an Internet application has to be matched against the expected abstract event from the specification. In our current prototype tool we simply scan for the occurrence of certain strings, but this does not seem to be a safe and generic approach. Future research should answer the question of how to match actual HTTP-replies against abstract events.

References

1. Regan, K.: U.S.: E-Commerce Topped \$45B in 2002. E-Commerce Times (2003)
2. ACM Committee on Computers and Public Policy, P.G. Neumann, moderator: The Risk Digest, Forum On Risks To The Public In Computers And Related Systems.
[http://catless.ncl.ac.uk/Risks/\(1985-2003\)](http://catless.ncl.ac.uk/Risks/(1985-2003))
3. Brinksma, E., Tretmans, J.: Testing Transition Systems: An Annotated Bibliography. In Cassez, F., Jard, C., Rozoy, B., Ryan, M., eds.: Summer School MOVEP'2k – Modelling and Verification of Parallel Processes, Nantes (2000) 44–50
4. Fielding, R., Gettys, J., Mogul, J.C., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext transfer protocol – HTTP/1.1. RFC 2616, The Internet Society, Network Working Group (1999)

5. Rice, J., Farquhar, A., Piernot, P., Gruber, T.: Using the web instead of a window system. In: Human Factors in Computing Systems, CHI'96 Conference Proceedings, Vancouver, B.C, Canada (1996) 103–110
6. Belinfante, A., Feenstra, J., de Vries, R., Tretmans, J., Goga, N., Feijs, L., Mauw, S., Heerink, L.: Formal test automation: A simple experiment. In: 12th Int. Workshop on Testing of Communicating Systems, Kluwer Academic Publishers (1999)
7. Brinksma, E., Alderden, R., Langerak, J., van de Lagemaat, R., Tretmans, J.: A formal approach to conformance testing. In: Second International Workshop on Protocol Test Systems, North-Holland (1990) 349–363
8. Tretmans, J.: A formal approach to conformance testing. In Rafiq, O., ed.: International Workshop on Protocol Test Systems VI. Volume C-19 of IFIP Transactions., North-Holland (1994) 257–276
9. ISO/IEC JTC1/SC21 WG7, ITU-T SG 10/Q.8: Proposed ITU-T Z.500 and committee draft on “formal methods in conformance testing”. CD 13245-1, ISO – ITU-T, Geneva (1996)
10. Bernot, G.: Testing against formal specifications: A theoretical view. In Abramsky, S., Maibaum, T.S.E., eds.: TAPSOFT '91: Proceedings of the International Joint Conference on Theory and Practice of Software Development. Volume 494 of Lecture Notes in Computer Science., Springer-Verlag (1991) 99–119
11. Tretmans, J.: Testing labelled transition systems with inputs and outputs. In Cavalli, A., Budkowski, S., eds.: Participants Proceedings of the Int. Workshop on Protocol Test Systems VIII – COST 247 Session, Evry, France (1995) 461–476
12. Heerink, L.: Ins and outs in refusal testing. PhD thesis, University of Twente, The Netherlands (1998)
13. Brinksma, E., Scollo, G., Steenbergen, C.: LOTOS specifications, their implementations and their tests. Protocol Specification, Testing and Verification VI, IFIP 1987 (1987) 349–360
14. Tretmans, J.: Test Generation with Inputs, Outputs and Repetitive Quiescence. Software—Concepts and Tools **17** (1996) 103–120
15. van Beek, H.: Internet protocols for distributed consensus – the DiCons language. Master's thesis, Technische Universiteit Eindhoven (2000)
16. Baeten, J., van Beek, H., Mauw, S.: Specifying internet applications with DiCons. In: Proc. 16th ACM Symposium on Applied Computing, Las Vegas, USA (2001)
17. CollabNet, Inc.: MaxQ. <http://maxq.tigris.org/> (1999–2003)
18. The Original Software Group Ltd.: TestWEB. <http://www.testweb.com/> (2003)
19. Ricca, F., Tonella, P.: Analysis and testing of web applications. In: Proceedings of the 23rd International Conference on Software Engineering (ICSE-01), Toronto, Ontario, Canada, IEEE Computer Society (2001) 25–34
20. Jia, X., Liu, H.: Rigorous and automatic testing of web applications. In: Proceedings of the 6th IASTED International Conference on Software Engineering and Applications (SEA 2002), Cambridge, MA, USA (2002) 280–285
21. Wu, Y., Offutt, J.: Modeling and testing web-based applications. ISE Technical ISE-TR-02-08, GMU (2002)
22. Niese, O., Margaria, T., Steffen, B.: Automated functional testing of web-based applications. In: Proceedings of the 5th Int. Conference On Software and Internet Quality Week Europe (QWE2002), Brussels, Belgium (2002)
23. Benedikt, M., Freire, J., Godefroid, P.: VeriWeb: Automatically testing dynamic web sites. In: Proceedings of the 11th international world wide web conference (WWW2002), Honolulu, Hawaii, USA (2002)
24. Dieselpoint, Inc.: dieselpoint. <http://www.dieselpoint.com/> (2001)
25. Fulmer, J.: Siege. <http://www.joedog.org/siege/> (2002)

A Use Case Driven Testing Process: Towards a Formal Approach Based on UML Collaboration Diagrams

Mourad Badri, Linda Badri, and Marius Naha

Département de Mathématiques et Informatique
Université du Québec à Trois-Rivières
C.P. 500, Trois-Rivières, Québec, Canada G9A 5H7
{Mourad_Badri, Linda_Badri, Marius_Naha}@uqtr.ca

Abstract. Dynamic interactions between a group of objects, for the realization of a use case or a complex operation of an object, may be specified by using UML collaboration diagrams. Collaboration defines the roles a group of objects play when performing a particular task and several aspects of the control related to their interactions. The specification described in a collaboration diagram must be preserved during the transformation process into an implementation. Test generation based upon collaboration diagrams is actually a poorly exploited approach. The testing methodology (generation and verification) proposed in this paper is based on the dynamic interactions between objects and takes into account several aspects related to their control. It supports an incremental verification of the implementation of the use cases. The generated sequences correspond to the different scenarios of the use cases, expected during the analysis and design phases. They allow verifying whether the implementation of each use case is in accordance with the corresponding specification. We also present in this paper a brief summary of the environment that we developed for supporting the proposed use case driven testing process.

1 Introduction

Object technology has been widely used during the last decade, especially in the industrial areas. Actual industrial systems require the development of increasingly complex software. Reliability is amongst their most important quality characteristics. Undetected defects may result in important consequences not only as to their quality, but also as to their development and maintenance costs. Software testing is recognized as a vital part of their development process. It represents an important action in their quality assurance [14]. Object-Oriented Testing addresses important questions that are not covered by the standard procedural testing approaches. These questions concern those aspects of software testing that arise specifically in relation to the concepts introduced by the object paradigm. An extensive body of literature exists to address the issues associated with testing object-oriented systems. Object-oriented systems can be tested at different levels. The basic unit for object-oriented systems is the class. Many authors have addressed several aspects of the unit testing of classes by considering different approaches such as white-box, black-box and state-based testing techniques. The large number of works related to object-oriented testing, as stated for

instance in [5, 11, 12] amongst several publications, have allowed giving many interesting responses taking into account the object paradigm concepts. However, relative little work has been conducted at the integration and system levels.

In object-oriented software, objects interact in order to implement the behavior. The dynamic interactions between a group of objects, for the realization of a system functionality (use case) or a complex operation of an object, may be specified by using UML collaboration diagrams. Collaboration defines the roles a group of objects play when performing a particular task and several aspects of the control related to their interactions. The specification described in a collaboration diagram must be preserved during the transformation process into an implementation. Consequently, just as we can envisage refining the specification into an implementation, we can also envisage using this specification for generating test sequences. Test generation based upon a specification described into a collaboration diagram represents an interesting way, which is actually poorly exploited as stated by A. Abdurazik and J. Offutt in [1]. These authors have also discussed the several advantages that such approach presents.

The testing methodology (generation and verification) proposed in this paper is a new approach. It is based on the dynamic interactions between objects and takes into account several aspects related to their control. It supports an incremental verification of the implementation of the use cases. The generated sequences correspond to the different scenarios of the use cases, expected during the analysis and design phases. They allow verifying whether the implementation of each use case is in accordance with the corresponding specification. Use cases have been universally adopted for requirements specification as stated in [8]. Use cases start in requirements, are translated into collaborations in analysis and design phases, and support test cases design in test.

The rest of the paper is organized as follows: section 2 presents the methodology of our approach and the main phases of the proposed testing process. Section 3 presents UML collaboration diagrams, some extensions that we propose to better specify the interactions between objects and the formal description of collaborations retained in the framework of our process. The main steps of the test sequences generation technique are presented in section 4. Section 5 gives a summary of the supported verification process. Section 6 presents the architecture of the environment that we developed for supporting our approach and illustrates it on a sample example. Section 7 gives some conclusions and perspectives of the present work.

2 Testing Process: Approach Methodology

The methodology of the proposed object-oriented testing process is illustrated in Figure 1. The testing process is divided in two main phases. The test sequences generation technique, first main phase, constitutes a complement of our previous work, which was based on the individual behavior of objects [3]. It represents a refinement and an extension of the technique proposed in [4]. It is based on a formal description of objects groups' behavior including the concept of message post condition related to the interactions between objects. This concept, in particular, gives a solid basis for the verification process, second main phase of the proposed process. Each generated sequence corresponds to a particular scenario of the considered use case.

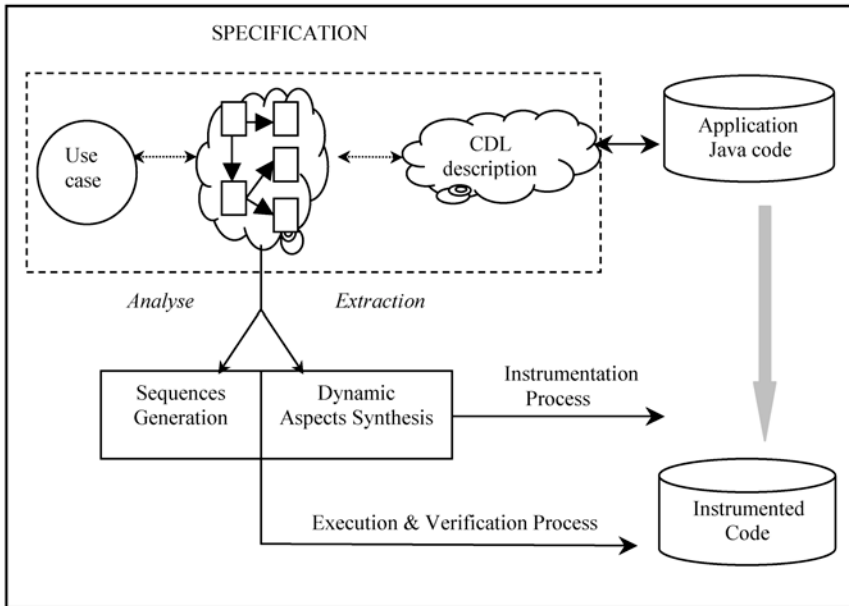


Fig. 1. Testing Process Methodology

The generated sequences allow verifying, during the testing process, whether the implementation of each use case is in accordance with the corresponding specification. These sequences take into consideration not only the dynamic interactions between objects, but several aspects related to their control as well (preconditions, post conditions, sequencing, control structures related to the interactions, etc.). The verification process is based on some extracted information during the analysis phase, such as messages post conditions, from the collaboration description and integrated automatically, during the instrumentation phase, to the code of the software under test. The generated sequences are executed incrementally.

Abdurazik et al. [1] have stated that test sequences can be generated based on the sorting of the interactions within collaboration. This work constitutes, in our opinion, an interesting starting point in this area. However, such a technique is certainly applicable in the case of simple collaboration, in which the sequence of messages is rather linear. In the case of complex collaborations, there might be, depending on the specified control, several possible sequences for a unique use case. The generated sequences have to allow testing the entire possible cases described in collaboration (basic flow and its alternatives). Hence, the sequences have to take into account the different aspects of the control and conditions leading to the interactions between objects involved in the collaboration.

3 Collaboration Diagrams

The different UML notations [6] allow specifying several aspects of object-oriented systems. Individual behavior of objects is described in the state charts. Dynamic interactions between objects are specified in collaboration diagrams. However, UML

collaboration diagrams, in their actual version [13], exhibit in our opinion a weak insufficiency. Indeed, they allow specifying many important aspects in collaboration within a group of objects, such as the order of messages, the conditions related to their execution, and some aspects related to the control in the collaboration. However, they do not allow specifying precisely all of the control structures related to the interactions. A deterministic and a complete description of the control structures are necessary to get the different possible scenarios in collaboration. This description will be taken as a basis for the test sequences generation process.

We propose some simple extensions illustrated in the example given by figure 2. They essentially allow a better expression of iteration according to a given condition. The two messages `msg6()` and `msg7()` are executed in the same iteration, which is a «repeat» iteration. In terms of control, hence messages sequences, this iteration is different from a «while» iteration as illustrated in figure 3. The distinction between the different types of iterations is performed by a code (F: for iteration, R: repeat iteration and W: while iteration), in replacement of the code `<*>` used in UML. Furthermore, since both messages `msg6()` and `msg7()` (Figure 2) are executed within the same loop, we just add a number to identify the actual loop (1 in the illustrated example), right after the iteration type.

