Lecture Notes in Computer Science 3997

Wolfgang Grieskamp Carsten Weise (Eds.)

Formal Approaches to Software Testing

5th International Workshop, FATES 2005
Edinburgh, UK, July 11, 2005
Revised Selected Papers

Springer

Volume Editors

Wolfgang Grieskamp
Microsoft Research
One Microsoft Way, Redmond, WA 98052, USA
E-mail: wrwg@microsoft.com

Carsten Weise
Ericsson Deutschland GmbH
52134 Herzogenrath, Germany
E-mail: Carsten.Weise@ericsson.com

Library of Congress Control Number: 2006926507

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

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-34454-3 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-34454-4 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media

springer.com

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11759744 06/3142 5 4 3 2 1 0

Foreword

Software testing is one of the most cost-intensive tasks in the modern software production process. The application of formal approaches to the testing process has gained steady attention in recent years. Effective and efficient test cases may be generated automatically from formal system models and specifications or be developed based on a formal analysis of the system. Formal approaches to testing use techniques from areas like theorem proving, model checking, constraint resolution, program analysis, abstract interpretation, Markov chains, and various others. These techniques are combined with traditional approaches to testing.

The workshop on Formal Approaches to Testing of Software (FATES) selected a number of high-quality submissions out of these fields for presentation at the workshop as well as inclusion in the workshop proceedings. The contributions show the state of the art in the application of formal methods in the testing process. The workshop had 38 submissions, of which 14 were accepted (13 full papers and 1 work-in-progress). In all, this proceedings volume collects the work of 38 authors from 12 countries. Each paper underwent 3 reviews, done by 36 reviewers.

This has been the fifth successful workshop in the history of the FATES workshops. Previous workshops were held in Aaalborg (Denmark) in 2001 and in Brno (Czech Republic) in 2002 as satellites of CONCUR, and in Montréal (Canada) in 2003 and Linz (Austra) in 2004 as satellites of the IEEE/ACM Conference on Automated Software Engineering (ASE).

We would like to express our gratitude to all authors for their valuable contributions and to the Workshop Organizing Committee of the CAV 2005 conference. In addition, we would like to thank all members of the FATES Program Committee and the additional reviewers, who were given the important and tedious task of reviewing many papers. The individuals who contributed to this effort are listed on the following pages.

Redmond and Aachen
March 2005

Wolfgang Grieskamp
Carsten Weise

Organization

Program Chair

Wolfgang Grieskamp, Microsoft Research, Redmond, USA
Carsten Weise, Ericsson Deutschland GmbH,
Research and Development, Aachen, Germany

Program Committee

Simon Burton, DaimlerChrysler AG, Germany
Rachel Cardell-Oliver, University of Western Australia, Australia
Marie-Claude Gaudel, Université de Paris-Sud, France
Jens Grabowski, University of Göttingen, Germany
Klaus Havelund, Kestrel Technology, NASA Ames Research Center, USA
Robert M. Hierons, Brunel University, UK
Thierry Jéron, IRISA/INRIA, France
Victor Kuliamin, Institute for System Programming,
Russian Academy of Science, Russia
David Lee, Ohio State University, USA
Brian Nielsen, Aalborg University, Denmark
Manuel Núñez, Universidad Complutense de Madrid, Spain
Jeff Offutt, George Mason University, USA
Doron Peled, University of Warwick, UK
Alexandre Petrenko, Computer Research Institute of Montréal, Canada
John Rushby, SRI International Computer Science Laboratory, USA
Ina Schieferdecker, Fraunhofer FOKUS, Berlin, Germany
Jan Tretmans, Radboud University Nijmegen, The Netherlands
Andreas Ulrich, Siemens AG, Corporate Technology, Munich, Germany
Mark Utting, Waikato University, New Zealand
Clay Williams, IBM T. J. Watson Research Center, USA
Burkhart Wolff, ETH-Zürich, Switzerland
Jianping Wu, Tsinghua University, Beijing, China

List of Reviewers

Alexandre Petrenko
Andreas Ulrich
Arne Skou
Blaise Genest
Brian Nielsen
Burkhart Wolff
Caixia Chi
Carsten Weise
Clay Williams
Dean Rosenzweig
Doron A. Peled
Edith Werner
Guoqiang Shu
Helmut Neukirchen
Hongyang Qu

Ismael Rodríguez
Jeff Offutt
Jens Grabowski
Jiale Huo
Keqin Li
Klaus Havelund
Machiel van der Bijl
Manuel Núñez
Margus Veanes
Marie-Claude Gaudel
Marielle Stoelinga
Marius Mikucionis
Mark Utting
Natalia López
Rachel Cardell-Oliver
Rene de Vries
Robert M. Hierons
Simon Burton
Tim Willemse
Victor Kuliamin
Wolfgang Grieskamp

List of Contributors

Achim D. Brucker
ETH Zürich
Zürich, Switzerland
brucker@inf.ethz.ch

Agnès Arnould
Université de Poitiers
Futuroscope Cedex, France
arnould@sic.univ-poitiers.fr

Alexandre Petrenko
CRIM, Centre de Recherche
Informatique de Montréal
Montréal, Canada
petrenko@crim.ca

Andreas Blass
University of Michigan
Ann Arbor, MI, USA
ablass@umich.edu

Antti Huima
Conformiq Software Ltd.
Espoo, Finland
antti.huima@conformiq.com

Antti Kervinen
Tampere University of Technology
Tampere, Finland
Antti.Kervinen@tut.fi

Axel Rennoch
Fraunhofer FOKUS,
Berlin, Germany
axel.rennoch@fokus.fhg.de

Bruno Marre
CEA/DRT/LIST/DTSI/SLA Saclay
Gif sur Yvette Cedex, France
bruno.marre@cea.fr

Burkhart Wolff
ETH Zürich
Zürich, Switzerland
bwolff@inf.ethz.ch

Clément Boin
Université d'Évry-Val d'Essonne
Évry Cedex, France
cboin@lami.univ-evry.fr

Doron Peled
University of Warwick
Coventry, UK
doron@dcs.warwick.ac.uk

Gaoyan Xie
University of Massachusetts
Dartmouth, USA
gxie@umassd.edu

Guven Bolukbasi
Koç University
Istanbul, Turkey
gbolukbasi@ku.edu.tr

Hongyang Qu
University of Warwick
Coventry, UK
hongyang@cmi.univ-mrs.fr

Ismael Rodríguez
Universidad Complutense de Madrid
Madrid, Spain
isrodrig@sip.ucm.es

Jaco van de Pol
Centrum voor Wiskunde
en Informatica
Amsterdam, The Netherlands
Jaco.van.de.Pol@cwi.nl

Johannes Mayer
University of Ulm
Ulm, Germany
johannes.mayer@uni-ulm.de

Lev Nachmanson
Microsoft Research
Redmond, WA, USA
levnach@microsoft.com

M. Erkan Keremoglu
Koç University
Istanbul, Turkey
mkeremoglu@ku.edu.tr

Manuel Núñez
Universidad Complutense de Madrid
Madrid, Spain
mn@sip.ucm.es

Marc Aiguier
Université d'Évry-Val d'Essonne
Évry Cedex, France
aiguier@lami.univ-evry.fr

Marcin Jurdziński
University of Warwick
Coventry, UK
m.j.u@dcs.warwick.ac.uk

Margus Veanes
Microsoft Research
Redmond, WA, USA
margus@microsoft.com

Mika Katara
Tampere University of Technology
Tampere, Finland
Mika.Katara@tut.fi

Mika Maunumaa
Tampere University of Technology
Tampere, Finland
Mika.Maunumaa@tut.fi

Natalia Ioustinova
Centrum voor Wiskunde en
Informatica
Amsterdam, The Netherlands
Natalia.Ioustinova@cwi.nl

Natalia Sidorova
Eindhoven University of Technology
Eindhoven, The Netherlands
n.sidorova@tue.nl

Nina Yevtushenko
Tomsk State University,
Russia
yevtushenko.RFF@elefot.tsu.ru

Pascale Le Gall
Université d'Évry-Val d'Essonne,
Évry Cedex, France
legall@lami.univ-evry.fr

Serdar Tasiran
Koç University
Istanbul, Turkey
stasiran@ku.edu.tr

Sergey Zelenov
Russian Academy of Sciences
Moscow, Russia
zelenov@ispras.ru

Stefan Blom
University of Innsbruck,
Innsbruck, Austria
Stefan.Blom@uibk.ac.at

Sophia Zelenova
Russian Academy of Sciences
Moscow, Russia
sophia@ispras.ru

Tayfun Elmas
Koç University
Istanbul, Turkey
telmas@ku.edu.tr

Tuula Pääkkönen
Nokia Technology Platforms
Tampere, Finland

Yuri Gurevich
Microsoft Research
Redmond, WA, USA
gurevich@microsoft.com

Zhe Dang
Washington State University
Pullman, USA
zdang@eecs.wsu.edu

Table of Contents

Proceedings FATES 2005

Simulated Time for Testing Railway Interlockings with TTCN-3*

Stefan Blom[1], Natalia Ioustinova[2], Jaco van de Pol[2,4],
Axel Rennoch[3], and Natalia Sidorova[4]

[1] Institute of Computer Science, University of Innsbruck, 6020 Innsbruck, Austria
Stefan.Blom@uibk.ac.at

[2] Centrum voor Wiskunde en Informatica, SEN2, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Natalia.Ioustinova@cwi.nl, Jaco.van.de.Pol@cwi.nl

[3] Fraunhofer FOKUS, Kaiserin-Augusta-Alee 31, D-10589, Berlin, Germany
axel.rennoch@fokus.fhg.de

[4] Eindhoven University of Technology, Dept. of Math. and Computer Science,
P.O. Box 513, 5612 MB Eindhoven, The Netherlands
n.sidorova@tue.nl

Abstract. Railway control systems are timed and safety-critical. Testing these systems is a key issue. Prior to system testing, the software of a railway control system is tested separately from the hardware. Here we show that real time and scaled time semantics are inefficient for testing this software. We provide a time semantics with *simulated time* and show that this semantics is more suitable for testing of software of railway control systems.

TTCN-3 is a standardized language for specifying and executing test suites. It supports real time and scaled time but not simulated time. We provide a solution that allows simulated time testing with TTCN-3. Our solution is based on Dijkstra's distributed termination detection algorithm. The solution is implemented and can be reused for simulated time testing of other systems with similar characteristics.

Keywords: testing, real time, discrete time, scaled time, simulated time, interlockings, TTCN-3.

1 Introduction

Railway control systems are safety-critical and therefore we have to ensure that they are designed and implemented correctly. The interlocking is a layer of railway control systems that guarantees safety. It allows to execute commands given by a user only if they are safe; unsafe commands are rejected. Interlockings also react in dangerous situations that can lead to derailments and collisions. In this

* This work is done within the project "TTMedal. Test and Testing Methodologies for Advanced Languages (TT-Medal)" [14].

W. Grieskamp and C. Weise (Eds.): FATES 2005, LNCS 3997, pp. 1–15, 2006.

paper we propose a *testing method* for interlockings and indicate the characteristics of systems for which this method will be suitable as well.

The software part of the interlocking is a program that consists of a large number of guarded assignments. The program defines a *control cycle* that is repeated by the system. The control cycle consists of two phases: an active phase and an idle phase. The *active phase* starts with reading the inputs, then proceeds by evaluating the guards and by computing new output values, and finally issues outputs. After the active phase, the system becomes idle for the rest of the control cycle. The point of the control cycle where the idle phase starts is further referred to as an *idleness point*. The total time of the active and the idle phases of the control cycle is fixed. Although the environment of the system changes continuously, the system sees only snapshots of the environment made at the beginning of each control cycle. Thus the environment is discrete from the system's point of view. The system is *timed*, delays are used to guarantee safety. To keep the logic of the system simple and safe, the delays are chosen based on the worst case assumptions about the environment behavior. In this paper, we try and choose a time semantics that is the most suitable and efficient to test this kind of systems.

Real time is usually considered to be the most adequate choice when testing timed systems. In real time, the system clock is driven by a physical clock. In the interlocking, the length of the active phase of the control cycle is much smaller than the length of the control cycle. Therefore, the total time spent by the system on being idle is much larger than the time spent on real computations. Hence, with testing interlockings in real time, we waste a large amount of time on idle phases.

When testing interlockings, we actually test a software system, so we have the control over the timing of test executions. The most simple, naïve solution is to test the system using *scaled time*. Scaled time is calculated as initial time plus the product of a time factor and a difference between the current physical time and the initial moment. The larger the factor is the faster we can execute tests. Choosing the time factor is however not as simple as it seems. The time factor must be small enough to make the longest active phase fit into the scaled control cycle. Hence, we have to determine the largest possible time factor that still satisfies this condition. Determining the largest time factor is difficult, time-consuming and potentially error-prone. Any simple change in the system or in the test suite implies that the factor has to be determined again.

Even if we have found the time factor, it still would not be optimal for testing. The time spent on computations differs from cycle to cycle. If computations in one control cycle take ten times as much time as computations in the other ten cycles, the total time spent by the system on being idle is still much larger than the total computation time. Hence, testing with scaled time is not the best choice for this kind of systems.

In this paper, we propose a solution based on *simulated time* where the system clock is a discrete logical clock. Simulated time is based on the assumption that the time spent by the system on computations is negligible compared to the duration of the external events. Therefore, the computations are considered to be instantaneous and time progresses only when the system is idle.