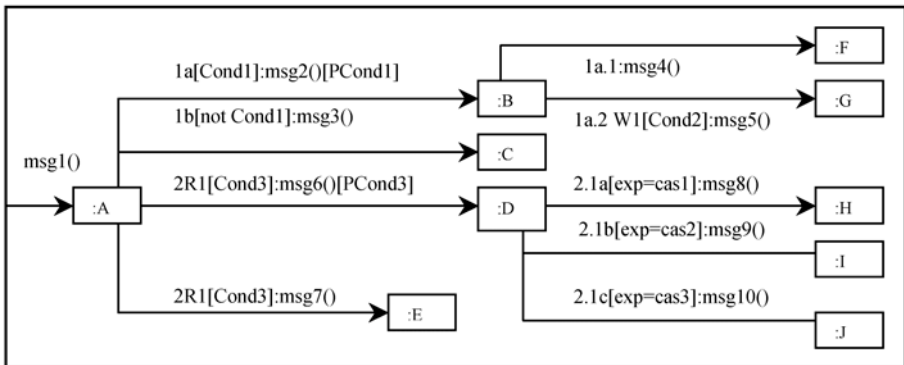


Fig. 2. Example of a collaboration diagram

These simple extensions, which to our knowledge do not exist in an explicit manner in the actual formalism of UML collaboration diagrams [13], allow a precise expression of the different control flow structures and conditions leading to the interactions. Moreover, the formal verification process is based on message pre and post conditions as illustrated in figure 2 (`msg2()` and `msg6()`). The post conditions are integrated in the code of the application during the instrumentation phase.

Collaboration diagrams are described, at the actual stage of our work, by using the CDL (Collaboration Diagrams Description Language) language that we have developed. CDL is a simple language allowing textual description of the semantic content of a collaboration diagram. This gives a solid basis for the testing process, as we will see it in the following sections. In its actual version, CDL allows essentially the description of high-level information contained in the specification of collaboration between objects (message starting the collaboration, different interactions, interactions order, control structures and the conditions leading to the interactions, post conditions, etc.). In fact, CDL represents a complementary tool to UML and

OCL [16]. CDL description represents a strong basis for the test sequences generation process. Figure 4 gives an example of a CDL description. Moreover, CDL descriptions will support the verification process of the compliance of a certain part of the obtained results, versus the expected results (based on the specified post conditions). The CDL language allows also performing, during the design phase, several static semantic controls.

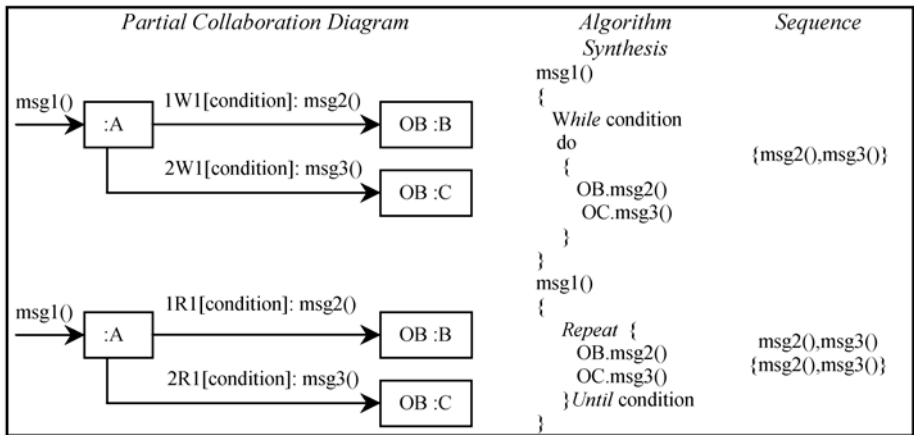


Fig. 3. While and Repeat Sequences

Message msg2()	Message msg5()
Sequence expression : 1a	Sequence expression : 1a.2 w1
Pre_condition : [Cond1]	Pre_condition : [Cond2]
Post_condition : [PCond1]	Post_condition :
Return value :	Return value :
Arguments :	Arguments :
Receiver_Object : B	Receiver_Object : G
Sender_object : A	Sender_object : B

Fig. 4. An example of CDL description

The main objective of our work was essentially to develop the main phases of the testing process. It is for this reason that we have developed a simple language (CDL) for allowing us to describe and analyze the collaboration diagrams. Formal notations such as Z [15] and VDM [9] are certainly rich notations. However, they suffer from the problem that they are very costly to introduce into software development environments, as is the case with most formal methods. Moreover, these notations are not adapted to support the complete description of collaboration diagrams. Notation such as OCL [16], which was created for the distinct purpose of navigating UML models, is ideal for describing constraints and expressing predicates when a system is modeled using UML. Object-Z [7] presents, in our opinion, a good opportunity for describing formally the collaboration diagrams as stated in [2]. We plan, in our future work, to examine the possibility to introduce notation such as OCL or Object-Z. This will improve the developed testing process by giving it a more solid basis.

4 Test Sequences Generation Process

The test sequences generation technique takes into account the several aspects of the control described in a collaboration diagram and related to the interactions between objects. The main objective of the adopted approach is the generation of the set of theoretical paths, from a given collaboration diagram, starting from the initialization of the collaboration to the end, while considering the nature of the interactions (conditional, unconditional, iterative, messages sequences, exclusion between messages, etc.). Actually, every path corresponds to a particular execution of the use case and will be the subject to a particular test sequence. The generated sequences will allow verifying whether the implementation of a use case is in accordance with the corresponding specification. The test sequences generation technique is organized in several steps.

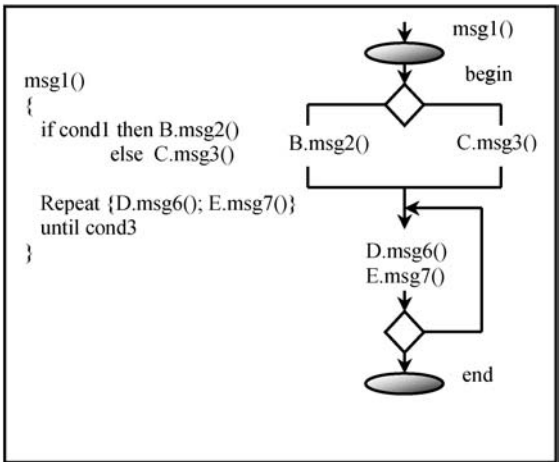


Fig. 5. Messages control flow graph

4.1 Messages Control Flow Graphs

The objective of this stage is to perform an analysis of the CDL description of a collaboration diagram, in order to construct a synthesis of the algorithms of the operations involved in the collaboration. These algorithms will allow the construction of control flow graphs, restricted to the messages of each operation. Control flow graphs have been used in several conventional structural testing techniques. Moreover, they provide a global view of the control in collaboration. Figure 5, presents a synthesis of the algorithm and the messages control flow graph of the operation (msg1()) that starts the collaboration described in Figure 2.

4.2 Messages Tree

During this step we perform, first of all, an analysis of the different control flow graphs corresponding to the operations implied in the collaboration. This analysis allows generating the main messages sequence corresponding to the analyzed dia-

gram. This sequence will be taken as a basis for a complete messages tree construction process. The starting point of the tree will be represented by the entry point of the operation starting the collaboration. Figure 6 shows the main messages sequence corresponding to the operation msg1().

A.msg1(), (B.msg2() / C.msg3()), D.msg6(), E.msg7(), { D.msg6(), E.msg7() }

Fig. 6. Main Sequences

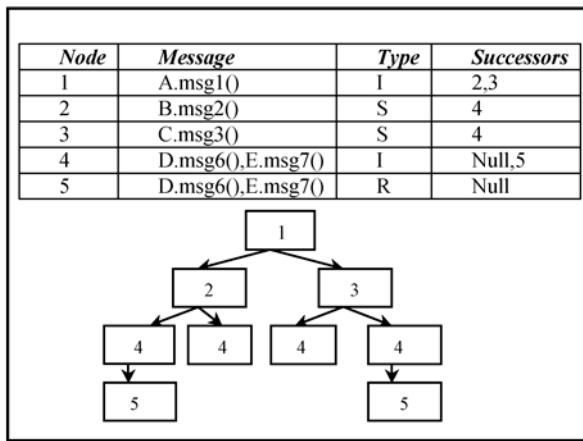


Fig. 7. Messages Tree

We use several notations in order to express the different possibilities in the sequences depending on the control. Notation $\{sequence\}$, expresses 0 or multiple executions of the sequence. Notation $(sequence\ 1\ /\ sequence\ 2)$ expresses an alternative between either sequence 1 or sequence 2. Notation $[sequence]$ refers to the fact that the sequence of interest may either be executed or not. The main messages sequence, which corresponds to the operation that starts the collaboration, is considered as a basis to messages tree construction process. Each message of the sequence will be replaced by its own main sequence. The substitution process will stop at the level of the messages that constitute the leaves of the tree. These latter correspond to operations, which do not call any other operation in the collaboration. Figure 7 illustrates the messages tree that corresponds to the main sequence presented in Figure 6.

4.3 Test Sequences

The proposed technique consists on generating, from the messages tree corresponding to a collaboration diagram, the set of theoretical paths, starting from the collaboration initialization to the end, while taking conditions (pre conditions and post conditions) into consideration. A particular test sequence will correspond to each generated path. The generated sequences correspond to the different possible cases of implemented functionality execution. The sequences generated from the messages tree correspond

to the entire theoretical paths of the tree, from the root to the leaves. At this point, the objective is to identify among the set of possible theoretical paths T , corresponding to a collaboration diagram CD an interesting set of executable paths T_r , by the application of some reduction rules. The set T_r of the paths obtained after reduction will be used as a basis for generating the set S of test sequences. To each path of T_r will correspond a test sequence from S , which represents a particular scenario of the use case. The retained rules at the actual point of our technique are:

Infeasible paths: The predicates analysis allows determining conflicting predicates in certain generated paths. These paths cannot be executed and will be reduced from the set T .

Reduction of paths including cycles: The collaboration diagrams including iterative interactions may present a large number (infinite in some cases) of paths. For obvious reasons, we cannot test all of these paths. We do consider through, the hypothesis that these paths constitute a family of similar paths (iterative part of the sequence) and testing only one path (unique iteration) will be sufficient. This hypothesis enables a considerable reduction in the number of potential paths. Figure 8 shows the sequences generated from the messages tree of Figure 7 after reduction.

A.msg1(), B.msg2(), D.msg6(), E.msg7(), D.msg6(), E.msg7() A.msg1(), C.msg3(), D.msg6(), E.msg7(), D.msg6(), E.msg7() A.msg1(), B.msg2(), D.msg6(), E.msg7() A.msg1(), C.msg3(), D.msg6(), E.msg7()

Fig. 8. Generated sequences

5 Verification Process

5.1 Testing Process: Main Phases

The testing process presented in this paper is incremental. It is organized in several phases (Figure 9). The objective is to execute, for each use case, at least once all the retained sequences and verify their execution accordance to the corresponding specification. The idea is also to verify on executing the use case that the executed sequence of messages corresponds to the expected one, according to the provided input data.

The verification process is supported partly by the instrumentation of the program whose aim is adding to the code some operations that allow tracking, by way of the execution analysis, whether the sequence was correctly executed or not. Any deviation of the executed sequence from the expected one will be considered as a failure. The verification process is also supported by some extracted information (essentially the post conditions in the actual version of the environment) from collaboration description and integrated as private methods in the receiver class, during the instrumentation phase, to the code of the application. The user intervenes (Figure 10), of course, during this process to verify the results of the use case execution, which are not specified in the post conditions.

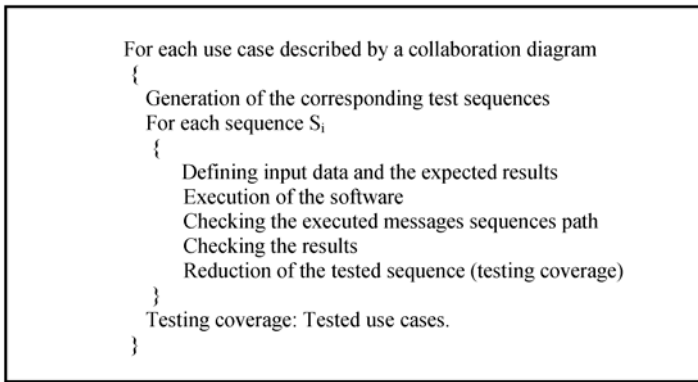


Fig. 9. Testing process phases

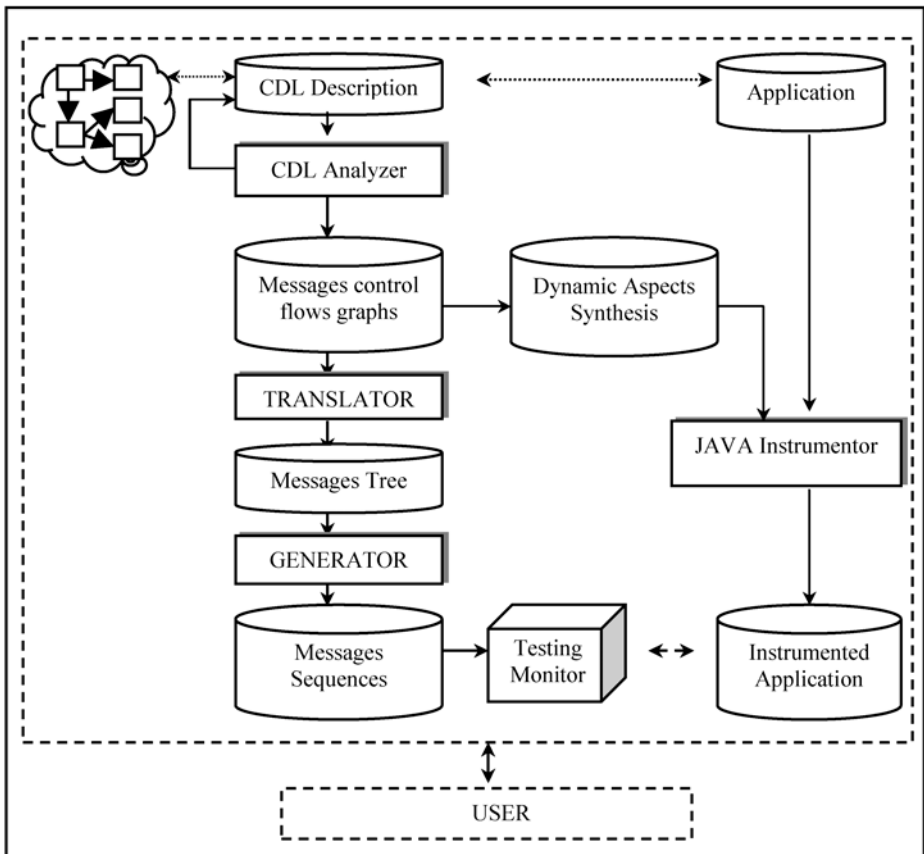


Fig. 10. Environment architecture

5.2 Testing Criteria

The generated sequences are executed incrementally. This process will allow, on the basis of several executions tracking analysis, determining the executed sequences (sequence coverage: main scenario and its different extensions), hence the sequences, which remain to be executed. We have defined, at the actual stage of our research, two types of testing criteria and the corresponding coverage:

Interactions Between Methods: Each interaction, in the collaboration diagram, must be executed at least once. The interactions coverage (IC) is defined as: $IC = \text{Number of executed interactions} / \text{Total number of interactions in the collaboration diagram}$.