The reasons why simulated time is adequate for testing this kind of systems are the following: The length of the control cycle is fixed by the design of the system. The environmental changes are seen by the system as snapshots made at the beginning of each control cycle. This provides natural discretization of the system behavior. Interlockings are designed in such a way that the duration of the control cycle is much smaller than the minimal time within which the system must react on the changes in the environment. Therefore, we may safely use simulated time for testing this kind of systems. In general, simulated time can be seen as scaled time with a dynamic time factor that is determined automatically. Since the factor is dynamic, the approach is efficient in case of varying computation times and allows adequate simulation of the environment in case the system cannot be tested in field.

We have chosen TTCN-3 to implement our solution for testing interlockings. TTCN-3 is a language with the syntax and the operational semantics standardized by ETSI [8, 2, 3]. TTCN-3 was originally developed for real-time testing of telecommunication systems. A TTCN-3 test executable has predefined standard interfaces [4, 5] that allows to offer TTCN-3 solutions that do not depend on the implementation details of a system under test (SUT). Therefore, applying TTCN-3 to domains other than telecommunication systems is potentially beneficial. Implementing simulated time for existing TTCN-3 interfaces is however not straightforward.

In simulated time, a test system and an SUT should agree on simulated time. To guarantee this, we provide a mechanism that detects an idleness point of an SUT together with a test system for each control cycle and then synchronizes them on time progression. A TTCN-3 test system and an SUT usually consist of several concurrent components, so we extend a distributed termination detection algorithm [1] to decide on idleness of all components and to synchronize them on time progression. Our implementation consists of a TTCN-3 module and Java classes for simulated time. The TTCN-3 module supports simulated time within the TTCN-3 executable entity. The Java classes provide implementation of simulated time for platform and system adapters. The solution is general and can be used to test systems other than interlockings.

The rest of the paper is organized as follows: Section 2 provides a brief survey on a general structure of a TTCN-3 test system [4]. In Section 3 we describe particularities of railway control systems and interlockings. In Section4 we define a time semantics for testing interlockings. In Section 5, we present the implementation of simulated time for TTCN-3 test systems. We conclude with Section 6 where we discuss the limitations of our solution, propose possible ways to resolve them and outline future work.

2 TTCN-3 Test Systems

TTCN-3 is intended for specification of (abstract) test suites [8]. The specifications can be generated automatically or developed manually. A specification of a test suite is a TTCN-3 *module* which possibly imports some other modules.

Modules are the TTCN-3 building blocks which can be parsed and compiled autonomously. A module consists of two parts: a definition part and a control part. The first one specifies test cases. The second one defines the order in which these test cases should be executed.

A test suite is executed by a TTCN-3 test system whose general structure is defined in [4]. Fig. 1 illustrates this structure. The Test Management (TM) entity controls the order of execution of test cases and logs test events. Typically, this entity also implements the user interface of the test system. The TTCN-3 executable (TE) entity actually executes or interprets a test suit. The SUT adapter (SA) implements communication between a TTCN-3 test system and an SUT. It adapts message- and procedure-based communication of the TTCN-3 test system to the particular execution platform of the test system. The SA entity also propagates messages and calls from the TE entity to the SUT and notifies the TE about messages and calls from the SUT. The platform adapter (PA) realizes platform-dependant issues like external functions and time.

The TE entity executes TTCN-3 modules. A call of a test case can be seen as an invocation of an independent program. Starting a test case leads to creating a *configuration*. A configuration consists of several test components running in parallel and communicating with each other and with an SUT by *message passing* or by *procedure calls*. The first test component created at the starting point of a test case execution is the main test component (MTC).

For communication purposes, a test component owns a set of ports. Each port has **in** and **out** directions. Infinite FIFO queues are used to represent **in** directions; **out** directions are linked directly to the communication partners. A configuration can be changed dynamically by performing configuration operations CREATE, CONNECT, MAP, START and STOP that allow to create a test component, to map and connect its ports to the ports of other components, to start the component with a certain behavior and finally to stop it. The behavior of a test component is defined by a function given as a reference to the START operation. All components and ports are implicitly destroyed at the termination of each test case, so each test case will completely create its required configuration of components and connections when its execution starts.

To specify time delays, TTCN-3 supports a timer mechanism. Timers are local, namely each timer belongs to a certain test component. For each test

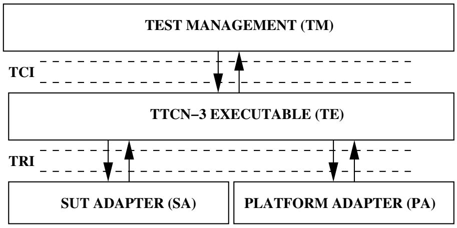


Fig. 1. General structure of a TTCN-3 test system

component, there exists a timeout list. A test component can start a timer for a certain duration by operation START, stop a timer by operation STOP, check whether a timer is running by operation RUNNING, read the elapsed time of a running timer by operation READ and consume timeouts from the timeout list by operation TIMEOUT. A timer can be *active* or *inactive*. An active timer runs from 0 up to the specified duration. When the specified duration is reached, a timer expires, i.e. it adds a timeout to the timeout list of the test component and becomes inactive. Operation TIMEOUT allows a test component to consume a timeout message from its list.

An implementation of timers is platform-dependent, so the timer instances created in the TE and operations on them are implemented by the PA entity. Timers are distinguished by unique timer identifiers (TID). The runtime interface [4] (TRI) allows the TE entity to invoke external functions and the operations on timers implemented by the PA entity. For invocations of some TTCN-3 operations, there exists a direct correlation to invocations of TRI operations. TTCN-3 timer operations START, STOP, READ, RUNNING are realized by TRI operations TRISTARTTIMER, TRISTOPTIMER, TRIREADTIMER, TRITIMERRUNNING respectively. These operations are invoked by the TE entity and performed by the PA entity.

If the TE invokes TRISTARTTIMER, the PA starts the indicated timer with the specified duration. If the TE invokes TRISTOPTIMER, the PA stops the timer. If the TE calls TRIREADTIMER, the PA returns the time elapsed from the moment of starting the timer. In case the timer has not been started or already expired, the PA returns zero. If the TE calls TRITIMERRUNNING, the PA replies whether the timer is active or not.

The PA is responsible for expiring the timers. If an active timer reaches its specified duration, the PA deactivates the timer and notifies the TE about the expiration by calling TRI operation TRITIMEOUT. On the invocation of this operation, the TE entity adds the timeout to the timeout list of the corresponding test component. When starting, stopping or expiring a timer whose timeout is still in the timeout list, the TE removes the timeout message from the timeout list.

In the next section, we give a short overview of railway control systems and describe the control cycle typical for railway interlockings.

3 Testing Railway Interlockings

Railway control systems consist of three layers: infrastructure, logistic, and interlocking. The infrastructure represents a railway yard that basically consists of a collection of linked railway tracks supplied with such features as signals, points and level crossings. The logistic layer is responsible for the interface with human experts, who give control instructions for the railway yard to guide trains. The interlocking guarantees that the execution of these instructions does not cause train collisions or derailments. Thus it is responsible for the safety of the railway system. If the interlocking considers a command as unsafe, the execution of the command is postponed until the command can be safely executed or discarded. Since the interlocking is the most safety-critical layer of the railway control system, we further concentrate on this layer.

Here we consider interlocking systems based on Vital Processor Interlocking (VPI) that is used nowadays in Australia, some Asian countries, Italy, the Netherlands, Spain and the USA [11]. A VPI is implemented as a machine which executes hardware checks and a program consisting of a large number of guarded assignments. The assignments reflect dependencies between various objects of a specific railway yard like points, signals, level crossings and delays on electrical devices and ensure the safety of the railway system. An example of a VPI specification can be found in [15]. In the TTMedal project [14], we develop an approach to testing VPI software with TTCN-3. This work is done in cooperation with engineers of ProRail who take care of capacity, reliability and safety on Dutch railways. They have formulated general safety requirements for VPIs. We use these requirements to develop a TTCN-3 test system for VPIs.

The VPI program has several read-only input variables, auxiliary variables used for computations and several writable variables that correspond to the outputs of the program. The program specifies a *control cycle* that is repeated with a fixed period by the hardware. The control cycle consists of two phases: an active phase and an idle phase. The active phase starts with reading new values for input variables. The infrastructure and the logistic layer determine the values of the input variables. After the values are latched by the program, it uses them to compute new values for internal variables and finally decides on new outputs. The values of the output variables are transmitted to the infrastructure and to the logistic, where they are used to manage signals, points, level crossings and trains. Here we assume that the infrastructure always follows the commands of the interlocking. The rest of the control cycle the system stays idle.

The duration of the control cycle is fixed. Delays are used to ensure the safety of the system. A lot of safety requirements to VPIs are timed. They describe dependencies between infrastructure objects in a period of time, e.g. "when freed, a train track must remain unoccupied for 120 seconds". VPIs control infrastructure objects. The objects of the infrastructure are represented in the VPI program by input and output variables. Thus the requirements defined in terms of infrastructure objects can be easily reformulated in terms of input and output variables of the VPI program. Hence VPIs are *time-critical systems.* Further we are going to propose a time semantics suitable for testing VPI software.

4 Time Semantics for Testing Interlockings

Originally TTCN-3 was developed for the real time testing of telecommunication systems. We use it here for testing VPIs. When testing VPI software in real time, we waste time on idle phases of each control cycle. Imagine that we have to execute 1000 tests of 6 minutes each. Executing all of them will require thus 100 hours. Suppose that the control cycle of VPI is repeated each second and that the active phase takes in average 0.2 seconds. Then we will lose 80 hours on idle phases.

We are testing VPI software separately from hardware. That gives us the control over the timing of test execution, so we could try to solve the problem by using scaled time. For testing with scaled time, we have to determine a time factor. Scaled time is calculated as initial time plus physical time that has passed from the initial moment multiplied by the time factor. When testing VPI software, we can scale *only the idle phase* of the control cycle. Time spent on active phases will still be determined by a hardware running the VPI program, so active phases cannot be scaled. Therefore, we have to choose a safe time factor so that the longest active phase still fits into the scaled control cycle.

Determining a safe time factor, we have to take into account not only the longest active phase of the VPI program but also the longest active phase of the test system. Therefore, determining a time factor is difficult, time consuming and potentially error-prone. Minor changes made in the program or in the test suite can lead to a change of the duration of active phases. Even if we determined a time factor, this time factor is still not optimal. The duration of the active phase of the VPI program together with a test system can differ from one control cycle to another. That means that we still lose time on idle phases of control cycles with a short active phase. Scaled time is not optimal for testing interlockings.

In this section we try and determine which time semantics is the most suitable for testing VPI software.

The first choice to be made is between dense and discrete time. It is normally assumed that real-time systems operate in “real”, continuous time (though some physicists contest against the statement that the changes of a system state may occur at any real-numbered time point). However, a less expensive, discrete time solution is for many systems as good as dense time in the modelling sense, and better than the dense one when testing and verification are concerned [10]. The duration of the control cycle of VPIs is fixed. The program sees only snapshots of the environment at the beginning of each control cycle, meaning the program observes the environment as a *discrete* system. Therefore, the choice for discrete time is obvious.

Often, it has been argued that models where any action takes some non-zero time allow more faithful descriptions. However, VPI software is designed in such a way that an active phase always fits into the control cycle. The duration of a control cycle is smaller than the time period within which the system must react on the environmental changes. In sequel, the actual duration of the active phase is *negligible* compared to the duration of the control cycle and to the reaction time of the system. Therefore, we can treat the active phase as instantaneous.

Time constraints in a VPI program are expressed by time delays that are much longer than the duration of the control cycle. Together with the negligible duration of an active phase, that leads us to the conclusion that we may safely use a logical clock instead of a physical one, namely, we may use *simulated time.* In simulated time, the time progress has the least priority in a system, and time may progress only if the system is *idle*. This property is known as *minimal delay* or *maximal progress* [12]. We refer to the time progress action as `tick` and to the period of time between two `tick`s as a time slice. Further we define the notion of idleness more formally.

Here we consider closed systems consisting of multiple components. Timers are used to express time constraints of the system. We say that a *component* is *idle* iff it cannot proceed by performing computations or by receiving messages. As a consequence, all **in** ports of an idle component are empty and the timeout list of the component is empty as well. Otherwise, the component could still proceed by receiving a message or by consuming a timeout. Further we refer to the idleness of a single component as *local idleness.* We say that a *system* is *idle* iff all components of the system are idle and none of the active timers can expire in the current time slice. We refer to the idleness of the whole system as *global* idleness. If the system is globally idle, the time progresses by action `tick` that increases the elapsed time of active timers by one. If a timer has reached its specified duration, it expires within the current time slice. Timers ready to expire within the same time slice expire in an arbitrary order.

In the next section, we provide a TTCN-3 solution for testing with simulated time.

5 Simulated Time in TTCN-3

TTCN-3 is developed for real time testing, simulated time is not included as an option of a TTCN-3 test system. Our goal is to implement simulated time within the existing structure of a TTCN-3 test system using only standard TRI interface and without introducing any changes into the syntax and the semantics of the TTCN-3 language. Here, we consider a closed system formed by an SUT and a TTCN-3 test system. In simulated time, we have to keep time of all system components synchronized. Therefore, we should provide a mechanism that allows to detect the idleness point of the system in each control cycle and to implement `tick`-steps.

An SUT is idle if it cannot progress further by performing internal computations or by receiving input messages from the test system, i.e. all its **in** ports have to be empty. When doing black-box testing, we do not have control over the computations of an SUT and we cannot observe its internal FIFO queues. Therefore, we make two assumption about an SUT: an SUT supports an interface notifying us of its status (active or idle); an SUT supports an interface for time progression. These are reasonable assumptions when interlockings are concerned. Interlockings have the control cycle with an explicit input/output structure, thus extending VPI software with such interfaces is straightforward.