Message Sequence Path: Each retained sequence must be executed at least once. Each sequence corresponds to an implemented messages sequence path. The sequences coverage (SC) is then defined as: $SC = \text{Number of executed sequences} / \text{Number of retained sequences}$.

6 Environment Supporting Our Approach

We believe that the environment that we have developed represents an interesting framework for the validation of the use cases of a system. It is composed of several tools as illustrated in Figure 10. Let us consider the example of collaboration diagram given in figure 11. This example was taken from [10] and adapted for the needs of our paper. Figure 12 illustrates the result of the analysis of the corresponding CDL description and the generated sequences.

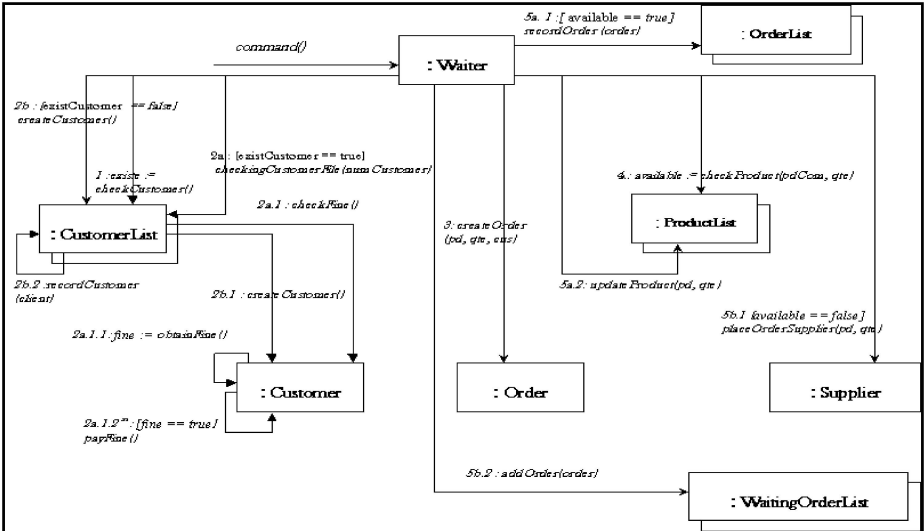


Fig. 11. Example of collaboration diagram

is not valid. The tester, in such cases, will be informed about the detected failure. Moreover, knowing that there will be an error exception, the tester will exactly know what part of the tested sequence is in failure. This will be useful for debugging the program. Figure 13-2 gives a case where the executed sequence is not in accordance with the expected one. The environment will stop the execution at the level of the first deviation of the sequence from the expected one. This also will be useful during the debugging process.

7 Conclusions and Future Work

The object-oriented testing process proposed in this paper is a new approach. It is based on the dynamic interactions between the different groups of objects, specified in the collaboration diagrams. It offers many advantages in comparison with a code-based generation approach. This allows emphasizing eventual omissions in regard with the specification, on the one hand, and design errors related to the different interactions between objects, on the other hand. In addition, test sequences generation early in the development process allows a better preparation of the testing process.

The test sequences generation technique proposed in the present article is based on formal description of objects groups' behavior, particularly the concept of message post condition. This gives a solid basis for the verification process. This technique allows, for each use case (or a complex operation of an object), from the analysis of the formal description of the corresponding collaboration diagram, the generation of a set of appropriate test sequences. The test sequences of interest, take into consideration not only the dynamic interactions between objects, but also the different aspects related to their control. Each sequence corresponds to a particular scenario of the use case, expected during the analysis and design phases. The objective is to cover, using the generated sequences, a large number of the different possible executions of a use case and check, thanks to the supported verification process, the accordance between its implementation and its specification.

The developed environment for supporting our approach has been experimented on simple Java projects. We plan, in our future work, to use it on real projects. We plan also to extend our approach by introducing formal notation such as OCL or Object-Z, in the one hand, and to extend the environment to others object-oriented languages such as C++.

Acknowledgements

For their financial support, we would like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) and the UQTR internal funding program (FIR).

References

1. Abdurazik, A., Offutt, J.: Using UML collaboration diagrams for static checking and test generation, Third International Conference on the Unified Modeling Language, 2000.
2. Araujo, J., Moreira, A.: Specifying The Behavior Of UML Collaborations Using Object-Z, AMCIS'00, 2000.

3. Badri, M., Badri, L.: Génération incrémentale de séquences de test fonctionnel des classes: Vers une approche formelle basée sur l'héritage du comportement des objets, 12th International Conference on Software and Systems Engineering and their Applications (ICSSEA'99), Paris, December 1999.
4. Badri, M., Badri, L.: Génération de tests à partir des diagrammes de collaboration UML, in *Génie Logiciel*, n° 60, March 2002.
5. Bashir, I., Goel, A.L.: *Testing object-oriented software: Life cycle solutions*, Springer-Verlag, 2000.
6. Booch, G., Rumbaugh, J., Jacobson, I.: *The Unified Modeling Language User guide*, Addison-Wesley, Object Technology Series, 2000.
7. Duke, D., King, P., Rose, G.A., Smith, G.: *The Object-Z Specification Language, Version 1*, Technical Report 91-1, Department of Computing Science, University of Queensland, Australia, 1991.
8. Jacobson, I.: Use Cases and Aspects – Working Seamlessly Together, in *Journal of Object Technology*, vol. 2, no. 4, July-August 2003.
9. Jones, C.B.: *Systematic Software Development Using VDM*, Prentice Hall, 1986.
10. Larman, G.: *Applying UML and Patterns: An introduction to object-oriented analysis and design and the unified process*, second edition, Prentice Hall, 2002.
11. Marick, B.: *The craft of software testing*, Prentice Hall, 1995.
12. McGregor, J.D., Sykes, D.A.: *A Practical Guide to Testing Object-Oriented Software*, Addison-Wesley, 2001.
13. Object Management Group: *OMG-UML Specification, Version 1.4*, September 2001.
14. Pressman, R.S.: *Software Engineering, A practitioner's approach*, Fifth edition, Mc Graw Hill, 2001.
15. Spivey, J.M.: *The Z Notation: A Reference Manual*, second edition, Oriel College, Oxford, 1998.
16. Warmer, J., Kleppe, A.: *The Object Constraint Language: Precise Modeling With UML*, Addison-Wesley, 1998.

VISWAS and on Diagnosability with IEEE Std P1522 and UML2.0 Testing Profile

Sita Ramakrishnan

School CSSE, Monash University, Australia
sitar@csse.monash.edu.au
<http://www.csse.monash.edu.au/~sitar>

Abstract. We first describe the work on validating interoperable distributed software and systems - VISWAS method [1], which extends UML models to include testability aspects with design by contract notions. In VISWAS, automated test sequence generation is produced from the extended Message Sequence Charts (MSCs) and Live State Charts with temporal action propagation list. Since testability is not independent of the diagnostic process, diagnosability was not explicitly stated as part of the SDLC and testing process in VISWAS. Next, we present our current work on capturing diagnosis flows in MSCs, and some discussion on why standards such as IEEE Std P1522, UML 2.0 Testing Profile, MSC2000 and TTCN-3 are useful for consideration in capturing diagnosability aspects in a testing environment.

1 Introduction

For validation testing in an O-O context, the testing strategy is broadened to incorporate the review of analysis and design models. UML models and technology have primarily been used in defining static system structures and dynamic behaviours. Before OMG RFP on a UML testing profile (refer: <http://www.fokus.gmd.de/u2tp>) in July 2001, UML officially provided only some limited mechanisms for describing test procedures, although UML was extended to cater for such tasks by various research groups and industry-based projects. With the recent trend in systems engineering approach with model driven architectures and automatic code generation, the need for conformance testing has increased.

1.1 Design for Testability

Testability and diagnosability are related attributes that must be taken into account when building, measuring and predicting the correctness and robustness of software components. Testability is viewed as a software design quality characteristic which allows the status of an item under test to be determined and the detection of faults within the item under test to be performed effectively. Diagnosability is a wider notion than testability and encompasses finding information or explanation about the state of a system under test and includes faults and no faults.

Software testability is defined in the *IEEE Standard Glossary of Software Engineering Terminology - IEEE Std. 610.12 - 1990* as “(1) the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met, and (2) the degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met” [2]. According to this definition, one needs to have appropriate test criteria to determine the degree of testability. Hence, testability is a measure of the difficulty in satisfying a specific testing goal. Testing can reveal faults whereas testability can suggest places where faults can hide from testing. Voas [2] defines “testability” of a program P to be the probability that a particular testing strategy will force failures to be *observable* if faults were to exist and the probability that the *oracle* will be precise enough to detect failures. This approach is different from the IEEE definition above.

As software testing is used to show conformance between specification and implementation, a good software engineering practice requires that testability be addressed by precise and clear specifications and rules to map between various phases of software cycle, for example, between implementation and design models.

Testability of O-O systems is lower because O-O design impacts on the controllability and observability of objects or components under test. A practical engineering method called *VISWAS* for testing the safety and liveness properties of distributed reactive systems using a mainstream method such as UML was developed in [1]. In the primary author’s work, testing method called *VISWAS* (Validating interoperable distributed software and systems) [1] attempts to:

- overcome the limitations of UML/OCL in expressing temporal constraints for expressing safety and liveness properties; precisely specify these constraints by expressing these extensions to MSCs and State Charts in the Temporal Logic formalism and Temporal Logic of Actions (TLA)
- integrate MSC and State Charts into the testing process; develop a repeatable model engineering process for a testing method
- develop an automated specification test generation tool as part of the test method.

1.2 UML2.0 Testing Profile

UML models focus primarily on system structure and behaviour, and does not provide details for specifying test procedures and objectives. In June 2001, an OMG RFP on UML2.0 Testing profile (UTP) was initiated to address this gap. UML 2.0 Testing Profile (UTP) is based on recent work on testing such as TTCN-3 [3]. UTP has the notion of an **arbiter** which is a new test component aimed at separating test behaviour from test evaluation. Some of the other major terms in UTP are: test architecture, test data and test behaviour. The test architecture is a set of related classes and/or components from which test cases can be specified. The test data package contains data sent to system under test

(SUT) and received from SUT. A test configuration contains the test components and the SUT. An arbiter is a specific test component for evaluating test results and to assign **verdicts** of a test case. A verdict is the outcome of a test case being **pass**, **fail**, **inconc** or **error** as defined in TTCN-3. During the execution of a test case, a **test trace** is generated and stored in the test log.

1.3 IEEE Std P1522

The Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE) standards are product information standards for test and diagnosis. The 1232 family of standards were developed for providing standard exchange formats and software services for reasoning systems used in system tests and diagnosis. As the information models for these standards become more complex, and systems became difficult to diagnose and repair, IEEE Std P1522 initiative got under way for standardising testability and diagnosability metrics. The metrics of P1522 are derived from the information models in IEEE 1232. [4] states that the main purpose of testing is diagnosis and that paper provides a detailed dynamic context model showing the relationships in a test/diagnosis session. P1522 includes both testability and diagnosability, and diagnosability includes all aspects of fault detection, fault localization and fault identification.

Diagnosability was not the focus of concern in VISWAS, and the test execution phase of test environment in VISWAS was not automated. We are exploring the IEEE P1522 standard and UML2.0 Testing Profile for modelling and automating the test execution phase of test environment in VISWAS.

1.4 From Testability to Diagnosability Aspects

While testability is concerned with detecting faults, diagnosability aspect of testing is about isolating and pointing out fault location to repair/correct them. Software testing involves executing code with test input and determining the outcome of the test. We look at testing from two complementary objectives:

- conformance-directed when the intent is to achieve conformance to requirement specification
- fault-directed when the intent is to reveal faults through failures. We treat this as a diagnosis activity.