A TTCN-3 test system is idle if all its entities are idle. The TE entity is idle if all the test components are idle, i.e. they cannot progress further by receiving new messages or by performing computations, meaning, the timeout lists are empty and the channels are empty as well. The PA is idle if it is not performing any external function and there are no timers that have reached their specified duration but not expired yet. The SA cares for the communication with the SUT, so we use it to decide on the idleness of the SUT.

5.1 Distributed Termination Detection Algorithm of Dijkstra

To decide on the global idleness of the system, we employ the well-known distributed termination detection algorithm of Dijkstra [1]. The algorithm allows to decide on the termination of a system of N components. Each component has a unique identity that is a natural number from 0 to $N - 1$. The algorithm differentiates two kinds of messages: (i) *basic* messages exchanged by the components; (ii) termination detection messages. The main assumption important for the correctness of the algorithm is that communication is reliable, meaning, no message is lost.

Each component has a status that is either active or idle. Active components can send messages, idle components are waiting. An idle component can become active only if it gets a basic message. An active component can always become idle. The system is terminated only if all components have the idle status and all channels are empty. The Dijkstra's algorithm allows one of the components, for example the 0-component, to detect whether termination has been reached.

We cannot decide on termination only by looking at the status of the components. The idle status of the components is necessary but not sufficient in this case. The status of a component changes from idle to active only by receiving a basic message, so we have to keep the track of all the messages in the network. Each component has a local message counter. A component decreases its counter when it receives a basic message. When a component sends a basic message, it increases its message counter. Moreover, each component has a local flag. The flag is initially *false*, and it turns *true* only when the component receives a basic message.

The components are connected into a ring that is used to transmit the termination message that is referred to as a *token.* The termination token consists of a global message counter and a global flag. The 0-component initiates a termination detection by sending a termination token with the counter equal to 0 and the flag equal to *false* to the next component in the ring. The 0-component expects that no messages are pending in the network and none of the components has the active status, which is to be checked by passing the token along the ring.

If the next component has the active status, it keeps the token until the status of the component becomes idle. If the component has the idle status it modifies the token by adding its local message counter to the global message counter. If the value of the local flag of the component is *true*, the component propagates the flag by changing the global flag to *true*, meaning, that maybe one of the system components is still active. Then the component forwards the token to the next component along the ring. After forwarding the token, the component changes its local flag to *false*, meaning that the token already got the up-to-date information about this component. The termination is detected by the 0-component only if the component gets back the token with the global flag equal to *false* and the sum of the global message counter with the local message counter of the 0-component is zero. In this case the 0-component can be sure that all other components have the idle status and there are no messages pending in

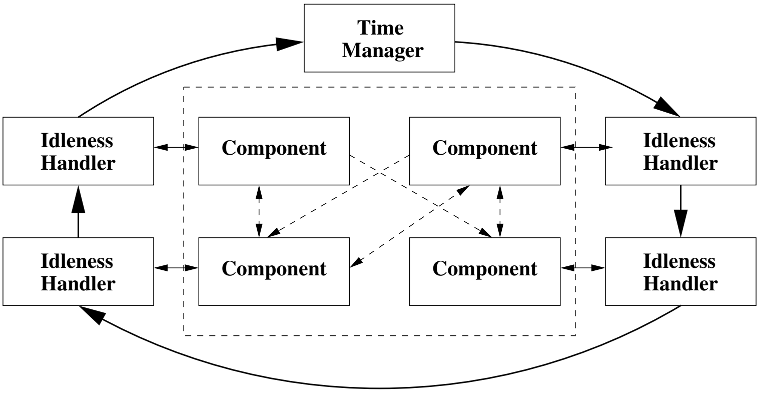


Fig. 2. A Closed System with Simulated Time

the FIFO queues representing the channels. Otherwise, the 0-component starts a new round of termination detection by sending a termination token with the counter equal to 0 and the flag equal to *false*.

5.2 An Extension of the Distributed Detection Algorithm

We extend the Dijkstra's distributed termination detection algorithm to decide on global idleness of the system and to provide time progression. Trying to build an ad-hoc idleness detection into the functions that define the behavior of the test components is error-prone and time-consuming. Therefore, we provide simulated time as a stand alone solution that can be reused for any TTCN-3 test system with simulated time. To check local idleness of the system components, we introduce an *idleness handler* for each system component, i.e. for each test component, for the PA entity and for the SA entity. To decide on global idleness and to progress time, we introduce a time manager. The time manager and the idleness handlers are connected into a ring illustrated in Fig.2. (There the dashed lines represent the border of the original system and the channels within the system.) Although this solution brings a certain overhead, it is generic and independent of the details of a test suit.

The implementation of simulated time consists of a TTCN-3 module and several Java classes. The TTCN-3 module defines the idleness handlers and time progression for the test components. The module can be imported by a specification of a test suite. The Java classes implement a time manager, a timer unit, idleness handlers for the timer unit and for the SUT. The classes are part of the platform and system adapters respectively. When implementing our approach, we have used a series of tools for TTCN-3-based testing provided by the TestingTech company [13].

To implement simulated time, we have to detect global idleness, not termination. After idleness is detected time progresses and detecting global idleness starts in the next time slice again. So we have to ensure time progression and restarting the idleness detection in each time slice. For the sake of simplicity, we consider here only communication based on message passing. The same approach can be used in the case of communication based on procedure calls.

The original algorithm works with two kinds of messages: basic messages and token. In a TTCN-3 test system, we also have to deal with time progression and timeouts. Timeouts are a special kind of messages that are not sent via usual ports but placed into timeout lists. Timeouts can disappear from lists as a result of stopping or resetting timers. The original algorithm works only in case all sent messages are received. Not all timeouts are received by the components. Some of timeouts are lost by stopping and resetting timers, so we handle timeouts separately from basic messages. For basic messages, we assume that the channels of TTCN-3 test system are reliable and that no dynamic reconfiguration takes place in the system.

Time Manager. A time manager initializes idleness detection, decides on global idleness and realizes time progression. The time manager initiates idleness detection by sending an idleness token. As in the original algorithm, the idleness token has a global message counter and a global flag. In order to support time progression, we extend the set of values of the global flag carried by the idleness token. In the original algorithm, the global flag was *true* or *false*. In the extended version, the global flag can be "IDLE_TAG" meaning that there are not active components in the system, "ACTIVE_TAG" meaning that maybe one of the system components is still active, and "TICK_TAG" meaning that time progresses by one time slice. The time manager initiates idleness detection by sending an idleness token with the counter equal to 0 and the flag equal to "IDLE_TAG" to the next idleness handler along the ring.

If the time manager receives the idleness token back with the zero message counter and the global flag with value "IDLE_TAG", it detects global idleness. Otherwise, it repeats idleness detection in the same time slice. If global idleness is detected, the time manager changes the global flag of the token to "TICK_TAG" and sends it along the ring to reinitialise the handlers for idleness detection in the next time slice. After the time manager gets back the token with the "TICK_TAG" global flag, it safely triggers time progression and then starts the idleness detection in the next time slice. Since all time issues are realized by the PA entity, we implement the timer manager as a part of the PA entity.

Idleness Handler. Here we define the general behavior of an *idleness handler*. A TTCN-3 function in Fig. 3 specifies the behavior of idleness handlers. In the Dijkstra's algorithm, termination detection was built into the functionality of components. We separate idleness detection from normal functionality of components by introducing idleness handlers. To guarantee the correctness of this extension of the algorithm, the communication between a component and its idleness handler is synchronized. An idleness handler acknowledges each message received from its component. An idleness handler and its component communicate via port COMP. Ports RINGIN and RINGOUT are used by a handler to receive a token from a previous handler and resp. to send a token to the next handler along the ring.

```
function IdlenessHandler() runs on IdlenessComponent {
  var Token token; var boolean TokenPresent := false;
  var boolean flag := true; var boolean idle := false;
  var integer count := 0;
  while(true){
    alt {
      [] Comp.receive(SEND)
         { count := count+1; Comp.send(ACK);}
      [] Comp.receive(RECV)
         {count := count-1; flag := true; idle := false; Comp.send(ACK); }
      [] Comp.receive(ACTIVATE)
         {flag := true; idle := false; Comp.send(ACK); }
      [] Comp.receive(IDLE)
         {idle := true; Comp.send(ACK); }
      [] RingIn.receive(Token) -> value token
         {TokenPresent := true;}
     }
    if (idle and TokenPresent)
       {if (token.flag==IDLE_TAG or token.flag==ACTIVE_TAG)
          {if (flag){token.flag:=ACTIVE_TAG; flag:=false;}
           token.count:=token.count+count;}
        if (token.tag==TICK_TAG)
           {log("time progression");
            count:=0; flag:=true; idle:=true;}
       RingOut.send(token);
       TokenPresent := false;
       }}}
```

Fig. 3. A TTCN-3 idleness handler

The local message counter, the status represented by the variable *idle* and the local flag of the component are now kept by its idleness handler. Initially, the idle status is *false*, meaning the component is potentially active, the local flag is *true*, meaning the token does not have up-to-date information about the component yet, and the local message counter is zero. The idleness handler keeps track of all the messages sent and received by the component, the component informs the idleness handler about receiving a basic message, sending a basic message or becoming idle by sending “RECV”, “SEND”, and “IDLE” messages respectively. In case of sending, receiving a message or becoming idle, the idleness handler follows the original distributed termination detection algorithm.

In a TTCN-3 test system, a component can become active also if it consumes a timeout. Therefore, the component client notifies its idleness handler about consuming a timeout by the “ACTIVATE” message. This message triggers the idleness handler to change the the local flag to *true* and the idle status to *false*.

Forwarding the idleness token with an “IDLE_TAG” global flag or “ACTIVE_TAG” global flag happens on the same conditions as forwarding the termination token with the *false* and *true* global flag respectively. In case the idleness handler gets the token with the “TICK_TAG”-flag, it reinitializes the local message counter counter, sets the local idleness status and the local flag to *true*. Now the handler is ready for the next time slice.

Transformation of the TTCN-3 Code. The idleness detection works correctly only if the TTCN-3 code of test components follows certain specification pattern and the whole system is configured correctly. By correct configuration we mean that each test component has an port for communication with a unique idleness handler. Moreover the handlers together with the time manager are connected into a ring. The SimulatedTime module implementing simulated time is imported. No dynamic reconfiguration is possible. By the specification pattern, we mean that the code specifying behavior of test components should satisfy the following conditions:

– every TTCN-3 blocking operation (`receive`, `timeout`, `done`, etc.) is preceded by sending "IDLE" to its idleness handler;
– every `receive` statement is followed by sending "RECV" to the idleness handler;
– every `send` statement is followed by sending "SEND" to the idleness handler;
– a timeout statement should be followed by sending "ACTIVATE" to the idleness handler
– sending "IDLE", "RECV", "SEND" and "ACTIVATE" are followed by receiving an acknowledgment from the idleness handler;
– an acknowledgment for "ACTIVATE" is followed by stopping the timer to inform the PA that the timeout is consumed.

The specification pattern can be implemented as an automatic transformation of TTCN-3 specifications. Further we consider implementation of the timer unit in the PA.

Timer Unit. A timer unit implements the TRI operations on timers. Our solution for timer unit keeps active timers in three tables: a "blocked" table for active timers that are not going to expire in the current time slice, a "ready" table for timers ready to expire, and an "expired" table for expired timers, whose timeout message is not consumed yet.

Starting a timer with the zero value leads to deleting the timer from all three tables and adding the timer into the "ready" table. This timer will cause a timeout during the current time slice. Therefore, the timer unit sends "ACTIVATE" to its idleness handler.

Starting a timer with a value greater than zero leads to deleting the timer from all three tables and to adding the timer into the "blocked" table. Stopping a timer leads to removing the timer from all the tree tables. Issuing a timeout moves an expired timer from the "ready" table to the "expired" table. In case there are no other "ready" or "expired" timers, the timer unit reports to its idleness handler "IDLE".

On time progression issued by the time manager, the timer unit increases the elapsed time of all active timers by one, moves the timers that expire in the next time slice into the "ready" table and notifies its idleness handler by the "ACTIVATE" message.

6 Conclusion and Future Work

Using formal methods for the verification of railway control systems is an active area of research. Model checking [7] and theorem proving [6] have been successfully applied to untimed verification of interlockings. Several domain-specific languages [15, 9] have been developed to support automatic verification, validation and system testing.

In this paper, we provided a time semantics that is the most efficient for testing VPI software. When testing with simulated time, we do not waste time on idle phases as in real time testing. Simulated time can be considered as scaled time with a dynamic time factor that is defined automatically. Hence simulated time provides a fair and effective scaling.

We provided a "simulated time" solution for TTCN-3 test systems. The solution is based on an extension of the well-known distributed termination detection algorithm [1]. We implemented our approach as a stand alone solution that can be used for any TTCN-3 test system when testing with simulated time is necessary. This work together with other case studies within the TT-Medal project showed the necessity of simulated time for testing. Formulating proposals for changing TRI so that it allows a straightforward implementation of simulated time in a TTCN-3 test system is the subject of future work.

The Dijkstra's algorithm that we use as a basis for idleness detection works correctly only if the channels of the system are reliable, i.e. no basic message gets lost. The TTCN-3 language provides operations that allow dynamic reconfiguration and clearing the contents of channels. Our current implementation has two limitations: no distributed testing, no dynamic reconfiguration. Dynamic reconfiguration means dynamically adding and removing test components and, consequently, mapping and unmapping ports. Dynamic reconfiguration is potentially dangerous because a reconfiguration can lead to loosing messages. An easy solution is to forbid dynamic reconfiguration of non-idle components.

Distributed testing, where a test system consists of multiple instances of a TTCN-3 test system, is not possible with the current implementation of our solution because such a system would have multiple copies of a time manager instead of a mandatory single copy. This can be solved by disabling time managers in slave copies and by extending termination ring across all the copies. The current implementation of the solution is available on `www.cwi.nl/∼ustin/stime.html`.

Acknowledgments. We would like to thank Daan van der Meij (ProRail, The Netherlands) and Wan Fokkink (Free University of Amsterdam) who provided us with detailed information on VPIs. We also appreciate discussions with Antti Huima (Conformiq Company, Finland) that helped us to improve our solution.

References

1. E. W. Dijkstra, W. H. J. Feijen, and A. J. M. v. Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16(5):217–219, June 1983.
2. ETSI ES 201 873-1 V2.2.1 (2003-02). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. ETSI Standard.

3. ETSI ES 201 873-4 V2.2.1 (2003-02). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 4: TTCN-3 Operational Semantics. ETSI Standard.
4. ETSI ES 201 873-5 V1.1.1 (2003-02). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI). ETSI Standard.
5. ETSI ES 201 873-6 V1.1.1 (2003-02). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 6: TTCN-3 Control Interface (TCI). ETSI Standard.
6. W. J. Fokkink. Safety criteria for the vital processor interlocking at hoornkersenboogerd. In J. Allan, C. A. Brebbia, R. J. Hill, G. Sciutto, and S. Sone, editors, *Proc. 5th Conference on Computers in Railways - COMPRAIL'96, Volume I: Railway Systems and Management, Berlin*, pages 101–110. Computational Mechanics, 1996.
7. S. Gnesi, D. Latella, G. Lenzini, C. Abbaneo, A. M. Amendola, and P. Marmo. An automatic spin validation of a safety critical railway control system. In *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*, pages 119–124. IEEE Computer Society, 2000.
8. J. Grabowski, D. Hogrefe, G. Rthy, I. Schieferdecker, A. Wiles, and C. Willcock. An introduction into the testing and test control notation (TTCN-3). *Computer Networks, Volume 42, Issue 3*, pages 375–403, June 2003.
9. A. E. Haxthausen and J. Peleska. Automatic verification, validation and test for railway control systems based on domain-specific descriptions.
10. T. A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In W. Kuich, editor, *ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 545–558. Springer, 1992.
11. U. Marscheck. Elektronische Stellwerke-internationale *Ü*berblick. *SIGNAL+DRAHT*, 89, 1997.
12. X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *Proc. of the Real-Time: Theory in Practice, REX Workshop*, pages 526–548. Springer-Verlag, 1992.
13. <testing_tech> Testing Technologies. http://www.testingtech.de.
14. TTMedal. Testing and Testing Methodologies for Advanced Languages. http://www.tt-medal.org.
15. F. J. van Dijk, W. J. Fokkink, G. P. Kolk, P. H. J. van de Ven, and S. F. M. van Vlijmen. Euris, a specification method for distributed interlockings. In W. Ehrenberger, editor, *Proc. 17th Conference on Computer Safety, Reliability and Security - SAFECOMP'98, Heidelberg*, volume 1516 of *Lecture Notes in Computer Science*, pages 296–305. Springer, 1998.

Model-Based Testing Through a GUI

Antti Kervinen[1], Mika Maunumaa[1], Tuula Pääkkönen[2], and Mika Katara[1]

[1] Tampere University of Technology, Institute of Software Systems,
P.O. Box 553, FI-33101 Tampere, Finland
firstname.lastname@tut.fi
[2] Nokia Technology Platforms,
P.O. Box 68, FI-33721 Tampere, Finland

Abstract. So far, model-based testing approaches have mostly been used in testing through various kinds of APIs. In practice, however, testing through a GUI is another equally important application area, which introduces new challenges. In this paper, we introduce a new methodology for model-based GUI testing. This includes using Labeled Transition Systems (LTSs) in conjunction with action word and keyword techniques for test modeling. We have also conducted an industrial case study where we tested a mobile device and were able to find previously unreported defects. The test environment included a standard MS Windows GUI testing tool as well as components implementing our approach. Assessment of the results from an industrial point of view suggests directions for future development.

1 Introduction

System testing through a GUI can be considered as one of the most challenging types of testing. It is often done by a separate testing team of domain experts that can validate that the clients' requirements have been fulfilled. However, the domain experts often lack programming skills and require easy-to-use tools to support their work. Compared to application programming interface (API) testing, GUI testing is made more complex by the various user interface issues that need to be dealt with. Such issues include input of user commands and interpretation of the output results, for instance, using text recognition in some cases.

Developers are often reluctant to implement system level APIs only for the purposes of testing. Moreover, general-purpose testing tools need to be adapted to use such APIs. In contrast, a GUI is often available and there are several general-purpose GUI testing tools, which can be easily taken into use. Among the test automation community, however, GUI testing tools are not considered an optimal solution. This is largely due to bad experiences in using so-called capture/replay tools that capture key presses, as well as mouse movement, and replay those in regression tests. The bad experiences are mostly involved with high maintenance costs associated with such a tool [1]. The GUI is often the most volatile part of the system and possible changes to it affect the GUI test automation scripts. In the worst case, the selected capture/replay tool uses bitmap comparisons to verify the results of the test runs. False negative results can then be obtained from minor changes in the look and feel of the system. In practice, such test automation needs maintenance whenever the GUI is changed.

W. Grieskamp and C. Weise (Eds.): FATES 2005, LNCS 3997, pp. 16–31, 2006.

The state of the art in GUI testing is represented by so-called *keyword* and *action word* techniques [2, 3]. They help in maintenance problems by providing a clear separation of concerns between business logic and the GUI navigation needed to implement the logic. Keywords correspond to key presses and menu navigation, such as “click the OK button”, while action words describe user events at a higher level of abstraction. For instance, a single action word can be defined to open a selected file whose name can be given as a parameter. The idea is that domain experts can design the test cases easily using action words even before the system implementation has been started. Test automation engineers then define the keywords that implement the action words using the scripting language provided by the GUI automation tool.

Although some tools use smarter comparison techniques than pure bitmaps, and provide advanced test design concepts, such as keywords and action words, the maintenance costs can still be significant. Moreover, such tools seldom find new bugs and return the investment only when the same test suites are run several times, such as in regression testing. The basic problem is in the static and linear nature of the test cases. Even if only 10% of the test cases would need to be updated whenever the system under test changes, this can mean modifying one hundred test cases from the test suite of one thousand regression tests.

Our goal is to improve the status of GUI testing with model-based techniques. Firstly, by using test models to generate test runs, we will not run into difficulties with maintaining large test suites. Secondly, we have better chances of finding previously undetected defects, since we are able to vary the order of events. Towards these ends, we propose a test automation approach based on Labeled Transition Systems (LTSs) as well as action words and keywords. The idea is to describe a test model as a LTS whose transitions correspond to action words. This should be made as easy as possible for also testers with no programming skills. The maintenance effort should localize to a single model or few component models. The *action machines* we introduce are composed in parallel with *refinement machines* mapping the action words to sequences of keywords. The resulting composite LTS is then read into a general-purpose GUI testing tool that interprets the keywords and walks through the model using some heuristics. The tool also verifies the test results and handles the reporting.

The contributions of this paper are in formalizing the above scheme, introducing novel test model architecture and applying the approach in an industrial case study. Finally, we have assessed the results from an industrial point of view. The rest of the paper is structured as follows. Sections 2 and 3 describe our approach in detail as well as the case study we have conducted. The assessment of the results is given in Section 4. Related work is discussed in Section 5 and conclusions drawn in Section 6.

2 Building a Test Model Architecture

In the following, we will develop a layered test model architecture for testing several concurrently running applications through a GUI. The basis for layering is in keyword and action word techniques, and therefore we will first introduce how to adapt these concepts to model-based testing.

As a running example, we will use testing of Symbian applications. Symbian [4] is an operating system targeted for mobile devices such as smartphones and PDAs. The variety of features available resembles testing of PC applications, but there are also characteristics of embedded systems. For instance, there is no access to the resources of the GUI. In the following, the term system under test (SUT) will be used to refer to a device running Symbian OS.

2.1 Action Words and Keywords

As Buwalda [3] recommends in the description of *action-based testing*, test designers should focus on high-level concepts in test design. This means modeling business processes and picking interesting sequences of events for discovering possible errors. These high-level events are called action words. The test automation engineer then implements the action words with keywords, which act as a concrete implementation layer of test automation.

An example of a keyword from our Symbian test environment is kwPressKey modeling a key press. The keyword could be used, for instance, in a sequence that models starting a calculator application. Such a sequence would correspond to a single action word, say awStartCalculator. Thus, action words represent abstract operations like "make a phone call", "open Calculator" etc. Implementation of action words can consist of sequences of keywords with related parameters as test data. However, the difference between keywords and action words is somewhat in the eye of the beholder. The most generic keywords can almost be considered as action words in the sense of functionality; the main difference is in the purpose of use and the level of abstraction.

Our focus is on the state machine side of the action-based testing. We do not consider decision tables, which are recommended as one alternative for handling test combinations [3]. However, there have been some industrial implementations using spreadsheets to describe keyword combinations to run test cases, and they have proven quite useful. Experiences also suggest that the keywords need to be well described and agreed upon jointly, so that the same test cases can be shared throughout an organization.

2.2 Test Model, Action Machines and Refinement Machines

We use the TVT verification toolset [5] to create test models. With the tools, the most natural way to express the behavior of a SUT is an LTS. We use two tools in the toolset: a graphical editor for drawing LTSs, and a parallel composition tool for calculating the behavior of a system where its input LTSs run in parallel. We will compose our *test model* LTS from small, hand-drawn LTSs with the parallel composition tool. The test model specifies a part of the externally observable behavior of the SUT. At most that part will be tested when the model is used in test runs.

In our test model architecture, hand-drawn LTSs are divided in two classes. *Action machines* are the model-based equivalent for test cases expressed with action words, whereas *refinement machines* give executable contents to action words, that is, refinement from action words to keywords. In the following we formalize these concepts.

Definition 1 (LTS). *A labeled transition system, abbreviated LTS, is defined as a quadruple* $(S, \Sigma, \Delta, \hat{s})$ *where* S *denotes a set of* states*,* Σ *is a set of* actions *(*alphabet*),* $\Delta \subseteq S \times \Sigma \times S$ *is a set of* transitions *and* $\hat{s} \in S$ *is an* initial state. □

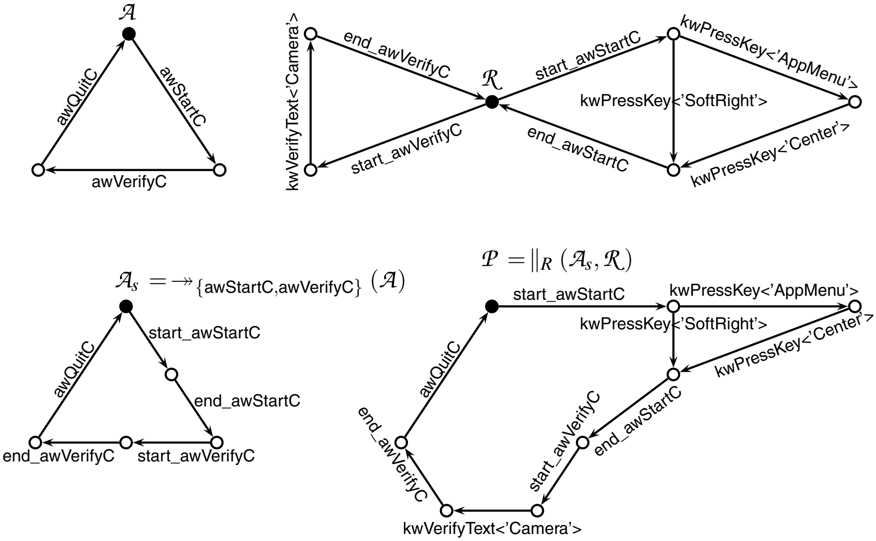


Fig. 1. Transition splitter and parallel composition

Our test model is a deterministic LTS. An LTS $(S, \Sigma, \Delta, \hat{s})$ is deterministic if there is no state in which any leaving transitions share the same action name (label). For example, there are four such LTSs in Figure 1, with their initial states marked with filled circles.

Action machines and refinement machines are LTSs whose alphabets include action words and keywords, respectively. In Figure 1, $\mathcal{A}$ is an action machine and $\mathcal{R}$ is a refinement machine. Action machines describe *what* can be tested at action word level. In $\mathcal{A}$, application can be first started, then verified to be running and finally quitted. After quitting, the application can be started again, and so on. Refinement machines specify *how* action words in action machines can be implemented. Keyword sequences that implement an action word *a* are written in-between start_*a* and end_*a* transitions. In Figure 1, $\mathcal{R}$ refines two action words in $\mathcal{A}$. Firstly, it provides two alternative implementations to action word awStartC. To start an application, a user can either use a short cut key (by pressing "SoftRight") or select the application from a menu. Secondly, verification that the application is running is always carried out by checking that there is text "Camera" on the screen. The action word for quitting the application is not refined by $\mathcal{R}$, but another refinement machine can be used to do that.