The word *Diagnosis* is derived from two Greek words and is defined as *to distinguish, and in a legal sense, to examine and offer an opinion*. Our objective is to ascertain with confidence the state of the software components under test such as correctness, robustness including faults and failures. Testing is considered effective when it reveals existing faults.

We present the VISWAS method in the next sections. In Section 3, we show how the test oracle in a test environment (see Figure 1) could be expanded to provide detailed diagnostics information. In Section 4, we describe how the diagnosability models for such software systems would benefit from using IEEE

Standard P1522, UML2.0 Test Profile, MSC2000 and TTCN-3 Graphical Notation. This will also involve capturing diagnosis flow in Figure 3 by using next event/signal relations in MSCs (see Figure 7) and [5]. The diagnosability models may start to look cluttered and decidedly procedural using UML2.0 Test Profile or MSC2000 and TTCN-3 as can be seen from [6,7]. However, in mission-critical applications domain, software component market and in embedded software systems with hardware/ software, it is important to capture diagnosis flow between objects or within an entity graphically in order to communicate our diagnosis requirements unambiguously in the detailed design phase. Such diagnostic flows and verdict information from the oracle can be used to develop software quality measures by collecting and analysing the various diagnosis information.

2 A Test Method for Validating Interoperable Distributed Software and Systems (VISWAS)

2.1 Defining VISWAS

In concurrent and distributed modelling, two properties called safety and liveness are important to consider. Mutual exclusion and absence of deadlock are important safety properties in developing concurrent models and programs. Progress property is a restricted form of liveness property. A progress property asserts that from a state that a system is in, it is always the case that a specified action will *eventually* be executed. A test environment needs to be defined for the development of this testing method (VISWAS) for distributed systems. The problem considered is the generation of test sequences at a tool level from an O-O model representation of the distributed system taking into account assertions on time-related constraints. The choice of an O-O model, which is testable and can be used for testing, is needed to design the test software. The choice of a suitable model of the distributed system is required for representing the system as a testable dynamic model of the system. In order to reduce the manual effort and improve the accuracy of the output, one of the requirements for testing is to automate the test sequence generation.

2.2 A Generic Test Environment

The test environment for VISWAS is a specific case of the generic test environment (see Figure 1).

The test environment includes both the test sequence generation and test execution phases. The generic testing environment shown in Figure 1 has three main aspects:

1. The test model, which is an application model augmented to include testing requirements by adding application-specific constraints as part of the object or component's interface protocol,
2. The test model (state model) is fed to the test design software. This software tool uses the critical properties of testing requirements built in the test model to generate automated test sequences.

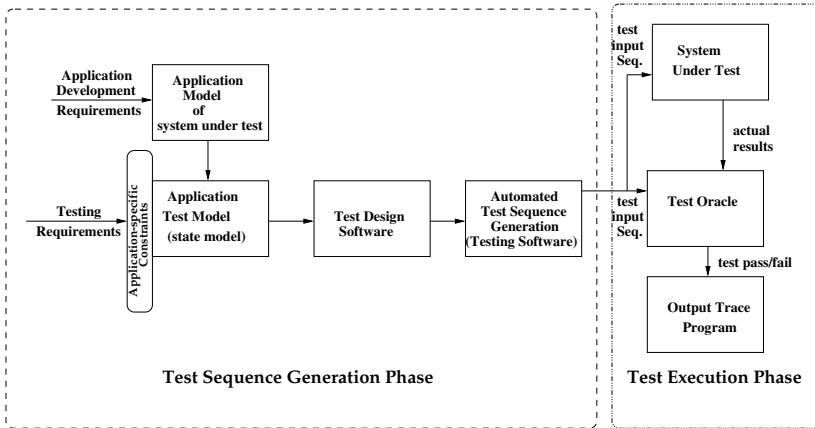


Fig. 1. Generic testing environment

3. The test execution phase is the diagnosis phase and accepts test inputs from the previous phase for the system under test, and the test is evaluated as pass or fail. The components shown in the test execution phase in Figure 1 are used in the validation of system under test.

2.3 Modelling for VISWAS

The model engineering process in VISWAS is defined by a three layered meta-model to facilitate the development of detailed models for implementation and testing of concurrent and distributed O-O components.

The three layers of the VISWAS architecture as shown in Figure 2 from the model engineering view point are:

1. Representational layer - to model the given concurrent and distributed system to be tested
2. Assertional layer - to capture the safety and liveness properties of the given system for testing
3. Tool layer - to develop software testing tools, for example, to automate test sequence generation in VISWAS

The three layered model is used to create a customised process specifically suited to the testing requirements of an interoperable distributed software system.

The representational layer has been chosen to be a distributed architectural model, which supports interoperability and address testability concerns of distributed software systems. The distributed architectural model of Ken-Gate by Schmidt [8] fits this requirement, and is used to represent the architecture of the software system. As part of this layer, an implementation model is also derived from the static architectural model to represent the distributed software system under test. Industry standard tools and languages such as CORBA/JAVA IDL

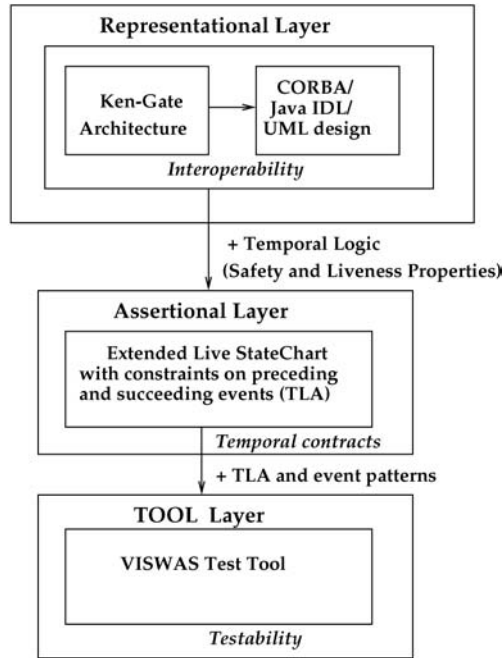


Fig. 2. Modelling layers in viswas test method

and Java are used in the implementation model for the software system under test.

The Assertional layer is used to represent the dynamic software model with state machines and MSCs. As part of the dynamic model, temporal properties must be included explicitly in order to capture the safety and liveness requirements for distributed component testing. However, since the current methods such as the UML and OPEN dynamic models do not currently support any temporal operators, OCL extended with temporal logic operators is used in MSCs, and the temporal logic of action (TLA) formalism is applied to the state machine. As shown in Figure 2, temporal operators, safety and liveness properties of TLA are added to the UML state machine to produce an extended statechart with constraints on preceding and succeeding events. These constraints together with the extended design by contract from the distributed architectural model represent the temporal contracts that are modelled as part of validating an interoperable distributed system (VISWAS).

The Tool layer is used to represent the automated VISWAS testing tool, and support the testability needs of a distributed software system. The extended live state chart with its temporal contracts, and event patterns that take important safety properties, such as mutual exclusion, absence of deadlock with *always* and *never*, and liveness properties such as *eventually enabled* into account, are fed to the tool layer to generate a *functional test sequence generator*.

3 An Automated Specification-Based Testing Tool

In the development of VISWAS, precise meaning for the various elements in the example MSC for a Robot have been provided. It is described briefly in Subsection 3.1. More details are available in [9,1]. Subsection 3.2 provides details of the *Live State Chart*, and how it is used in the automated test sequence generation. A test sequence generation is one of the components of the testing environment. A test environment is presented for an automated specification-based testing tool for Distributed Object Software Testing (DOST) in VISWAS method in Subsection 3.3.

3.1 Message Sequence Chart of a Robot Component

Our MSC for a Robot (see Figure 3) has a clear definition of the mandatory vs provisional elements of MSC. It is therefore possible that one can proceed with defining the interaction semantics on the horizontal dimension of MSC using either OCL or events with guards. Guards may be empty or contain predicates which must be met for the interaction to occur.

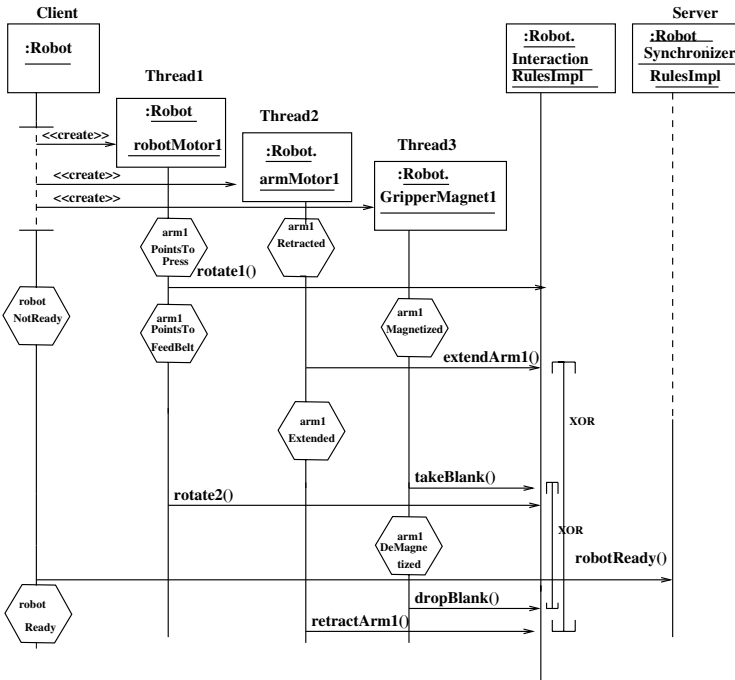


Fig. 3. Message sequence chart of a robot component

We have described MSC object interactions precisely in terms of **what**, **when** and **where** they occur. Our Robot example Figure 3 describes a robot drop operation scenario. There are six object/thread instances: Client Robot and its three

threads - `robotMotor1`, `armMotor1`, `gripperMagnet1`, CORBA (`InteractionRulesImpl`) and a `Server`, each of which is depicted with a line on the vertical axis. Each of these instances is represented by a rectangle with its name underlined followed if appropriate by the thread or CORBA specific stub or skeleton name. The thread creation messages from the Client life line to the three thread objects are shown coming from a **coregion**. The coregion is drawn as a dashed vertical line. The events in the coregion are not ordered whereas events in an object instance is totally ordered along the vertical line. A hexagon crossing over an instance represents the condition to be satisfied by the instance during execution. For example, `robotMotor1` thread object must be in the `arm1PointstoPress` state at the beginning of this scenario. Shared conditions are represented by hexagon crossing over two or more objects. A horizontal arrow means communication via message passing between two instances. For example, the message `rotate1()` is passed from `thread1` to `InteractionRulesImpl`. The server object is shown with a dashed life line to show existentiality. It need not be alive until all the thread events of the robot component object `rotate1()`, `extendArm1()`, `takeBlank()`, `rotate2()` have happened. The server life line is a solid line, as it must exist (live) when the client object sends a `robotReady()` message to the server.

Two extensions to the MSC notation were introduced: One is a textual annotation - XOR to show mutual exclusion. Beside the annotation is the second extension which is a timeline shown as a solid line that stretches between the two mutually exclusive methods. This timeline has a cap (see Figure 3) at the top and bottom of the line. The cap ensures clarity in the depiction of the scope of mutually excluded methods. The aim was also to integrate MSCs and statecharts in the development and testing process. The prototype test sequence generator tool covers intra component validation testing of distributed reactive systems.

3.2 Live State Chart and Test Sequence Generation

The *Live State Chart* is a visual dynamic UML model, and this model is used in the automation of test sequence generation. A textual extended BNF notation is used to provide a Grammar for the *Live State Chart*. The UML metamodel is used to provide a textual description for the *Live State Chart*. The *Live State Chart* of the distributed reactive component, Robot example is transformed into an event tree using JavaCC tools. A prototype implementation tool for the automated test sequence generation is derived from the event tree of the *Live State Chart* Algorithm developed for automating the generation of test sequences for a distributed reactive system is described in [1].

As shown in Figure 4, the test sequence generation phase involves the following:

- *Live State Chart* (LSC) with Temporal Action Propagation ($\mathcal{TAP}$ Sequences)
- Textual Model of LSC with $\mathcal{TAP}$ that conforms to the Grammar defined here
- JavaCC tools (JJTree and JavaCC)
- test sequence generator tool

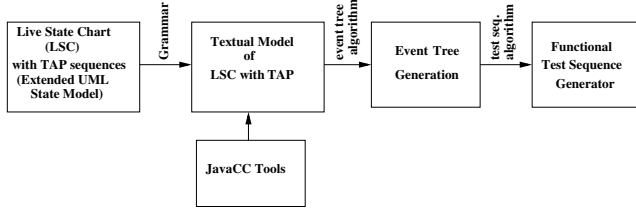


Fig. 4. Test sequences generation phase

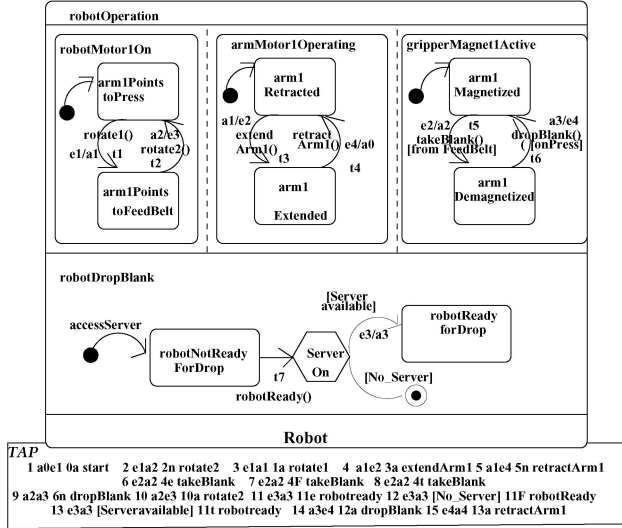


Fig. 5. Live state chart with TAP (Robot Example)

The essential (core) requirements of the testing environment considered are: the specification model to be fed to the JavaCC tools in the context of a Grammar, creation of a *functional test sequence generator* tool, and a system under test for testing against the input test sequences.