During the test execution, we keep track of the current state of the test model, starting from the initial state. LTS $\mathcal{P}$ in Figure 1 would be a simple test model if all of its action words were refined. One of the transitions leaving the current state is chosen. If the label of the transition is not a keyword, the test execution continues from the destination state of the transition. Otherwise, the action corresponding to the keyword is taken: a key is pressed, a text is searched on the display etc. These actions either succeed or fail. For example, text verification succeeds if and only if the searched text can be found on the display. Because sometimes failing an action is allowed, or even required, we need a way to specify the expected successfulnesses of actions in the test model. For that, we use the labeling of transitions. There can be two labels (with and without a tilde) for some keywords; kwVerifyText<'Clock alarm'> and ~kwVerifyText<'Clock alarm'>, for instance. The former label states that in the source state of the transition searching text 'Clock alarm' is allowed to succeed and the latter allows the search to fail.

If the taken action succeeded (failed) a transition without (with) a tilde is searched in the current state. If there is no such transition, an error has been found (that is, the behavior differs from what is required in the test model). Otherwise, the test execution is continued from the destination state of the transition.

Hence, our testing method resembles "exploration testing" introduced in [6]. However, we do not need separate output actions. This is because the only way we can observe the behavior of the SUT is to examine its GUI corresponding to the latest screen capture. In addition, there are many actions that are neither input (keyword) nor output actions. They can be used in debugging (in the execution log, one can see what action word we tried to execute when an error was detected) and in measuring the coverage (for instance, covered high-level actions can be found out).

2.3 Composing a Test Model

We use parallel composition for two different purposes. The main purpose is to create test models that enable extensive testing of concurrent systems. This means that we can test many applications simultaneously. It is clearly more efficient than testing only one application at a time, because now interactions between the applications are also tested. The other purpose is to refine the action machines by injecting the keywords of their refinement machines in correct places in them.

Refinement could be carried out to some extent by replacing transitions labeled by action words with the sequences of transitions specified in refinement machines. However, this kind of macro expansion mechanism would expand action words always to the same keywords, which might not be wanted. For example, it is handy to expand action word "show image" to keywords "select the second menu item" and "press show button" when it is executed for the first time. Later on, the second item should be selected by default in the image menu, and therefore the action word should be expanded to keyword "press show button". We avoid the limits of macro expansion mechanism by using *transition splitting* on action machines and then letting the parallel composition to do the refinement.

The transition splitter divides transitions with given labels in two by adding a new state between the original source and destination states. If the label of a split transition is "*a*" then the new transitions are labeled as "start_*a*" and "end_*a*".

Definition 2 (Transition splitter "$\twoheadrightarrow_A$"). *Let $\mathcal{L}$ be an LTS $(S, \Sigma, \Delta, \hat{s})$ and A a set of actions. $S_{new} = \{s_{s,a,s'} \mid (s,a,s') \in \Delta \wedge a \in A\}$ is a set of new states ($S \cap S_{new} = \emptyset$). Then $\twoheadrightarrow_A (\mathcal{L})$ is an LTS $(S', \Sigma', \Delta', \hat{s}')$ where*

- $S' = S \cup S_{new}$
- $\Sigma' = (\Sigma \setminus A) \cup \{start\_a \mid a \in A\} \cup \{end\_a \mid a \in A\}$
- $\Delta' = \{(s,a,s') \in \Delta \mid a \notin A\}$
  $\cup \{(s, start\_a, s_{s,a,s'}) \mid (s,a,s') \in \Delta \wedge a \in A\}$
  $\cup \{(s_{s,a,s'}, end\_a, s') \mid (s,a,s') \in \Delta \wedge a \in A\}$
- $\hat{s}' = \hat{s}$ □

In Figure 1, LTS $\mathcal{A}_s$ is obtained from $\mathcal{A}$ by splitting transitions with labels aw_StartC and aw_VerifyC.

As already mentioned, we construct the test model with parallel composition. We use a parallel composition that resembles the one given in [7]. Whereas traditional parallel compositions synchronize syntactically the same actions of participating processes, our parallel composition is given explicitly the combinations of actions that should be synchronized and the results of the synchronous executions. This way we can state, for example, that action a in process $\mathcal{P}_x$ is synchronized with action b in process $\mathcal{P}_y$ and their synchronous execution is observed as action c (the result). The set of combinations and results is called *rules*. The parallel composition combines component LTSs to a single composite LTS in the following way.

Definition 3 (Parallel composition "$\|_R$"). $\|_R\ (\mathcal{L}_1,\ldots,\mathcal{L}_n)$ *is the* parallel composition *of n LTSs according to* rules *R. LTS $\mathcal{L}_i = (S_i,\Sigma_i,\Delta_i,\hat{s}_i)$. Let Σ_R be a set of resulting actions and $\surd$ a "pass" symbol such that $\forall i : \surd \notin \Sigma_i$. The rule set $R \subseteq (\Sigma_1 \cup \{\surd\}) \times \cdots \times (\Sigma_n \cup \{\surd\}) \times \Sigma_R$. Now $\|_R\ (\mathcal{L}_1,\ldots,\mathcal{L}_n) = (S,\Sigma,\Delta,\hat{s})$, where*

- $S = S_1 \times \cdots \times S_n$
- $\Sigma = \{a \in \Sigma_R \mid \exists a_1,\ldots,a_n : (a_1,\ldots,a_n,a) \in R\}$
- $((s_1,\ldots,s_n),a,(s'_1,\ldots,s'_n)) \in \Delta$ *if and only if there is* $(a_1,\ldots,a_n,a) \in R$ *such that for every i* $(1 \leq i \leq n)$
  - $(s_i,a_i,s'_i) \in \Delta_i$ *or*
  - $a_i = \surd$ *and* $s_i = s'_i$
- $\hat{s} = \langle \hat{s}_1,\ldots,\hat{s}_n\rangle$ □

A rule in a parallel composition associates an array of actions (or "pass" symbol $\surd$) of input LTSs to an action in resulting LTS. The action is the result of the synchronous execution of the actions in the array. If there is $\surd$ instead of an action, the corresponding LTS will not participate in the synchronous execution described by the rule.

In Figure 1, $\mathcal{P}$ is the parallel composition of $\mathcal{A}_s$ and $\mathcal{R}$ with rules

$$
\begin{aligned}
R = \{\ & \langle \text{start\_awStartC}, \text{start\_awStartC}, \text{start\_awStartC}\rangle,\\
& \langle \text{end\_awStartC}, \text{end\_awStartC}, \text{end\_awStartC}\rangle,\\
& \langle \text{start\_awVerifyC}, \text{start\_awVerifyC}, \text{start\_awVerifyC}\rangle,\\
& \langle \text{end\_awVerifyC}, \text{end\_awVerifyC}, \text{end\_awVerifyC}\rangle,\\
& \langle \text{aw\_QuitC}, \surd, \text{aw\_QuitC}\rangle,\\
& \langle \surd, \text{kwPressKey<'AppMenu'>}, \text{kwPressKey<'AppMenu'>}\rangle,\\
& \langle \surd, \text{kwPressKey<'Center'>}, \text{kwPressKey<'Center'>}\rangle,\\
& \langle \surd, \text{kwPressKey<'SoftRight'>}, \text{kwPressKey<'SoftRight'>}\rangle,\\
& \langle \surd, \text{kwVerifyText<'Camera'>}, \text{kwVerifyText<'Camera'>}\rangle\}
\end{aligned}
$$

2.4 Test Model Architecture

In the SUT, several applications can be run simultaneously, but only one can be active at a time. The active application receives all user input except the one that activates a task switcher. The user can activate an already running application with the task switcher.

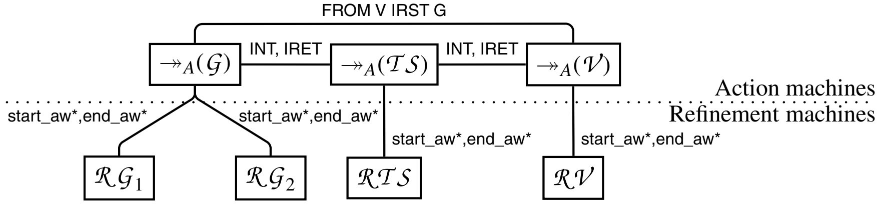


Fig. 2. Test model architecture

This setting forces us to restrict the concurrency (interleavings of actions) in the test model. Otherwise, the test model would allow executing first one keyword in one application and then another keyword in another application without activating the other application first. This would lead to a situation where the test model assumes that both applications have received one input, but in reality, the first application received two inputs and the other none.

Because the activation itself must be expressed as a sequence of keywords, it is natural to model the task switcher as a special application, a sort of a scheduler. The task switcher starts executing when an active application is interrupted, and stops when it activates another (or the same) application. Although the absence of interleaved actions might make the parallel composition look an unnecessarily complicated tool for building the model, it is not. The composition generates a test model that contains all combinations of states in which the tested application can be inactive. Thus, it enables rigorous testing of every application in every combination of states of the other applications in background.

Technically, we have one action machine for every application to be tested, and one action machine for task switching: action machines $\mathcal{G}$ (Gallery application), $\mathcal{V}$ (Voice recorder application) and $\mathcal{TS}$ (task switcher), for instance. Action machines are synchronized with each other and with their refinement machines, as shown in Figure 2. Before the synchronization, all action words of action machines are split.

In the figure, lines that connect action machines to refinement machines represent synchronizing the split action words of the connected processes. For instance, we have a rule for synchronizing $\twoheadrightarrow_A(\mathcal{G})$ and $\mathcal{RG}_1$ with action start_awVerifyImageList and another rule for $\twoheadrightarrow_A(\mathcal{G})$ and $\mathcal{RG}_2$ with start_awViewImage. There are also rules that allow execution of every keyword in refinement machines without synchronization.

Synchronizations that take care of task switching are presented with lines that connect $\mathcal{G}$ and $\mathcal{V}$ to $\mathcal{TS}$ in the figure. Both $\mathcal{G}$ and $\mathcal{V}$ include actions INT and IRET that represent interrupting the application and returning from interrupt. Initially, Gallery application is active. If $\mathcal{G}$ executes INT synchronously with $\mathcal{TS}$, $\mathcal{G}$ goes to a state where it waits for IRET. On the other hand, $\mathcal{TS}$ executes keywords that activate another (or the same) application in the SUT and then execute synchronously IRET with the corresponding action machine.

Finally, there is a connector labeled FROM V IRST G in Figure 2. It represents "go to Gallery" function in Voice recorder. In our SUT, the function activates Gallery application and opens its sound clips menu. Voice recorder is deactivated but left in background. In the test model, action FROM V IRST G is the result of synchronizing actions IGOTO<Gallery> in $\mathcal{V}$ and IRST<VoiceRecorder> actions in $\mathcal{G}$. The first action leads $\mathcal{V}$ to an interrupted state from which it can continue only by executing IRET synchronously with $\mathcal{TS}$. The second action lets $\mathcal{G}$ to continue from an interrupted state, but forces it to a state where sound clip menu is assumed to be on the screen.

Formally, our test model $\mathcal{TM}$ is acquired from the expression:

$$\mathcal{TM} = \|_R \left(\twoheadrightarrow_A (\mathcal{G}), \twoheadrightarrow_A (\mathcal{TS}), \twoheadrightarrow_A (\mathcal{V}), \mathcal{RG}_1, \mathcal{RG}_2, \mathcal{RTS}, \mathcal{RV}\right)$$

where set A contains all the action words and rule set R is as outlined above.

One advantage of this architecture is that it allows us to reuse the component LTSs with a variety of SUTs. For example, if the GUI of some application changes, all we need to change is the refinement machine of the corresponding action machine. If a feature in an application should not be tested, it is enough to remove the corresponding action words from application's action machine. If an application should not be tested, we just drop out its LTSs from the parallel composition. Accordingly, if a new application should be tested, we add the action and the refinement machine for it (also $\mathcal{TS}$ and $\mathcal{RTS}$ must be changed to be able to activate the new application, but they are simple enough to be generated automatically). Moreover, if we test a new SUT with the same features but with completely different GUI, we redraw the refinement machines for the SUT but use the same action machines.

While refinement machines can be changed without touching their action machines, changing an action machine causes easily changes in its refinement machines. If a new action word is introduced in an action machine, either its refinement machine has to be extended correspondingly or a new refinement machine added to the parallel composition. In addition, changing the order of action words inside an action machine may cause changes in its refinement machine. For example, action word awChooseFirstImageInGallery can be unfolded to different sequences of keywords depending on the state of the SUT in which the action word is executed. In one state, Gallery application may already show the image list. Then the action can be completed by a keyword that selects the first item in the list. However, in another state, Gallery may show a list of voice samples, and therefore the refinement should first find out the list of images before the first image can be selected. Thus, action words may contain hidden assumptions on SUT's state where the action takes place. Of course, one can make these assumptions explicit, for example, by extending the action label: awChooseFirstImageInGalleryWhenImageListIsShown.

3 System Testing on Symbian Platform

The above theory was developed in conjunction with an industrial case study. In this section, we will describe the case study including the test environment and setting that we used. Moreover, we outline the implementation of our model-based test engine, and explain the modeling process concerning keyword selection and creation of the test model itself. In addition, we will briefly evaluate our results.

3.1 Test Environment and Setting

The system we tested was a Symbian-based mobile device with Bluetooth capability. The test execution system was installed on a PC, and it consisted of two main components: test automation tool, including our test execution engine, and remote control

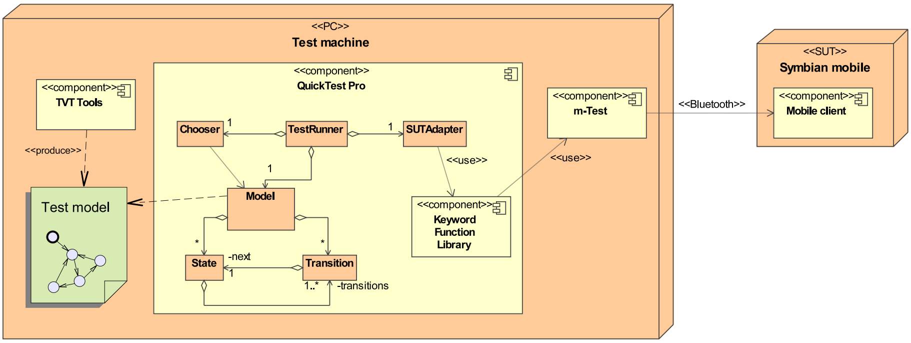


Fig. 3. Test environment

software for the SUT. The test environment is depicted as a UML deployment diagram in Figure 3. We created the test model with TVT tools. As a test automation tool we used Mercury's QuickTest Pro (QTP) [8]. QTP is a GUI testing tool for MS Windows capable of capturing information about window resources, such as buttons and text fields, and providing access to those resources through an API. The tool also enables writing and executing test procedures using a scripting language (Visual Basic Script, VBScript in the following) and recording a test log when requested.

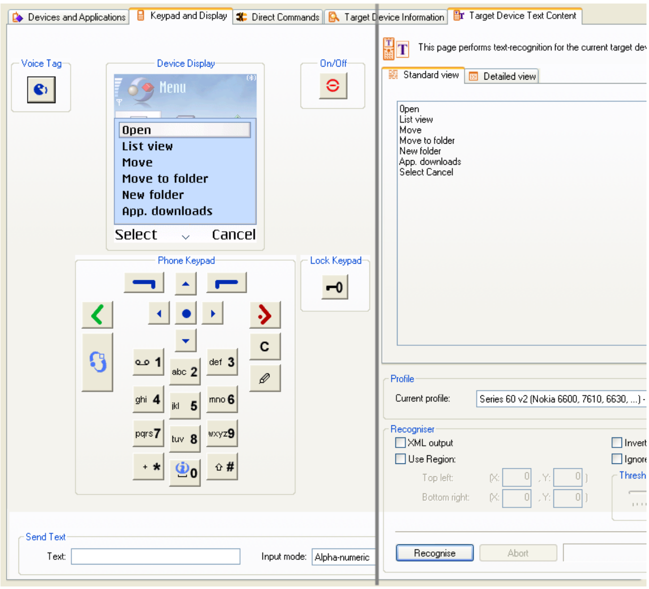
The remote control tool we used was m-Test by Intuwave [9]. It provides access to the GUI of the SUT and to some internal information such as a list of running processes. m-Test makes it possible to remotely navigate through the GUI (see Figure 4, on the left-hand side). GUI resources visible on the display cannot be obtained, only the bitmap of the display is available. Fortunately, m-Test is capable of recognizing text in the bitmap (see Figure 4, on the right-hand side). m-Test supports various ways to connect to the SUT; in the study we used a Bluetooth radio link.

Moreover, in the beginning of the study, we obtained a VBScript function library. It was originally developed to serve as a library of keyword implementations for conventional test procedures in system testing of the SUT. For example, for pushing a button there was a function called 'Key' etc.

3.2 Test Engine

The test execution engine, which executes the LTS state machine, consisted of four parts: execution engine TestRunner, state model Model, transition selector Chooser, and keyword proxy SUTAdapter (see Figure 3). TestRunner was responsible for executing transition events selected by Chooser using the keyword function library via SUTAdapter. Based on the result of executing a keyword, TestRunner determines if the test run should continue or not. If the run can continue, the cycle continues until the number of executed transitions exceeds the maximum number of steps. The test designer determines the step limit that is provided as a parameter.

Model was constructed from states (State), transitions (Transition), and their connections to each other. The test model (LTS) is read from a file and translated to a state object model, which provides access to the states and transitions.



This page performs text-recognition for the current target dev

Open
List view
Move
Move to folder
New folder
App. downloads
Select Cancel

Current profile: Series 60 v2 (Nokia 6600, 7610, 6630, ...) -

Input mode: Alpha-numeric

Fig. 4. Inputs and outputs of SUT as seen from m-Test

Chooser selects a transition to be executed in the current state. The selection method can be random or probabilistic based on weights attached to the transitions. Naturally, more advanced Chooser could also be based on an operation profile [10] or game theory [11], for instance. Since the schedule of our case study was tight, we chose the random selection algorithm because it was the easiest to implement.

The keyword function library that we obtained served as our initial keyword implementation. However, during the early phases of the study it became apparent that it was not suitable for our purposes. The library had too much built-in control over the test procedure. In contrast, our approach requires that the test verdict must be given by the test engine. The reason is that sometimes a failure of a SUT is actually the desired result. In addition, since the flow control was partly embedded in the library, we did not have any keyword that would report the status of the SUT. For that reason, we created a keyword proxy (SUTAdapter). Its purpose was to hide original function library keywords, use or re-implement them to fit our purpose, and to add some new keywords.

3.3 Keyword Categories

We discovered that there must be at least five types of keywords: command, navigate, query, control, and state verification. As an example, some keywords from each category are shown in Table 1. The command type keywords are the most obvious ones: They send input to the SUT, for instance, “press key” or “write text”. Navigation keywords, such as “select menu item”, are used to navigate in the GUI. Query keywords are used to compare texts or images on the display. Control keywords are used to manage the state of the SUT. These four keyword groups are well suited for most of the common

Table 1. Keyword categories

Category	Keyword	Param.	Description
Command	kwPressKey kwWriteText	’keyLeft’ ’Hello’	A key press Send text
Navigate	kwSelectMenu kwSelectAppMenu	’Move’ ’Clock’	Select a menu item Activates an application if started
Query	kwVerifyText	’Move’	Verifes that given text is visible on the display
Control	kwSetTarget kwStartApp	’Phone1’ ’Recorder’	Activates a device to receive subsequent commands Start an application
State verification	kwIsMenuSelected	’Move’	Confirms that the given menu item is selected

testing situations. However, our approach allowed us to create several situations where also the state verification keywords were needed.

State verification keywords verify that the SUT is in some particular state (for instance, “Is menu text selected“) or that some sporadic event, like a phone call, has occurred. These keywords were essential, because the environment did not allow us to capture such information otherwise. The state of the SUT was only available through indirect clues that were extracted from the display bitmap. Because of this, the test model occasionally misinterpreted the state of the SUT or missed an event. This made test modeling somewhat more complicated than we anticipated. The biggest difference between the query and state verification keywords is in the intent of their use. Queries are used to determine the presence of texts etc. on the display, whereas state verification keywords check if the GUI is in a required state. The latter are used to detect if the SUT is in a wrong state, i.e. the failure has occurred.

The missing of an event was the most common error in the model, which occurred often when exact timing was required (like testing an alarm). This problem was probably caused by the slow communication between QTP and m-Test. There were several occasions when some event was missed just because the execution of a keyword was too slow or the execution time varied between runs.

3.4 Modeling Process

When we started our model creation process, we had three constraints: tight schedule, many features to test, and no specifications what so ever. However, we obtained a user guide of the device and a more mature product from the same product family. The latter was used as a test oracle when developing the test model. Using the two, we were able to create a mental model[1] of the behavior of the SUT in various situations. One of our objectives was to find concurrency-related defects. We explored manually through several applications and tried several basic exploration techniques like opening the same application in different ways. Within days, we found the first defect.

Since we believed that “bugs are social creatures”, we put more emphasis on the particular application involved with the first defect. After a couple of days, we found

[1] According to El-Far, human testers develop mental models and the basic idea behind model-based testing is to formalize those models and use them for automatic testing [12].

Table 2. Defects and their resulting states

Defect #	Effect	Result
1	Top bar disappears from Gallery	The top bar remains missing
2	Gallery crashes I	Gallery dies while Camera stays alive
3	Gallery crashes II	Gallery dies while Voice Recorder stays alive
4	Device has to be rebooted	No playback with Real Player and no access to Gallery
5	GUI in busy loop	Recorder's GUI blinks, no sound, and no error message
6	Off by one	The selection in Gallery's menu changes either up or down

several minor defects and one major defect. All these defects where found during the development of the model.

The model itself consisted of seven components (as shown in Figure 2) and modeled a system where two applications run concurrently. After composing the component models, the final test model consisted of 297 states and 351 transitions, and used 17 action words and 18 keywords.

We found only one minor defect while executing the model. However, it should be noted that the SUT was already thoroughly tested before our case study began, and that the model was relatively small. With a larger model and the use of more advanced heuristics, we hope to find more defects. The fact that several defects were discovered before the test model was executed is mainly due to the overall benefits of precise modeling, i.e. it reveals defects very effectively. This is in line with the similar observation in [12].

3.5 Evaluation of Results

Initially, our hypothesis was that model-based testing is suitable for GUI testing and it can find more bugs than linear script-based testing. We also acknowledged that model-based GUI testing does face the same problems as conventional GUI testing, for instance in detecting the state of the SUT. In the following, we evaluate our findings and reflect those to our hypothesis.

Defects. We found six defects in total: two related to the Gallery application, three to the co-operation of Gallery and other applications (Real Player and Voice Recorder), and one to Voice Recorder only. In Table 2, the defects are described in more detail.

The only critical defect was #4, which made it almost impossible to use the device after the failure. The defect was the only one that had been reported previously; others were previously unreported. Defects #2 and #3 can be considered moderate since they allow continuing the normal use of the device. They only interfere with the use of the Gallery application. Defects #1, #5, and #6 are minor since they do not even interfere with the use of the applications.

Defect #6 was the only one that we found while running the model. Even though we are not sure whether #2–#4 are different instances of the same error, this defect was clearly related to GUI component reuse (we had another product of the same family).

As discussed above, defects #1–#5 were found while getting used to the device and developing the model. Excluding #5, they could have been found by executing the model. Automatic detection of the bugs similar to #5 is generally very difficult; the blinking of the screen was so rapid that we could not capture two subsequent screens being different. It should be noted that this problem relates to the underlying GUI test automation tools, not our components or the model.

Even though our case study was not very extensive since the test model covered only a fraction of the functionality of the SUT, it gave us some promising indications towards the suitability of the approach. Especially the development of the test model was considered beneficial, because we found most of the defects while doing that.

Tools. Since our purpose was not to create a model-based GUI testing tool but to prove the concept of model-based GUI testing, the components we implemented were somewhat limited. The major problem was that we did not have a coverage-based selection for the transitions and, thus, we did not measure any model coverage. In addition, we did not provide any indication of what part of the model is being tested and when. In practice, the tester should see the test run advancing by observing the model visually. Another disadvantage was the performance of the underlying GUI testing tools; it took 5–20 seconds to execute a single keyword on an efficient PC.

One limiting factor we found was the scripting language. VBScript has several good features, but it also lacks several key features (dictionaries, inheritance etc.) that are included in other versions of Visual Basic. Those features are not so important in developing linear scripts. However, they would be very helpful in implementing state model executor and similar complex components needed in model-based testing.

4 Assessment from Industrial Point of View

Overall, the Symbian case study proved out to be beneficial. In the most practical sense, it helped by finding a few defects. It was also useful to start considering how model-based testing methodology would fit into the normal way of working. The methodology seems to be an ever-growing promise as a testing approach, but there has not been so many industrial experiences published, especially in the GUI setting.

4.1 The First Step

One starting goal was to evaluate how existing test assets could be used in transition to model-based testing. It seems that the step towards it might not be as steep as assumed. Naturally, model-based testing is a new concept to most testers. However, with some real-life testing scenarios and demonstrations it was possible to disseminate information while keeping a practical point of view.

After initial phase, the lift-off was rapid and we could get some results quite fast. Because of the selected approach, the model was generated manually. Naturally, it would be more interesting to be able to take it into use directly from a modeling tool used by designers or generate it semi-automatically. Creation of the test model was quite straightforward and the model created in the beginning kept its original basic content quite well, i.e. maintenance effort associated with the model was not an issue here.

The tools selected as an underlying GUI test automation have been used for testing the actual products. There were certain functionality and performance problems, but otherwise the tools suited quite well for the new approach. However, Visual Basic Script was found to be quite limited with regard to programming capabilities.

It would have been interesting to see how the practical deployment of the approach would have succeeded. The commercial tools used in the study are familiar to testers of the products, but the TVT toolset is an unsupported university prototype, and its deployment would have taken some effort.

4.2 The Next Step

From a practical test management point of view, model-based testing requires a mindset change from test suites and test cases to test models. Reporting the test results is different from the traditional test cases. With a good model, you can cover many test cases and more, but it is not easy to map fully the coverage of the model to the coverage of the conventional test cases. The test management aspect would also require further studies on how to tackle the situation from the point of view of metrics collecting and test design. This has also been noticed earlier by Robinson [13]. Additional questions would also include whether model-based testing is metrics-wise more effective than conventional techniques as Apfelbaum and Doyle [14] suggest. It is also unclear how much time and money the deployment of model-based testing would take, and how the competences of the testing personnel should be developed.

Another interesting study topic would be to investigate where to obtain the model. If we would obtain a design model, how it should be modified to be used as a test model? More research is also neeeded on how well model-based testing responds to the changes in the product family, i.e. how portable the action word and keyword architecture is when a SUT undergoes changes at different levels (operating system vs. application software, GUI languages and locales), which all create new product variants.