The rest of the infrastructure such as the test oracle and the output trace programs deal with the diagnosability aspects and are required to provide a fully functional automated specification tool. However, this test execution phase is not automated with this environment (Figure 6) in [1] which is the focus of our present work.

Live State Chart's event activation sequences and the extensions, introduced in the MSC to show mutual exclusive methods in a previous subsection are revisited here to introduce $\mathcal{TAP}$. Temporal operators with the formalisms from TLA are shown explicitly with event/action sequences in the LSC. In order to reduce the clutter in the body of the diagram, a temporal action propagation ($\mathcal{TAP}$) list was introduced. $\mathcal{TAP}$ lists the event action propagation sequence for the transition with the appropriate temporal operator (as shown in the rectangular attachment entitled $\mathcal{TAP}$ shown as part of the Live State Chart Figure 5).

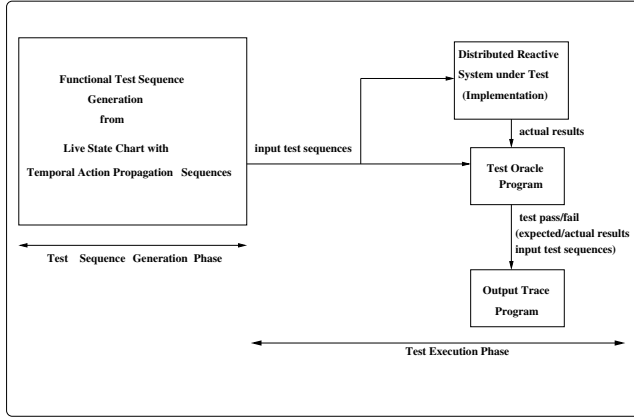


Fig. 6. Testing environment

In $\mathcal{TAP}$ in Figure 5, $a0e1$ means always 0 eventually 1. These operators are *always* (a) and *never* (n) for mutual exclusive methods, and hence represent safety properties. Although *never* (n) constraints are not shown in the body of the live state chart, they are part of the testing requirements. Hence, *never* (n) constraints are included in the TAP list. The extension to MSC (see Figure 3) shows the mutual exclusion condition (*always* and *never*) clearly. This information from the MSC has been used to include the *never* (n) constraints in the TAP list. The liveness properties are shown as *eventually* (e) true (t) or false (F). For an *Eventually* (e) constraint in the live chart, both (t) and (f) are included in the TAP list to accommodate *eventually true* and *eventually false* conditions. The *Live State Chart* has the temporal action propagation($\mathcal{TAP}$) sequence list, which lists for each activation sequence:

- a sequence number to indicate the order in which the event occurrences are considered
- the *event action sequence IDs* which match the IDs included in the body of the *Live State Chart* plus the default *start* event
- the *true* and *False* activation for an *eventually true or false* event
- temporal constraint number
- temporal property and
- the transition label.

From the description of the Live State Chart with $\mathcal{TAP}$ for a Robot example, a generalised, abstract model of the Live State Chart with $\mathcal{TAP}$ has been derived in [1]. This abstract model definition is one of the steps in the setting up of a repeatable validation testing method (VISWAS) for a distributed reactive component.

3.3 Testing Environment in VISWAS Method

Our testing environment in VISWAS supports such an automated process (see Figures 1 and 6), and includes:

- the Test sequences Generation Phase, and
- System Under Test.

In order to support an automated validation process, the test execution phase of the test environment must include the following:

- The system under test accepts the automated test sequences generated under the control of the test oracle program; the test oracle program to run the system under test with the test data provided by the test sequence generator
- The actual results are compared against the expected results and the test is deemed as pass or fail and reported in the output trace program; the output trace is a reporting mechanism (verdict) by the test oracle of tests that passed and those that failed

The components of the test execution phase are described briefly.

Test Oracle. A test oracle program verifies the behaviour of the test execution by using the oracle data (*expected results*), together with the test input and output (*actual results*). The accuracy of the behavioural verification depends on how good the test oracle is.

A test oracle program can be built to use the *functional test sequences* produced by the test sequence generator, which is described in the next section. The test oracle once built and compiled, can be used to execute against the system under test to detect any violations of properties covered by the test sequences. A test oracle is used either with an entire test sequence (test suite) or with a single test case. The current design (as shown in Figure 6) allows the test oracle to compare the outcome with the expected results. Further analysis of the test results is conducted by the output trace program.

Output Trace. The output trace is used to collect the verdict from the test oracle. The output trace should receive the input test sequences from the automated test sequence generation phase. The test sequences are grouped into valid and invalid sequences, and cover all the event intervals. It should also obtain the *the actual result* and the *expected result* from the test oracle. Hence, the output trace also contains coverage metrics for the system under test. The output trace mechanism is therefore useful in further analysis of test results.

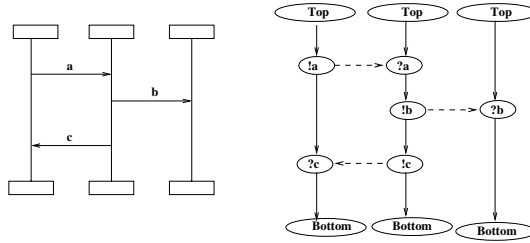
4 Capturing Diagnosis Flow in MSC

We revisit MSCs here to show how Ladkin et al. 's [10] work described in [1] and Padilla's [5] work on an execution semantics for MSCs can be used in describing a set of possible traces which are treated here as test sequences and diagnosis flows.

4.1 MSC to Next Event/Signal (ne/sig) Relation Graphs

Ladkin et al. [10] use ne/sig graphs as abstract syntactic representation for MSCs. ne/sig graphs have two kinds of edges - **next event** (ne) and **signal**

(sig) edges. These edges represent signals and the progression of processes between events. The nodes represent events and are labeled with the event type. The event node sending a message of type **a** at the sig edge is labeled **!a**, and the event node receiving that message of type **a** labeled **?a**. A ne/sig graph has start nodes labeled **Top** and end nodes labeled **Bottom** (Figure 7).



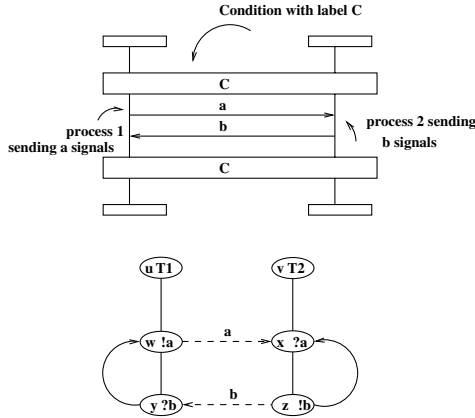
source: Ladkin and Leue 1994 (FDT VI Proceedings)

Fig. 7. An msc and its corresponding ne/sig graph

MSCs have vertical lines drawn from each object/process/thread. Events are ordered temporally from top to bottom. The horizontal lines in the graph represent an event at which signals of the specified type are sent or received by the object/process/thread. The system is terminated when all the objects/processes/threads have terminated. To translate an MSC into a ne/sig graph, events are drawn as nodes, and the signal edges become relations on the nodes (Figure 7). Two types of signal edges are needed to handle both the asynchronous and synchronous communication. Therefore, there are two signal relations on graph nodes. The state transition relation on these two types of nodes in the ne/sig graph is defined differently, and whether an action is enabled, depends on whether it is an asynchronous or synchronous action. In synchronous communication, both **send** and **receive** events occur simultaneously, communication is atomic, and both the events block until both are ready.

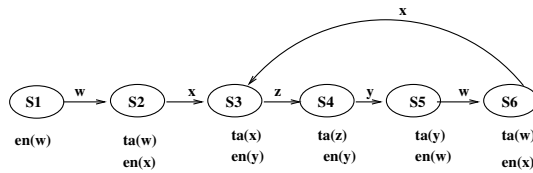
MSCs can contain labeled conditions, shown in Figure 8 as a rectangle spanning horizontally across the object/process/thread axis. In this figure, there is a condition label **C** at the top and bottom. MSC may be joined to itself creating an iterative non-terminating loop, with signal **a** alternating with signal **b**. Translating MSCs with conditions into ne/sig graphs involves introducing extra condition nodes on each process axis, then joining the graphs at these nodes, and unfolding an MSC into a single ne/sig graph (Figure 8).

The triple, called the global state transition graph (GSTG), with global states (Q), the start state (q_o), and the state transition function (T_M) is defined for the ne/sig graph to derive a finite-state automaton. The triple $\langle S1, w, S2 \rangle$ is a transition relation. Say, in the state labeled $S1$, event of type **!a** at node **w** is enabled, as node **w** denotes a **send** node. Node **x** is not enabled, as its **send** has not been taken in $S1$. Since **w** is enabled, the event corresponding to it may be **taken** next to enter a new state $S2$. GSTG is annotated with the list of



adapted from Ladkin and Leue 1994 (FDT VI Proceedings)

Fig. 8. Iterations in msc (msc 2) and its corresponding ne/sig graph



source: Ladkin and Leue, 1994 (FDT VI Proceedings)

Fig. 9. Global state transition graph for msc 2

actions enabled ($en()$) and taken ($ta()$) in each state (Figure 9). Ladkin et al. [10] proposed that MSC specifications may be enhanced by making which liveness properties (weak or strong fairness property) must be satisfied for a given specification. We used these ideas in defining an extended MSC in [1] for deriving automatic test sequences.

We revisit Figure 7 again here to describe possible diagnostic sequences. For each instance, the vertical time axis shows a total order among events called *instance order*. There is a one-to-one correspondence between sending and receiving of each message signal (relation) represented by arrows as shown in (Figures 7 and 8)) [5]. The event of sending message $!a$ is related to the receiving event $?a$. An MSC defines a partial ordering of events composed from the *instance order* and *send-receive relation*, and describes a set of possible test/diagnostic sequences. Figure 7 with its explicit representation of *send* and *receive* signals describes a set of traces that can be used in diagnostic sequences:

trace 1: $!a, a?, !b, ?b$; trace 2: $!a, a?, !b, !c, ?c$

Using this notation in conjunction with the extended MSC proposed in [1] and the UML2.0 test profile provides us with the ability to capture test and diagnosis information with UML.

5 Discussion

5.1 Testability and Diagnosability with Industry Standards

Researchers working on improvement to software quality attributes have produced novel solutions to address testability and diagnosability concerns. Software slicing and dicing are diagnosis techniques from researchers such as [11,12] focus on software code at the unit and integration levels. Another diagnosis method is the use of assertions [13] to locate faulty states in code. [14,15,16,17] use diagnosability to measure the expected effort and difficulty in fault localization, and help with improving the design quality. A number of these researchers use SDL and Testing and Test Control Notation (TTCN-3) notation.

UML is also a widely used industry standard and for software components modelled using UML, it makes sense to use them for test description as well. The primary author [1] has used extended MSCs to describe tests for temporal properties. A test description language, *!TeLa!* is introduced in [18] to be used as input by the executable test generator and synthesise tests specified in UML based formalism. They use UML interaction diagrams to describe tests for temporal ordering properties.

Also, the problems associated with temporal ordering have been explored in the telecom domain and in conformance testing of telecom protocols (see ISO Standard Testing Language TTCN [19]. TTCN-3 [20,3] is the only standardised language for the specification and implementation of test cases. [7,21] describe a graphical presentation format for TTCN-3 (GFT). GFT is based on MSCs and UML and extends it with test specific concepts such as verdicts and defaults. GFT is also the basis for the definition for UML2.0 Test Profile.

[22,23,24] have described parallel test architectures and have used MSCs and activity diagrams to show the testing and diagnosis process. We had (refer to http://www.csse.monash.edu.au/~sitar/se_educ_proj/casestudies/game/PACT.html using Netscape) used Parallel architecture for component Testing (PACT) [22] for building test cases, and PACT architecture parallels the architecture of production classes. A test case may have four outcomes: Pass, Fail, inconclusive or abort without verdict to cater for apriori exception cases. UML2.0 Test Profile, GFT, *!TeLa!* have similar diagnosis features.

A number of papers from [25,26,4] point out that in electronic and other complex hardware systems, artificial intelligence is employed as a primary component in system test and verification. However this has led to proliferation of AI design, test and diagnostic tools, and the lack of standard interfaces between reasoning systems has led to increase in product life cycle costs. The primary purpose of standardization effort such as the Artificial Intelligence Exchange and Services Tie to All Test Environment (AI-ESTATE) for IEEE1232 family and the standard on testability and diagnosability metrics, P1522 is to facilitate the development of diagnostic tools and systems that can be widely used and are predictable. Integrated diagnostics conceptual model proposed in [27] provides direct ties to the testability and diagnosability standard, P1522. The model depicts relationships between test requirements, tests and outcome. The outcome

is based on diagnostic rules with which conclusions are drawn with levels of confidence. Diagnosis can point to failures or faults and corrective actions are undertaken. [27] argues for diagnostic components to be constructed according to standards to facilitate competition in the market place in terms of risks, cost and quality. Standards also imply a maturity in the underlying technology, thus adding to the level of confidence.

We draw inspiration from such standards work in our ongoing work on deriving design and test models in the context of industry standards such as UML models, UML2.0 test profile, TTCN-3, GFT and *!TeLa!* for test and diagnosis.

We are currently investigating various avenues for further work in this area. Some of these areas are:

- building an automated test oracle (diagnostics tool) to capture diagnostics (pass/fail) status of test inputs (from the automated test sequence generator).
- analysing the diagnostic information to measure the goodness of the design and effectiveness of test strategies using the above or existing diagnostic tools.

References

1. Ramakrishnan, S.: Validating Interoperable Distributed Software and Systems - VISWAS. PhD Thesis, School CSSE, Monash University, Australia (2000)
2. Voas, J.M., Miller, K.W.: Software Testability - The New Verification. IEEE Software (1995)
3. ETSI: The Latest Tree and Tabular Combined Notation (TTCN-3) Standard - six-part standard written within the Technical Committee MTS (Methods for Testing and specification) ES 201 813 parts 1 - 5, part 6 to be released in summer 2003. ETSI (The European Telecommunications Standards Institute) (2003)
4. Sheppard, J., Kaufman, M.: An Integrated View of Test and Diagnostic Information Standards. In: AutoTestCon 2002, IEEE Systems Readiness Technology Conference, IEEE (2002) 445–455
5. Padilla, G.: An Execution semantics for MSC2000. Thesis, Uppsala University, Sweden (2000)
6. Haugen, O.: MSC-2000 interaction diagrams for the new millenium. Computer Networks **35** (2001) 721–732
7. Schieferdecker, I., Grabowski, J.: The Graphical Format of TTCN-3 in the context of MSC and UML. In: 3rd Workshop of the SDL Forum Society on SDL and MSC (SAM'2002), Aberystwyth (UK). (2002)
8. Schmidt, H.: Compatibility of Interoperable Objects in Information Systems Interoperability, B Kraemer, M Papazoglou and H Schmidt (eds.). Research Studies Press (1998)
9. Ramakrishnan, S.: Precise Specification Modelling for Testing OO Distributed Reactive Systems. In: 18th IASTED International Conference - Applied Informatics. (2000)
10. Ladkin, P.B., Leue, S.: What do Message Sequence Charts Mean. In: R L Tenny, P D Amer, M U Uyar (eds.), 6th International Conference of Formal Description Techniques. (1994) 301–316

11. Weiser, M.: Programmers use slices when debugging. *Communications of ACM* **25** (1982) 446–452
12. Agrawal, H., Horgan, J.R., London, S., Wong, W.E.: Fault Localization using Execution Slices and Dataflow Tests. In: 6th IEEE International Symposium on Software Reliability Engineering, Toulouse, France, IEEE Computer Society (1995) 143–151
13. Meyer, B.: *Object-oriented Software Construction*. Prentice-Hall (1988)
14. Traon, Y.L., Ouabdesselam, F., Robach, C.: Software Diagnosability. In: IEEE International Symposium on Software Reliability Engineering (ISSRE'98), Paderborn, Germany, IEEE Computer Society (1998) 257–266
15. Baudry, B., Traon, Y.L., Jezequel, J.M.: Robustness and Diagnosability of OO Systems Designed by Contracts. In: 7th International Software Metrics Symposium, London, England, IEEE Computer Society (2001) 4–6
16. Briand, L.C., Labiche, Y., Sun, H.: Investigating the Use of Analysis Contracts to Improve the Testability of Object Oriented Code. Carleton University Technical Report TC SCE-01-10, vers. 3 (2002) 1–48
17. Traon, Y.L., Ouabdesselam, F., Robach, C., Baudry, B.: From diagnosis to diagnosability: axiomatization, measurement and application. *The Journal of Systems and Software* **65** (2003) 31–50
18. Pickin, S., Jard, C., Heuillard, T., Jezequel, J.M., Desfray, P.: A UML-integrated test description language for component testing. In: UML 2001 Workshop - Practical UML-based Rigorous Development Methods, held jointly with UML 2001 Conference, Toronto, Canada. (2001)
19. ISO/IEC: ISO/IEC Information Technology - Open Systems Interconnection - Conformance Testing Methodology Framework - Part I - General Concepts. ISO/IEC International Standards **9646** (1992)
20. ISO/IEC: ISO/IEC Information Technology - Open Systems Interconnection - Conformance Testing Methodology Framework - Part 3 - The Tree and Tabular Combined Notation (TTCN). ISO/IEC International Standards **9646** (1992)
21. I Schieferdecker and Z R Dai and J Grabowski and A Rennoch: The UML 2.0 Testing Profile and its Relation to TTCN-3. In: Proceedings of the 15th IFIP International Conference on Testing of Communicating Systems (Testcom 2003), Sophia Antipolis, France. (2003)
22. McGregor, J.: The parallel architecture for component testing. *Journal of Object-Oriented Programming* (1997)
23. Ramakrishnan, S., McGregor, J.D.: Extending OCL to support Temporal Operators. In: Workshop on Testing Distributed Component-based Systems held in conjunction with the 21st International Conference on S.E. (1999)
24. Born, M., Schieferdecker, I., Li, M.: UML Framework for Automated Generation of Component-Based Test Systems. In: 1st. Int. Conf. on Software Engineering Applied to Networking and Parallel/Distributed Computing (SNPD'00). (2000)
25. Sheppard, J.W., Simpson, W.: A Systems View of Test Standardization. In: AutoTestCon 1996, IEEE (1996)
26. Sheppard, J., Bartolini, A., Orlidge, L.A.: Standardizing Diagnostic Information Using IEEE AI-ESTATE. In: AutoTestCon 1997, IEEE (1997) 82–87
27. Sheppard, J., Kaufman, M.: Information-based Standards and Diagnostic Component Technology. In: International Workshop on System Test and Diagnosis, IEEE (1999)

Towards a Tool Environment for Model-Based Testing with AsmL

Mike Barnett, Wolfgang Grieskamp, Lev Nachmanson,
Wolfram Schulte, Nikolai Tillmann, and Margus Veanes

Microsoft Research, Foundations of Software Engineering, Redmond, USA
fse@microsoft.com

Abstract. We present work on a tool environment for model-based testing with the Abstract State Machine Language (AsmL). Our environment supports semi-automatic parameter generation, call sequence generation and conformance testing. We outline the usage of the environment by an example, discuss its underlying technologies, and report on some applications conducted in the Microsoft environment.

1 Introduction

Over the last two decades, the area of formal software modeling has been extensively explored, developing various methods, notations and tools. Formal specification languages like VDM, Z, B, CSP, ASM etc. have been developed and applied to numerous problems. Verification technology has had success in certain areas, in particular if based on model checking. However, in spite of promising results, a widely expected break-through of these technologies has not yet appeared.

The goal of our group at Microsoft Research is to bring rigorous, formal modeling to praxis, trying to avoid (suspected) obstacles of earlier approaches to formal modeling. We have developed the Abstract State Machine Language (AsmL), an executable modeling language based on the ASM paradigm [1] and fully integrated into the .NET framework and Microsoft development environment.

One important application we see for AsmL is automated testing. A huge amount of work is spent on testing in Microsoft's and other companies' product cycle today. Models not only enhance understanding what a product is supposed to do and how its architecture is designed, but enable one to semi-automatically derive test scenarios at an early development stage where coding has not yet finished. Given manually or automatically generated test scenarios, formal models can be used to automate the test oracle. A great advantage of model-based testing is seen in its adaptability: during the product cycle, various versions of the product are published at milestones, each of which requires thorough testing. Whereas manual test suites and harnesses are hard to adapt to the variations of the product, a model makes this work easier.

We have developed an integrated tool environment for model-based testing with AsmL. This environment comprehends the following technologies:

- *Parameter generation* for providing method calls with parameter sets;
- *FSM generation* for deriving a finite state machine from a (potentially infinite) abstract state machine;
- *Sequence generation* for deriving test sequences from the FSM;
- *Runtime Verification* for testing whether an implementation performs conforming to the model.

Our environment realizes a semi-automatic approach, requiring a user to annotate models with information for generating parameters and call sequences, and to configure bindings between model and implementation for conformance testing. This annotation process is supported by a GUI. The approach is novel, to the best of our knowledge, in its combination as well as in many of its ingredients. In this paper, we will discuss the environment's methodology and underlying implementation by a walk-through of an example.

2 The Abstract State Machine Language

Space constraints prevent us from giving a systematic introduction to AsmL; instead we rely on the readers' intuitive understanding of the language as used in the examples. AsmL is a fusion of the Abstract State Machine paradigm and the .NET common language runtime type system. From a specification language viewpoint, one finds the usual concepts of earlier specification languages like VDM or Z. The language has sets, finite mappings and other high level data types with convenient and mathematically-oriented notations (e.g., comprehensions). From the .NET integration viewpoint, AsmL has all the ingredients of a .NET language, namely interfaces, structures, classes, enumerations, methods, delegates, properties and events. The close embedding into .NET allows AsmL to interoperate with any other .NET language and the framework: AsmL models can call out into frameworks and AsmL models can be called and referred to from other .NET languages, up to the level that e.g. an AsmL interface (with specification parts) can be implemented by a .NET language, enabling checking that the interface contract is obeyed [3].

The most unique feature of AsmL is its foundation on Abstract State Machines (ASM) [1]. An ASM is a state machine that in each step computes a set of *updates* of the machine's variables. Upon the completion of a step, all updates are "fired" (committed) simultaneously; until that happens, updates are not visible, supporting a side-effect free view on the computation inside a step. The computation of an update set can be complex, and the numbers of updates calculated is not statically bound. Control flow of the ASM is described in AsmL in a programmatic, textual way: there are constructs for parallel composition, sequencing of steps, non-deterministic (more exactly, random) choice, loops, and exceptions. On an exception, all updates are rolled back, enabling atomic transactions to be built from many sub-steps.

AsmL supports meta-modeling which allows a programmatic exploration of the non-determinism in the model and dealing with state as a first-class citizen (i.e., the current state is accessible as a normal value that can be manipulated just as any other

data value). This allows us to realize various state exploration algorithms for AsmL models, including explicit state model-checking and in particular test generation and test evaluation.

AsmL documents are given in XML and/or in Word and can be compiled from Visual Studio .NET or from Word; the AsmL source is embedded in special tags/styles. Conversion between XML and Word (for a well-defined subset of styles) is available. This paper is itself a valid AsmL document; it is fed directly into the AsmL system for executing the formal parts it contains or for working with the AsmL test environment.

3 Example: Web Shop

Throughout this paper, we will use as an example a simplified model of a web shop. Our web shop allows clients to order gifts like flowers or perfume using the common shopping cart metaphor. Real-world details are heavily abstracted in this example to make it comprehensible (we should emphasize at this point that our approach scales to richer examples; see Sect. 0 for applications in the Microsoft environment).

The web shop's items are introduced below:

```
enum Item
  Flowers
  Perfume
const prices as Map of Item to Integer =
  { Flowers -> 30, Perfume -> 20 }
```

A *shopping cart* is represented as a bag (multi-set) of items:

```
type Cart = Bag of Item
```

A client to the web shop is described by the class below. A client has an identifier and a session state, given by its shopping cart. If the client is not in a session the cart is null (The type $T?$ in AsmL denotes a type where null is an allowed value; by default, types in AsmL do not contain null):

```
class Client
  const id as String
  var cart as Cart? = null
  override ToString() as String?
    return id
```

The state of the web shop model is given by a set of clients:

```
var clients as Set of Client = {}
```

We now define the actions of the clients. A client can be constructed in which case he is added to the set `clients`; a client can enter the shop (if he is not in a session), can add an item to his cart (if he is in a session), or remove an item (if he is in a session and the item is on his cart). Finally, a client can checkout, obtaining the bill and ending his session (In this simplified model, the client can *only* leave the shop by paying):

```

class Client
  Client(id as String)
    require not exists client in clients
      where client.id = id
    add me to clients
  EnterShop()
    require cart = null
    cart := Bag of Item()
  AddToCart(item as Item)
    require cart <> null
    cart := cart.Include(item)
  RemoveFromCart(item as Item)
    require cart <> null and then item in cart
    cart := cart.Exclude(item)
  Checkout() as Integer
    require cart <> null and then cart.Size > 0
    var bill as Integer = 0
    step foreach item in cart
      bill := bill + prices(item)
    step
    cart := null
    return bill

```

4 FSM and Sequence Generation

The AsmL tool environment allows generating a finite state machine from models such as that for the web shop. From the FSM, call sequence can be generated using standard techniques [4]. The FSM is generated by exploring the state space of the model in a similar way an explicit-state model-checker works [5]. Starting at the initial state, enabled actions are fired, leading to a set of successor states, from where the exploration is continued. An action hereby is a shared or an instance based method; parameters to this method (including the instance object if necessary) are provided by a configurable parameter generator (see Sect. 0) An action is enabled if the method's precondition (**require**) is true in the current state.

Various ways are available to prune the exploration. Pruning is strictly necessary for infinite models (like the one for the web shop where a client could add items again and again to the cart). But pruning might be also required for large finite models in order to focus on certain test purposes. The AsmL environment provides a collection of different pruning techniques; the most important are:

- *State abstraction*: state abstractions map a concrete state to an abstract state. Exploration stops when a state is reached whose abstract equivalent has already been seen.
- *Filters*: a filter allows excluding certain states from exploration; only those states that pass the filter are considered for continuation during exploration.
- *Model coverage*: a percentage of model branch coverage can be given; exploration stops when this coverage is reached.