Reusability and portability are promises of model-based testing. Thus, it should be investigated how the testing of relatively similar kinds of products within one product family would benefit from a model-based approach. In further studies, model-based techniques should be used in a real-life testing project already from the beginning. This would allow us to observe what kinds of defects it would reveal. The product development cycle would be enhanced if we could find the most critical ones first.

5 Related Work

There has been some research in the area of model-based testing of GUI systems. The idea of using general purpose GUI test automation tools for model-based testing originates from Robinson [15]. Ostrand et.al. [16] proposed a visual test design environment to create, edit, and maintain test scripts. They used commercial test tool to capture GUI information and replay that information back to the SUT.

Memon proposed in his Ph.D. thesis [17] a framework for testing GUI applications. The framework is based on the knowledge of GUI components. The author derives test cases from GUI structure and usage, measures test coverage and determines the correct actions of the GUI using an oracle based on previously generated test cases and run-time execution information. Belli [18] extended state machines to show not only correct GUI actions but also incorrect transitions. However, in this context, the author prefers regular expression to state machines.

In the Symbian setting, there are some fundamental restrictions compared to conventional GUI testing. Unlike usually, there is no access to the GUI resources, which makes comparisons much harder. Instead of comparing values of text fields, for instance, bitmaps and strings obtained by text recognition must be used.

Compared to Buwalda's work [3], instead of defining finite state machines using spreadsheets, we use LTSs, which is probably the simplest visual formalism for the purpose. To avoid the usual problem of visual models being cluttered, we restrict to small component models that are composed automatically. We map action words to keywords using separate refinement machines that are also defined as LTSs. Furthermore, a separate heuristic component is used to walk through the model.

An example of a related industrial tool for model-based testing is Conformiq Test Generator [19]. It uses UML state machines for test modeling and communicates with a SUT using a SUT-specific adapter. Compared to LTSs, obviously, UML state machines are much more expressive. Naturally, this helps in test modeling and introduces possibilities for data variation, for instance, by generating random data for input fields. However, making mistakes becomes easier with a formalism that is more expressive, especially when the modelers are not UML experts.

6 Conclusions

We have demonstrated how to leverage model-based testing practices in system testing through a GUI. For test modeling, we propose combining Labeled Transition Systems (LTSs) with action words, a proven method for GUI testing. Such action machines are composed in parallel with refinement machines that map the action words to keywords corresponding to the navigation in the GUI. Finally, the composite model is read into a general-purpose GUI test automation system, which executes the model using a heuristic component, and handles the reporting. We have also conducted an industrial case study in which we tested a mobile device and found previously unreported bugs.

To summarize, our results on model-based GUI testing are quite promising. The approach is built on solid theory using proven concepts. Moreover, the associated test automation architecture is simple and includes general-purpose components. From the practical point of view, the execution of test steps should be more dependable and faster, but the issue is in the general-purpose components that are hard to replace. In the future work, a special emphasis should be placed on the topics addressed in the assessment of our results. This could pave the way for an industrial use of the approach.

References

1. Kaner, C., Bach, J., Pettichord, B.: Lessons Learned in Software Testing. Wiley (2001)
2. Fewster, M., Graham, D.: Software Test Automation. Addison–Wesley (1999)
3. Buwalda, H.: Action figures. STQE Magazine, March/April 2003 (2003) 42–47

4. Symbian: Symbian Operating System homepage. (At URL `http://www.symbian.com`)
5. Virtanen, H., Hansen, H., Nieminen, J., Erkkilä, T.: Tampere verification tool. In: Proceedings of TACAS 2004. Volume 2988 of LNCS. Springer-Verlag (2004)
6. Helovuo, J., Leppänen, S.: Exploration testing. In: Proc. 2nd IEEE International Conference on Application of Concurrency to System Design. (2001) 201–210
7. Karsisto, K.: A new parallel composition operator for verification tools. Doctoral dissertation, Tampere University of Technology (number 420 in publications) (2003)
8. Mercury Interactive: QuickTest Pro homepage. (At URL `http://www.mercury.com`)
9. Intuwave: m-Test homepage. (At URL `http://www.intuwave.com`)
10. Musa, J.D.: Software reliability engineering in industry. In: Proc. SAFECOMP'99. Number 1698 in LNCS, Springer-Verlag (1999)
11. Kervinen, A., Virolainen, P.: Heuristics for faster error detection with automated black box testing. In: Proc. International Workshop on Model Based Testing (MBT'04). Number 111 in Electronic Notes in Theoretical Computer Science, Elsevier (2004)
12. El-Far, I.K.: Enjoying the perks of model-based testing. In: Proc. Software Testing, Analysis, and Review Conference (STARWEST) 2001. (2001)
13. Robinson, H.: Obstacles and opportunities for model-based testing in an industrial software environment. (At URL `http://www.geocities.com/harry_robinson_testing/ObstaclesAndOpportunities.pdf`)
14. Apfelbaum, L., Doyle, J.: Model based testing. Software Quality Week Conference (1997)
15. Robinson, H.: Finite state model-based testing on a shoestring. (Software Testing, Analysis, and Review Conference (STARWEST) 1999. At URL `http://www.geocities.com/model_based_testing/shoestring.htm`)
16. Ostrand, T., Anodide, A., Foster, H., Goradia, T.: A visual test development environment for GUI systems. In: ISSTA '98: Proc. 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis, New York, NY, USA, ACM Press (1998) 82–92
17. Memon, A.M.: A comprehensive framework for testing graphical user interfaces. PhD thesis, University of Pittsburgh (2001)
18. Belli, F.: Finite-state testing of graphical user interfaces. In: Proc. 12th International Symposium on Software Reliability Engineering (ISSRE 2001), (IEEE CS) 34–43
19. Conformiq Software: Conformiq Test Generator homepage. (At URL `http://www.conformiq.com`)

Play to Test

Andreas Blass[1],*, Yuri Gurevich[2], Lev Nachmanson[2], and Margus Veanes[2]

[1] University of Michigan, Ann Arbor, MI, USA
ablass@umich.edu
[2] Microsoft Research, Redmond, WA, USA
gurevich, levnach, margus@microsoft.com

Abstract. Testing tasks can be viewed (and organized!) as games against nature. We study reachability games in the context of testing. Such games are ubiquitous. A single industrial test suite may involve many instances of a reachability game. Hence the importance of optimal or near optimal strategies for reachability games. One can use linear programming or the value iteration method of Markov decision process theory to find optimal strategies. Both methods have been implemented in an industrial model-based testing tool, Spec Explorer, developed at Microsoft Research.

1 Introduction

If you think of playful activities, software testing may not be the first thing that comes to your mind, but it is useful to see software testing as a game that the tester plays with the implementation under test (IUT). We are not the first to see software testing as a game [2] but our experience with building testing tools at Microsoft leads us to a particular framework.

An industrial tester typically writes an elaborate test harness around the IUT and provides an application program interface (API) for the interaction with the IUT. You can think of the API sitting between the tester and the IUT. It is symmetric in the sense that it specifies the methods that the tester can use to influence IUT and the methods that the IUT can use to pass information to the tester. From tester's point of view, the first methods are *controllable actions* and the second methods are *observable actions*.

The full state of the IUT is hidden from the tester. Instead, the tester has a model of the IUT's behavior. A model state is given by the values of the model variables which can be changed by means of actions whether controllable or observable. But this is not the whole story. In addition, there is an implicit division of the states into *active* and *passive*; in other words there is an implicit Boolean state variable "the state is active". The initial state is active but, whenever the model makes a transition to a target state where an observable action is enabled, the target state is passive; the target state is active otherwise. At a passive state, the tester waits for an observable action. If nothing happens within a state-dependent timeout, the tester interprets the timeout itself as a default observable action which changes the passive state into an active state

[*] Much of the research reported here was done while the first author was a visiting researcher at Microsoft Research.

W. Grieskamp and C. Weise (Eds.): FATES 2005, LNCS 3997, pp. 32–46, 2006.

with the same values of the explicit variables. At an active state the tester applies one of the enabled controllable actions. Some active states are *final*; this is determined by a predicate on state variables. The tester has an option of finishing the game whenever the state is final.

We presume here that the model has already been checked for correctness. We are testing IUT for the conformance to the model. Here are some examples of how you detect nonconformance. Suppose that the model is in a passive state s. If only actions a, b are enabled in s but you observe an action c, different from a and b, then you witness a violation of the conformance relation. If the model tells you that any non-timeout action enabled in s returns a positive integer but the IUT throws an exception or returns -1, then, again, you have discovered a conformance violation. This kind of conformance relation is close to the one studied by de Alfaro [11].

In a given passive state the next observable action and its result are not determined uniquely in general. What are the possible sources of the apparent nondeterminism? One possible source is that the IUT interacts with the outside world in a way that is hidden from the tester. For example, it is in many cases not desirable for the tester to control the scheduling of the execution threads of a multithreaded IUT; it may be even impossible in the case of a distributed IUT. Another possible source of nondeterminism is that the model state is more abstract than the IUT state. For example, the model might use a set to represent a collection of elements that in reality is ordered in one way or another in the IUT.

The group of Foundations of Software Engineering at Microsoft Research developed a tool, called Spec Explorer, for writing, exploring, and validating software models and for model-based testing of software. Typically the model is more abstract and more compact than the IUT; nevertheless its state space can be infinite or very large. It is desirable to have a finite state space of a size that allows one to explore the state space. To this end, Spec Explorer enables the tester to generate a finite but representative set of parameters for the methods. Also, the tester can indicate a collection of predicates and other functions with finite (and typically small) domains and then follow only the values of these functions during the exploration of the model [13]. These and other ways of reducing the state space are part of a finite state machine (FSM) generation algorithm implemented in the Spec Explorer tool; the details fall outside the scope of this paper. The theoretical foundations of the tool are described in [8]. The tool is available from [1].

The game that we are describing is an example of so-called games against nature which is a classical area in optimization and decision making under uncertainty going back all the way to von Neumann [26]. Only one of the two players, namely the tester, has a goal. The other player is disinterested and makes random choices. Such games are also known as $1\frac{1}{2}$-player games. We make a common assumption that the random choices are made with respect to a known probability distribution. How do we know the probability distribution? In fact, we usually don't. Of course symmetry considerations are useful, but typically they are insufficient to determine the probability distribution. One approximates the probability distribution by experimentation.

The tester may have various goals. Typically they are cover-and-record goals e.g. visit every state (or every state-to-state transition) and record everything that happened in the process. Here we study *reachability games* where the goal is to reach a final state. It is easy to imagine scenarios where a reachability game is of interest all by itself. But we are interested in reachability games primarily because they are important auxiliary games. In an industrial setting, the tester often runs test suites that consist of many test segments. The state where one test segment naturally ends may be inappropriate for starting the next segment because various shared resources have been acquired or because the state is littered with ancillary data. The shared resources should be freed and the state should be cleaned up before the segment is allowed to end. Final states are such clean states where a new segment can be initiated. And so the problem arises of arriving at one of the final states.

It is a priori possible that no final state is reachable from the natural end-state of a test segment. In such a case it would be impossible to continue a test suite. Spec Explorer avoids such unfortunate situations by pruning the FSM so that that it becomes *transient* in the following sense: from every state, at least one final state is reachable (unless IUT crashes). The pruning problem can be solved efficiently using a variation of [10, Algorithm 1] (which is currently implemented in Spec Explorer), or the improved algorithm in [9, Section 4].

The tester cannot run a vast number of test segments by hand. The testing activity at Microsoft gets more and more automated. The Spec Explorer tool plays an important role in the process. Now is the time to expose a simplification that we made above speaking about the tester making moves. It is a testing tool (TT) that makes moves. The tester programs a game strategy into the TT.

The reachability games are so ubiquitous that it is important to compute optimal or nearly optimal strategies for them. You compute a strategy once for a given game and then you use it over and over many times. Since reachability games are so important for us, we research them from different angles.

In Section 2, reachability games are formulated, analysed and solved by means of linear programming and various known results, in particular [10, Theorem 9]. We associate a state dependent cost with each action. The optimal strategy minimizes the expected total cost which is the sum of the costs incurred during the execution. We observe that a reachability game can be viewed as a negative Markov decision process with infinite horizon [22]; the stopping condition is the first arrival at a final state. This allows us, in Section 3, to solve any reachability problem using the well known value iteration method. Theorem 7.3.10 in [22] guarantees the convergence. In Section 4 we provide experimental results by applying both methods to typical model programs using the Spec Explorer tool. Section 5 is devoted to related work. Section 6 gives a short conclusion.

Often the value iteration method works faster than the simplex method, but linear programming has its advantages and sheds some more light on the problem. In general, the applicability of one method does not imply the applicability of the other. Spec Explorer makes use of both, linear programming and value iteration, to generate strategies. Recall that strategy generation happens upon completion of FSM generation and a possible elimination of states from which no final state is reachable. The step of getting from the model program to a particular test graph is illustrated with the following example.