We illustrate the FSM generation for the web shop example. First we have to provide suitable definitions for the parameter domains of the actions. The actions of interest here are the client constructor and the instance methods of a client for enter-

ing a shop, adding and removing items, and checking out. The parameters required are identifiers for clients, client objects and items. For the first one we provide a given set of names like a, b, c and so on. For the client object domain it is natural to actually use the model variable `clients` itself: it provides in each state the set of clients created so far. For the items, finally, we use the domain as given by the enumeration. We discuss the configuration of parameter domains in greater detail in the next section. For now it is important to note that the configured parameter domains can depend on the *dynamic* state of the model. Thus, as clients are created, the domain for the instance parameter of client domains, given by the model variable `clients`, grows.

Once we have configured parameter domains, we define the variables and actions of the state machine, and add a so-called abstraction property for pruning the state exploration. The state abstraction properties group the concrete states into equivalence classes; exploration is stopped if we see a concrete state for which an equivalent one has been already seen before.

Finding the right abstraction property is a creative task and requires experience and trial and error. If the purpose of the generated FSM is to create scenarios for adding and removing two different items by just one client the following property does fine:

```
property SomeItemsInCart as Set of (Bag of Item)?
get return
{ (if client.cart <> null then
  client.cart * Bag{Flowers,Perfume}
  else null) | client in clients }
```

This property maps the state space of the model into a set of carts, for each client one cart in the set; it does not distinguish from which client the cart comes. Each cart in turn is pruned to not contain more than one `Flowers` and one `Perfume` item (we use multi-set intersection for this purpose: for example, $\{a, a, b\} * \{a\} = \{a\}$).

Here, we want to further prune to state space by filtering out states with more than one client. We use the following filter:

```
property AtMostOneClient as Boolean
get return Size(clients) <= 1
```

The complete configuration for the web shop is shown in the screenshot in Figure 1. The domains part of the configuration contains annotations of model elements for parameter domains. The state machine part contains annotations for variables and actions of the state machine as well as the abstraction property.

Given this configuration, we generate an FSM as shown in Figure 2. Only one client will be created in this FSM, since our abstraction property does not distinguish from which client a cart comes (and hence if a second client enters the shop, no difference is seen in the abstract state to the first client). In the FSM, S3 is associated with the state where the client's cart is empty, S4 where the client has `Perfume` on his cart, S5 where he has `Flowers` on his cart, and S6 where he has both. Among these states, various transitions exist, adding and removing items.

From an FSM as shown we generate test sequences using the well-known FSM traversal techniques (we use a variation of the transition tour method based on an algorithm from [6]). For the shown FSM we get a single traversal with 19 steps.

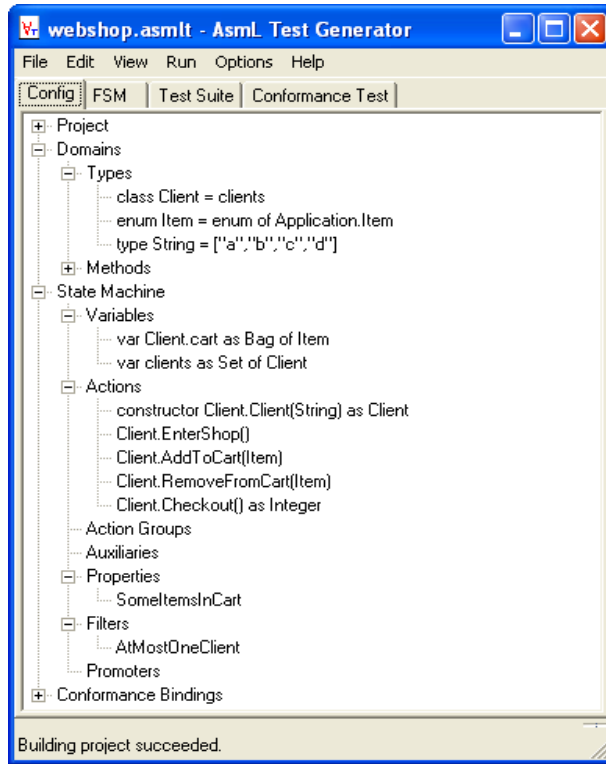


Fig. 1. Configuration

The simple example of the web shop can produce much richer FSMs. The following property allows for more items: each client can buy up to two flowers and two perfume sets:

```
property MoreItemsInCart as Set of (Bag of Item)?
get let maxItems = Bag{
    Flowers, Flowers, Perfume, Perfume}
return { if client.cart <> null
        client.cart * maxItems
        else null
        | client in clients }
```

The FSM generated from this abstraction property and a filter that restricts the number of clients to 4 consists of around 900 relevant transitions (transitions leading to a new state under the abstraction); 6000 transitions have been tried out to find these transitions, and the construction time was around 4 minutes with a maximal memory footprint of 110 MB. Indeed, such an FSM is not feasible to visualize as a whole; however, with the methodology we are proposing one first tries out with a smaller abstract state space to understand the abstraction and then scale up parameters for the actual generated FSM and test suite.

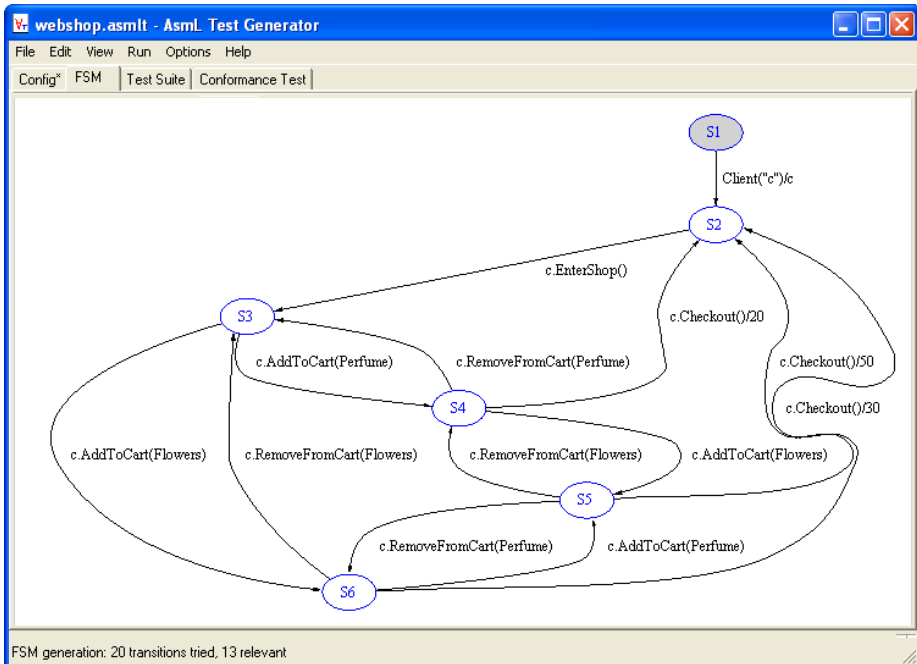


Fig. 2. Generated FSM

5 Parameter Generation

The AsmL test environment uses a parameter generator based on access driven filtering (ADF) which is an enhancement of an existing framework called Korat [7]. ADF can generate values of recursive value types and object graphs. Given a predicate and a domain configuration, ADF generates all non-isomorphic valid inputs whereby an input is regarded as valid if the predicate holds. The domain configuration contains descriptions of finite sets as the domains of basic types, and information about how to generate objects of class types and elements of value types, and imposes bounds on the size of the generated input. The domain configuration classifies the domain of each type into one of the following three categories.

- *Defined Domain*: A defined domain is given by an arbitrary AsmL expression which is evaluated in the scope of the model. It can depend on the dynamic state of the model.
- *Inherited Domain*: An inherited domain is composed of domains as they are for other types. That is, an inherited domain just refers to one or more types, and the union of the domains of these types constitutes the inherited domain. A typical application of inherited domains is abstract types. The domain of those types is naturally the union of the domains of all of its subtypes.

- *Generated Domain*: The domain of a class or value type will be generated by ADF. ADF must be given a domain configuration for each field of the type in one of the three ways described here. A bound on the maximal number of objects/elements of the type in a single input can be imposed. Finally, each field is assigned a cost; all assignments to fields in a given input are summed up to compute the cost of this input. The predicate is configured to have a maximal cost.

ADF exhaustively finds all valid inputs which are within the bounds imposed by the domain configuration. To this end, ADF considers the parameters of the predicate and the fields of generated domains as free variables. ADF executes the predicate with an input which initially only consists of the parameters of the predicate as free variables. Whenever the execution of the predicate accesses a free variable, then ADF will instantiate this variable by choosing an object or value that is allowed by the domain configuration (thus the name access driven filtering). If the bounds imposed by the domain configuration are exceeded or the predicate returns false, then this assignment of the free variables is discarded. Otherwise, if the predicate returns true, any instantiations for the remaining free variables can be chosen to create a valid input. By exhaustively exploring all choices that are possible when instantiating free variables ADF will find all valid inputs within the given bounds.

Two kinds of bounds are imposed on generated domains by the domain configuration.

- A maximal number of objects/elements of a single type: No input will contain more objects/different elements of a single type. This is an effective bound if only a small number of generated domains are involved.
- As an extension to Korat, a maximal accumulated cost along field accesses is maintained. The intuition behind this bound is that one often wants to generate asymmetric inputs which tend to be more complex only in certain areas. In this case, one would assign low costs to fields which lead to the desired complex areas, and high costs to fields which lead to areas which should not be considered. ADF stops the generation of bigger inputs when the accumulated costs exceed a given maximal cost.

As an example of ADF's usage, suppose our web shop allows inputting search queries which are simple boolean expressions over string literals. These can be defined in AsmL as below (where an AsmL structure is a value type which allows recursion):

```
abstract structure Query
structure Literal extends Query
  literal as String
structure Conjunction extends Query
  left as Query
  right as Query
structure Disjunction extends Query
  left as Query
  right as Query
```

Suppose we want to generate those queries as parameter inputs which are in disjunctive normal form. We define a filter predicate as below:

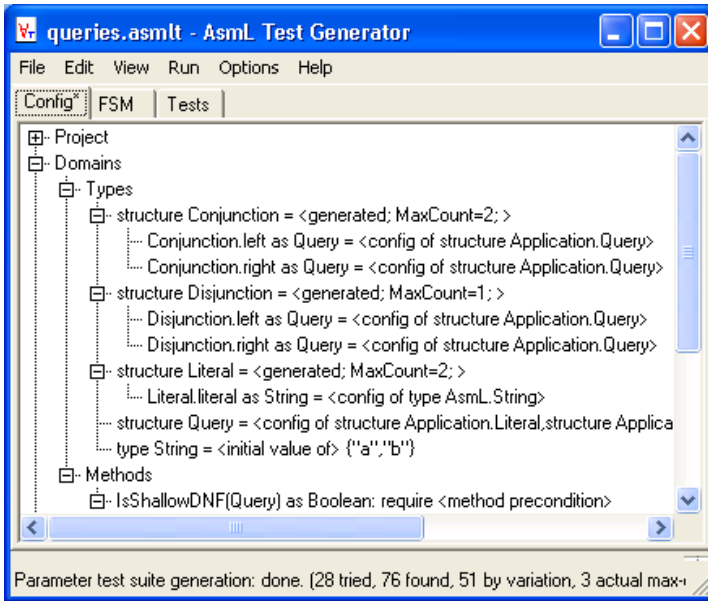


Fig. 3. Domain Definitions

```

IsShallowDNF(query as Query) as Boolean
match query
  q as Conjunction:
    return not q.left is Disjunction and then
           not q.right is Disjunction
  q as Disjunction:
    return IsShallowDNF(q.left) and then
           IsShallowDNF(q.right)
  q as Literal:
    return true

```

Since ADF inductively generates input by instantiating free variables in already generated input, the shallow DNF test as above is sufficient for generating a tree which is in full DNF.

Our test environment allows annotating the configuration for parameter generation with a GUI. Domains can be defined on a per-type base, per-field/parameter base, or per-method base. For the query example, the configuration is given in Figure 3.

The super-type `Query` inherits its domain from the union of the configuration of its sub-types (the full definition is not displayed because of window size). The sub-types are generated using ADF, where the recursive fields point back to the configuration of the `Query` super-type. The recursion causes no problem because the input size is bounded; we allow 2 instances for literals and conjunctions, and 1 instance for disjunction. The run of the parameter generator results in 76 parameter combinations which are of the obvious shape.

6 Conformance Testing

The AsmL test environment allows interactively configuring bindings between a model and an implementation, instrumenting the model as a test oracle. The implementation can be given as any managed .NET assembly, written in any of the .NET languages. A wizard supports the binding of model classes and methods with implementation classes and methods by signature matching.

To enable conformance testing, the implementation assemblies are rewritten on the intermediate code level inserting callbacks for monitored methods to the runtime verification engine. This engine is able to deal with non-determinism in the model by maintaining a set of admissible model behaviors. Each time a monitored method is called in the implementation, its parameters and output result will be propagated to the conformance test manager. On each of the currently possible model states, the according model method will be called. If the method contains nondeterministic behavior, several resulting states can arise. The resulting states of those calls which produce a conformant output constitute the set of next model states. If this set becomes empty, the conformance test fails. In addition to comparing just method results, a predicate which relates the model and implementation state can be employed, which may prune the state-space evolution earlier than by just observing method return values.

The problem of relating object identities is dealt with as follows. A mapping from model to associated implementation objects is maintained. Whenever a monitored implementation returns an object, the according model method's returned object must either map to exactly that object in the mapping, or no entry in the mapping exists, in which case one is created. One can think of this mechanism as letting object identities in the model being distinct logical variables which are "bound" with the associated object identities of the implementation.

7 Discussion and Conclusion

We presented aspects of a first version of an integrated environment for model-based testing with AsmL and illustrated its use by an example. The environment combines and refines the techniques for parameter generation, FSM generation, call sequence generation, and conformance testing in a novel way. We conclude with discussing applications, related work, and future work.

7.1 Applications

Though the AsmL test environment is still in a prototypical stage, it has been applied in several non-trivial projects at Microsoft.

- The parameter generator has been used for testing an implementation of the XPath language. The stateless model of XPath used for that purpose consists of around 33 pages. More than a million tests have been generated, out of which the system identified 120 test cases which already resulted in 90% model code branch cover-

age. To achieve full model code branch coverage the test engineer added 10 tests manually. The recovery of the manual test cases was easy since the system tracks branches which haven't been covered.

- The FSM and sequence generator has been used for testing web-services protocols, among them reliable messaging (RM). The model for RM consists of around 40 pages. The FSM generator produces a machine with around 1500 transitions out of 30000 possible in a couple of minutes, simulating various kinds of wire failure and recovery operations.
- Within the few months of its introduction, the AsmL test environment has gained considerable interest in the model-based testing community at Microsoft. Model-based testing using finite state machine models are in use at Microsoft for quite some time (a couple of hundred people are registered for the internal mailing list, to give an impression). The more powerful approach provided by AsmL is investigated by many of these users, and we expect a couple of new applications in the near future.

7.2 Related Work

Our approach to parameter generation is based on and extends the work found in [7], which is a later branch of the work of the authors of [8]. Whereas the authors use a Java data type to describe what they call the "finitization", we use a richer interactive method for what we call "domain configuration". Other extensions of our approach include a cost function for the generation of recursive domains and detection of isomorphisms for value types.

The conformance testing conducted by our tool environment can be classified as *grey box* testing. Traditional FSM based testing techniques with either Mealy or Moore machines typically amount to *black box* testing, where the actual states of the implementation are unobservable. In contrast, our testing approach *allows* the user to specify *conformance relations* connecting the model state to the state of the implementation, in addition to pure input/output behavior reflected at the API level. In other words, the tool may be used to perform a *limited* form of *white box* testing, where the limitations depend on what part of the implementation state is accessible (if no state is accessible, the tool works as well, but may not be able to detect errors as early as they occur.) This approach is possible due to the intermediate language platform provided by the .NET runtime, which the tool architecture is based on, that allows binary level access to the state of the implementation.

The basic FSM generation algorithm that is implemented in the test tool has been significantly extended since its first description in [5]. Test case generation that is performed on the basis of the generated FSM can be classified as a T-method [13]. We have not considered utilizing more powerful methods, such as U- D- or W-methods [13] used in pure black box testing.

One of the first automated techniques for extracting FSMs from model-based specifications for the purpose of test case generation, introduced in [9], is based on a finite partitioning of the state space of the model using full disjunctive normal forms. While our partition of the state space is similar to that of the DNF approach, the two

approaches are quite different. Most importantly, the DNF approach employs symbolic techniques while we build the FSM by executing the specification. This enables us to support the full spectrum of AsmL, including call-outs from the model into framework code.

In [10] projections on state machines are used to restrict them for a certain test purpose; also filters on states are used. This is related to our pruning technique of state exploration, though we never look at the larger FSM but generate the projected one from the beginning.

In model checking, data abstraction is used to cope with state explosion when the original model M is too large. Data abstraction groups states of M and produces a reduced model M_r which is analogous to the FSM produced in our test environment by using properties. However, whereas in model checking operations need to be lifted to the abstract domain as well, which is the fundamental difficulty there, we still work with the operations on the concrete data, which can be realized using full AsmL. Due to efficiency considerations, the standard data abstraction algorithms of model-checking may yield an *over-approximation* of M_r ; see [11]. In contrast, our approach may yield an *under-approximation* of the true abstraction, in other words some transitions may be missing, but there are no false transitions, which is important for using the FSM for test case generation.

In general, model checking techniques have been considered in the context of ASM based test case generation; in [12] the counter examples of SPIN are considered as test cases generated from a given ASM and a given property. The technique of using a model-checker with negated goal-states, and then letting the model-checker produce a counter-example which can be interpreted as a test sequence to reach this state, has been proposed by many other authors for automatic test generation. We believe this approach is highly restricted, on the one hand by the input language restrictions most model-checkers have to obey, on the other hand because tailored search machines for finding tests can be more efficient. For example, a model checker used to generate tests finds just one test sequence per exploration whereas our approach finds all test sequences in one exploration of the ASM.

Currently our tool supports the Rural Chinese Postman Tour method to traverse the generated FSM. For an efficient implementation of the postman tour the tool uses the algorithm for Maximal Weight Bipartite Matching given in [6]. In general, the test methodology of the tool is an extension of the FSM approach. The bulk of the work in this area has dealt with deterministic FSMs. See [4, 13] for comprehensive surveys and [14] for an overview of the literature. The *Extended* Finite State Machine (EFSM) approach has been introduced mainly to cope with the state explosion problem of the FSM approach. Typically the problem arises when the system to be modeled has variables with values in large, even infinite, domains, for example integers. In an EFSM, such variables are allowed, and the transitions may depend on and update their values; see [15] [16]. In EFSMs, the control part is finite and is separated from the data part, which distinguishes them from ASMs. An interesting problem in our FSM generation algorithm is to fiddle with the properties in order to avoid non-determinism. This problem is related to the stabilization problem of EFSMs [16]. The use of input/output FSMs for fault coverage based test case generation is studied in

[17]. The specification FSMs in [17] are (possibly non-deterministic) Mealy machines.

Conformance testing plays a central role in testing communication protocols where it is important to have a precise model of the observable behavior of the system. This has lead to a testing theory based on labeled transition systems. See an overview of the approach in [18] and an overview of related literature in [19]. Labeled transition systems are in general nondeterministic. In the LTS approach, verification techniques can be used to deal with state explosion and to generate test cases. TGV [20] is an industrial tool that utilizes the LTS approach to generate test cases from SDL specifications. Fault model based FSM testing methodology has been recently considered for labeled transition systems as well [21,31].

There are many different groups doing work related to runtime verification. Perhaps the closest is the JML runtime assertion checking provided for components written in Java [22]. Eiffel [23] also provides for the checking of pre- and post-conditions, but only for components written in Eiffel. There are many similar design-by-contract tools for Java, such as JMSAssert [24], iContract [25], Handshake [26], Jass [27], and JContract [28]. However, all lack any facility for maintaining the state-space separation between the specification and the implementation. More general component-oriented work has been done by Edwards [29] to generate wrapper components for checking pre- and post-conditions, but cannot handle more general synchronization issues that require model programs.

7.3 Future Work

Several extensions of the AsmL test environment are on the way. High priority on our agenda is dealing with non-determinism in the model. Though we can handle non-determinism on the level of runtime verification, the test generator can not deal with it, not at least because its output, sequences, is not a suitable representation. We are looking at two different approaches. One promising approach is *on-the-fly testing* [30], which in our setting amounts to fusing the FSM generation with conformance testing. This approach has the advantage that non-determinism of the model is immediately pruned by the decisions of the implementation. However, in our experience some user groups require the tests as data in their development process. For these applications, we look at generating DAGs (directed acyclic graphs) instead of sequences. Test cases for non-deterministic systems are usually tree structures (as one form of DAG). A further topic of future work is employing symbolic computation by means of constraint resolution, lifting restrictions of our approach implied by computing with ground data.

References

1. Y. Gurevich, "Evolving Algebras 1993: Lipari Guide," in *Specification and Validation Methods*, E. Boerger, Ed.: Oxford University Press, 1995, pp. 9–36.
2. F. o. S. Engineering, "The AsmL Release,"., 2.1.5.9 ed. Redmond: Microsoft Research, 2000–2003.

3. M. Barnett and W. Schulte, "Runtime Verification of .NET Contracts," *The Journal of Systems and Software*, pp. 199–208, 2002.
4. D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines - a survey," *Proceedings of the IEEE*, vol. 84, pp. 1090–1123, 1996.
5. W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes, "Generating Finite State Machines from Abstract State Machines," in *Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2002)*. Rome, Italy: IEEE, 2002, pp. 112–22.
6. K. Mehlhorn and G. Schaefer, "A Heuristic for Dijkstra's Algorithm with Many Targets and Its Use in Weighted Matching Algorithms," *Lecture Notes in Computer Science*, vol. 2161, pp. 242–253, 2001.
7. C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated Testing Based on Java Predicates," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2002)*. Rome, Italy: IEEE, 2002.
8. D. Marinov and S. Khurshid, "TestEra: A Novel Framework for Automated Testing of Java Programs," presented at Proceedings of ASE-2001: The 16th IEEE Conference on Automated Software Engineering, Coronado, CA, USA, 2001.
9. J. Dick and A. Faivre, "Automating the Generation and Sequencing of Test Cases from Model-Based Specifications," in *FME '93: Industrial Strength, Formal Methods*, vol. 670, *Lecture Notes in Computer Science*, J. C. P. Woodcock and P. G. Larsen, Eds.: Springer-Verlag, 1993, pp. 268–284.
10. G. Friedman, A. Hartman, K. Nagin, and T. Shiran, "Projected State Machine Coverage for Software Testing," in *Proceedings of the ACM SIGSOFT 2002 International Symposium on Software Testing and Analysis (ISSTA-02)*, vol. 27, 4, *SOFTWARE ENGINEERING NOTES*, P. G. Frankl, Ed. New York: ACM Press, 2002, pp. 134–143.
11. E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*: MIT Press, 1999.
12. Gargantini, E. Riccobene, and S. Rinzivillo, "Using Spin to Generate tests from ASM Specifications," presented at Proc. Abstract State Machines 2003, 2003.
13. D. P. Sidhu and T.-K. Leung, "Formal Methods for Protocol Testing: A Detailed Study," *IEEE Transactions on Software Engineering*, vol. 15, pp. 413–426, 1989.
14. Petrenko, "Fault Model-Driven Test Derivation from Finite State Models: Annotated Bibliography," *Lecture Notes in Computer Science*, vol. 2067, pp. 196–?, 2001.
15. Bourhr, R. Dssouli, and E. M. Aboulhamid, "Automatic test generation for EFSM-based systems," Publication departementale 1043, Departement IRO, Universite de Montreal August 1996 1996.
16. K.-T. Cheng and A. S. Krishnakumar, "Automatic generation of functional vectors using the extended finite state machine model," *ACM Transactions on Design Automation of Electronic Systems*, vol. 1, pp. 57–79, 1996.
17. Petrenko, N. Yevtushenko, and G. v. Bochmann., "Testing deterministic implementations from nondeterministic FSM specifications," presented at Proc. IFIP TC6 9th International Workshop on Testing of Communicating Systems, Darmstadt, Germany, 1996.
18. J. Tretmans and A. Belinfante, "Automatic testing with formal methods," presented at EuroSTAR'99: 7th European Int. Conference on Software Testing, Analysis & Review, Barcelona, Spain, 1999.
19. E. Brinksma and J. Tretmans, "Testing Transition Systems: An Annotated Bibliography," *Lecture Notes in Computer Science*, vol. 2067, pp. 187–?, 2001.
20. J. C. Fernandez, C. Jard, T. Jérón, and C. Viho, "An experiment in automatic generation of test suites for protocols with verification technology," *Science of Computer Programming - Special Issue on COST247, Verification and Validation Methods for Formal Descriptions*, vol. 29, pp. 123–146, 1997.

21. Petrenko, N. Yevtushenko, and J. L. Huo, "Testing Transition Systems with Input and Output Testers," presented at Proc. IFIP TC6 9th International Workshop on Testing of Communicating Systems, Sophia Antipolis, France, 2003.
22. G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok, "How the design of JML accomodates both runtime assertion checking and formal verification," Technical Report TR #03-04 Department of Computer Science, Iowa State University March 2003 2003.
23. B. Meyer, *Eiffel: The Language*. New York, NY: Prentice Hall, 1992.
24. M. M. Systems, "JMSAssert,"., 2002.
25. R. Kramer, "iContract -- the Java Designs by Contract tool. In Proc. Technology of Object-Oriented Languages and Systems," presented at Proc. Technology of Object-Oriented Languages and Systems, TOOLS 26, Santa Barbara, CA, 1998.
26. Duncan and U. Hoelzle, "Adding Contracts to Java with Handshake," University of California, Santa Barbara. Computer Science., Technical Report TRCS98-32, December 9, 1998 1998.
27. D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim, "Jass - Java with assertions," in Workshop on Runtime Verification held in conjunction with the 13th onference on Computer Aided Verification, CAV'01, 2001.
28. P. Corporation, "Using design by contract to automate Java software and component testing," , February 2003 2003.
29. S. H. Edwards, "A framework for practical, automated black-box testing of component-based software," *Software Testing, Verification and Reliability*, vol. 11, 2001.
30. R. G. d. Vries and J. Tretmans, "On-the-fly conformance testing using Spin," *Software Tools for Technology Transfer*, vol. 2, pp. 382–393, 2000.
31. Q.M. Tan and A. Petrenko, "Test Generation for Specifications Modeled by Input/Output Automata", in Proc. IFIP 11th Int. Conf. on Testing of Communicating Systems, IWTC'S'98, Russia, 1998.