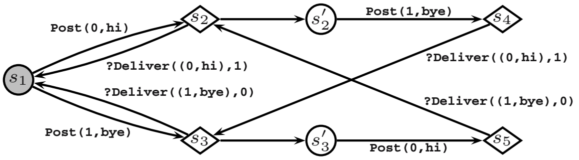


Fig. 1. Sample test graph generated by Spec Explorer from the chat model; diamonds represent passive states; ovals represent active states; links to s'_2 and s'_3 represent transitions to active mode; observable actions are prefixed by a question mark

Example: Chat Session. We illustrate here how to model a simple reactive system. This example is written in the AsmL specification language [16]. The chat session lets a client post messages for the other clients. The state of the system is given by the tuple `(clients,queue,recipients)`, where `clients` is the set of all clients of the session, `queue` is the queue of pending (sender,text) messages, and `recipients` is the set of remaining recipients of the first message in the queue called the *current* message.

```
var clients as Set of Integer
var queue as Seq of (Integer,String)
var recipients as Set of Integer
```

Posting a message is a *controllable* action. The action is enabled if the Boolean expression given by the `require` clause holds. Notice that the second conjunct of the enabling condition is trivially true if the queue is empty.

```
Post(sender as Integer, text as String)
  require sender in clients and
          forall msg in queue holds msg.First <> sender
  if queue.IsEmpty then recipients := clients - {sender}
  queue := queue + [(sender,text)]
```

Delivery of a message is an *observable* action. The current message must be delivered to all the clients other than the sender. Upon each delivery, the corresponding receiver is removed from the set of recipients. If there are no more recipients for the current message, the queue is popped and the next message (if any) becomes the current one. In other words, the specification prescribes that the current message must be delivered to all the recipients before the remainder of the queue is processed.

```
Deliver(msg as (Integer, String), recipient as Integer)
  require not queue.IsEmpty and then
          queue.Head = msg and recipient in recipients
  if recipients.Size = 1 then
    if queue.Length = 1 then recipients := {}
    else recipients := clients - {queue.Tail.Head.First}
    queue := queue.Tail
  else recipients := recipients - {recipient}
```

A good example of a natural finality condition in this case is `queue.IsEmpty`, specifying any state where there are no pending messages to be delivered.

If we configure the chat session example in Spec Explorer so that the initial state is $(\{0,1\},[],\{\})$ with two clients 0 and 1, where client 0 only posts "hi", and client 1 only posts "bye", then we get the test graph illustrated in Figure 1. The initial state is s_1, and that is also the only final state with the above finality condition.

2 Reachability Games and Linear Programming

We use a modification of the definition of a test graph in [20] to describe nondeterministic systems. A *test graph* G has a set V of *vertices* or *states* and a set E of directed *edges* or *transitions*. The set of states splits into three disjoint subsets: the set V^{a} of *active* states, the set V^{p} of *passive* states, and the set V^{g} of *goal* states. Without loss of generality, we may assume that V^{g} consists of a single goal state g such that no edge exits from g; the reduction to this special case is obvious.

There is a *probability function* p mapping edges exiting from passive nodes to positive real numbers such that, for every $u \in V^{\mathrm{p}}$,

$$\sum_{(u,v)\in E} p(u,v) = 1. \tag{1}$$

Notice that this implies that for every passive state there is at least one edge starting from it, and we assume the same for active states. Finally, there is a *cost function c* from edges to positive real numbers. One can think about the cost of an edge as, for example, the time for IUT to execute the corresponding function call. Formally, we denote by G the tuple $(V, E, V^{\mathrm{a}}, V^{\mathrm{p}}, g, p, c)$.

We assume also that for all $u, v \in V$ there is at most one edge from u to v. (This is not necessarily the case in applications; the appropriate reduction is given in Section 2.3.) Thus $E \subset V \times V$. It is convenient to extend the cost function to $V \times V$ by setting $c(u,v) = 0$ for all $(u,v) \notin E$.

2.1 Reachability Game

Let $G = (V, E, V^{\mathrm{a}}, V^{\mathrm{p}}, g, p, c)$ be a test graph and u a vertex of it. The *reachability game* $R(u)$ over G is played by a testing tool (TT) and an implementation under test (IUT). The vertices of G are the states of $R(u)$, and u is the initial state. The current state of the game is indicated by a marker. Initially the marker is at u. If the current state v is active then TT moves the marker from v along one of graph edges. If the current state v is passive then IUT picks an edge (v,w) with probability $p(v,w)$ and moves the marker from v to w. TT wins if the marker reaches g. With every transition e the cost $c(e)$ is added to the total game cost.

A *strategy* for (the player TT in) G is a function S from V^{a} to V such that $(v, S(v)) \in E$ for every $v \in V^{\mathrm{a}}$. Let $R(u,S)$ be the sub-game of $R(u)$ when TT plays according to S.

We would like to evaluate strategies and compare them. To this end, for every strategy S, let $M_S[v]$ be the expected cost of the game $R(v,S)$. Of course, the expected cost may diverge, in which case we set $M_S[v] = \infty$. We say that M_S is *defined* if $M_S[v] < \infty$ for all v. If, for example, c reflects the durations of transition executions then M_S reflects the expected game duration. The expected cost function satisfies the following equations.

$$\begin{aligned} &M_S[g] = 0 \\ &M_S[u] = c(u, S(u)) + M_S[S(u)] \quad \text{for } u \in V^{\mathrm{a}} \\ &M_S[u] = \sum_{(u,v)\in E} \{p(u, v)(c(u, v) + M_S[v])\} \quad \text{for } u \in V^{\mathrm{p}} \end{aligned} \tag{2}$$

We call a strategy S *optimal* if $M_S[v] \le M_{S'}[v]$ for every strategy S' and every $v \in V$, or, more concisely, if $M_S \le M_{S'}$ for every strategy S'. How can we construct an optimal strategy? Our plan is to show that the cost vector M of an optimal strategy is an optimal solution of a certain linear programming problem. This will allow us to find such an M. Then we will define a strategy S such that, for all active states u,

$$c(u, S(u)) + M[S(u)] = \min_{(u,v)\in E}\{c(u, v) + M[v]\}. \tag{3}$$

We will define transient test graph and prove that the strategy S is optimal when the test graph is transient.

Let us suppose from here on that the set V of states is $\{0, 1, ..., n-1\}$ and that the goal state $g = 0$. Consider a strategy S over G. We denote by P_S the following $n \times n$ matrix of non-negative reals:

$$P_S[u, v] = \begin{cases} p(u, v), & \text{if } u \in V^{\mathrm{p}} \text{ and } (u, v) \in E\ ; \\ 1, & \text{if } u \in V^{\mathrm{a}} \text{ and } v = S(u) \text{ or if } u = v = 0; \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

$P_S[u, v]$ is the probability of the move (u, v) when the game $R(u, S)$ is in state u, except that there is no move from state 0. (We could have added an edge $(0, 0)$ in which case there would be no exception.) So P_S is a *probability matrix* (also called a *stochastic matrix*) [12] since all entries are nonnegative and each row sum equals 1.

A strategy S is called *reasonable* if for every $v \in V$ there exists a number k such the probability to reach the goal state within at most k steps in the game $R(v, S)$ is positive. Intuitively, a reasonable strategy may be not optimal but eventually it has some chance of leading the player to the goal state.

Lemma 1. *A strategy S is reasonable for a test graph G if and only if, for some k, there exists, for each vertex v, a path P_v of length at most k from v to the goal state such that, whenever an active vertex w occurs in P_v, then the next vertex in P_v is $S(w)$.*

Proof. The "only if" half is obvious, because a play in $R(v, S)$ that reaches the goal state in at most k steps traces out a path P_v of the required sort. For the "if" half, recall that all the edges of G have positive probabilities. Thus, P_v has a positive probability of being traced by a play of $R(v, S)$, and so this game has a positive probability of reaching g in at most k steps. □

A nonempty subset U of V is *closed* if the game never leaves U after starting at any vertex in U. If S is reasonable, no subset U of $V - \{g\}$ is closed under the game $R(u, S)$, for any $u \in U$. This property is used to establish the following facts.

We let P'_S denote the *minor* of P_S obtained by crossing out from P_S row 0 and column 0. The following lemma follows from [12, Proposition M.3].

Lemma 2. *Let S be a reasonable strategy. Then*

$$\lim_{k\to\infty} {P'_S}^k = 0; \tag{5}$$

$$\sum_{k=0}^{\infty} {P'_S}^k = (I - P'_S)^{-1}. \tag{6}$$

Reasonable strategies can be characterized in terms of their cost vectors as follows. Complete proofs are given in [7].

Lemma 3. *A strategy S is reasonable if and only if M_S is defined. Moreover, if M_S is defined then $M'_S = (I - P'_S)^{-1} b'_S$ where M'_S and b'_S are the projections to the set $V - \{0\}$ of the expected cost vector M_S and the "immediate cost" vector b_S defined by*

$$b_S[u] = \sum_{v\in V} P_S[u,v]c(u,v) \quad (\forall u \in V).$$

A vertex v of a test graph is called *transient* if the goal state is reachable from v. We say that a test graph is *transient* if all its non-goal vertices are transient. There is a close connection between transient graphs and reasonable strategies.

Lemma 4. *A test graph is transient if and only if it has a reasonable strategy.*

In practice, the probabilities and costs in a test graph may not be known exactly. It is therefore important to know that, as long as the graph is transient, the optimal cost is robust, in the sense that it is not wildly sensitive to small changes in the probabilities and costs. This sort of robustness is, of course, just continuity, which the next lemma establishes.

Lemma 5. *For transient test graphs, the optimal cost vector M is a continuous function of the costs $c(u,v)$ and the probabilities $p(u,v)$.*

Proof. Throughout this proof, "continuous" means as a function of the costs $c(u,v)$ and the probabilities $p(u,v)$.

Temporarily consider any fixed, reasonable strategy S for the given test graph. Thanks to Lemma 1, S remains reasonable when we modify the probabilities (and costs) as long as they remain positive.

The formula for b_S in Lemma 3 shows that this vector is continuous. So is the matrix $I - P'_S$. Since the entries in the inverse of a matrix are, by Cramer's rule, rational functions of the entries of the matrix itself, we can infer the continuity of $(I - P'_S)^{-1}$ and therefore, by Lemma 3, the continuity of M'_S. Since the only component of M_S that isn't in M'_S is 0, we have shown that M_S is continuous.

Now un-fix S. The optimal cost vector M is simply the component-wise minimum of the M_S, as S ranges over the finite set of reasonable strategies. Since the minimum of finitely many continuous, real-valued functions is continuous, the proof of the lemma is complete. □

Of course, we cannot expect the optimal strategy to be a continuous function of the costs and probabilities. A continuous function from the connected space of cost-and-probability functions to the finite space of strategies would be constant, and we certainly cannot expect a single strategy to be optimal independently of the costs and probabilities. Nevertheless, the optimal strategies are robust in the following sense.

Suppose S is optimal for a given test graph, and let an arbitrary $\varepsilon > 0$ be given. Then after any sufficiently small modification of the costs and probabilities, S will still be within ε of optimal. Indeed, the continuity, established in the proof of Lemma 5, of the function M_S and of its competitors $M_{S'}$ arising from other strategies, ensures that, if we modify the costs and probabilities by a sufficiently small amount, then no component M_S will increase by more than $\varepsilon/2$ and no component of any $M_{S'}$ will decrease by more than $\varepsilon/2$. Since $M_S \leq M_{S'}$ before the modification, it follows that $M_S \leq M_{S'} + \varepsilon$ afterward.

A similar argument shows that, if S is strictly optimal for a test graph G, in the sense that any other S' has all components of $M_{S'}$ strictly larger than the corresponding components of M_S, then S remains strictly optimal when the costs and probabilities are modified sufficiently slightly. Just apply the argument above, with ε smaller than the minimum difference between corresponding components of M_S and any $M_{S'}$.

2.2 Linear Programming

Ultimately, our goal is to compute optimal strategies for a given test graph G. We start by formulating the properties of the expected cost vector M as the following optimization problem. Let $\mathbf{d}$ be the constant row vector $(1, ..., 1)$ of length $|V| = n$.

LP: Maximize $\mathbf{d}M$, i.e. $\sum_{u \in V} M[u]$, subject to $M \geq 0$ and

$$\begin{cases} M[0] \leq 0 \\ M[u] \leq c(u,v) + M[v] \quad \text{for } u \in V^{\mathrm{a}} \text{ and } (u,v) \in E \\ M[u] \leq \sum_{(u,v)\in E}\{p(u,v)(c(u,v) + M[v])\} \quad \text{for } u \in V^{\mathrm{p}} \end{cases}$$

Intuitively, a feasible solution M to LP, e.g. $M = \mathbf{0}$, approximates an expected cost vector from below. Assuming that an optimal solution exists, the larger the value of $\mathbf{d}M$ is for a feasible solution M, the closer M is to an optimal cost vector.

Test graphs reduce to a subclass of negative stationary Markov decision processes (MDPs) with an infinite horizon, where rewards are negative and thus regarded as costs, strategies are stationary, i.e. time independent, and there is no finite upper bound on the number of steps in the process. A transient test graph reduces to a negative MDP where the probability to reach the goal from every state is positive. The optimization criterion for our strategies corresponds to the expected *total* reward criterion, rather than the expected *discounted* reward criterion used in discounted Markov decision problems. The total reward optimization problem is in general harder than the discounted reward optimization problem. However, for this subclass of negative MDPs the total reward optimization problem is known to be solvable by linear programming and yields a unique optimal solution [10, Theorem 9]. From Alfaro's Theorem 9 [10] we get the following corollary for test graphs. A careful self-contained proof of the corollary is given in [7].

Corollary 1. *The following statements are equivalent for all test graphs G.*

(a) G is transient.
(b) G has a reasonable strategy.
(c) LP for G has a unique optimal solution M. Moreover, $M = M_S$ for some strategy S and the strategy S is optimal.

Now we presume that the test graph is transient and show how to construct an optimal strategy. By applying Corollary 1 and solving LP, find the cost vector M of some optimal strategy O. In our notation, $M = M_O$. Construct strategy S so that equation (3) is satisfied for every active state u.

Proposition 1. *The constructed strategy S is optimal.*

Notice that, even though an optimal strategy S yields a unique cost vector M_S, S itself is not necessarily unique. Consider for example a test graph without passive states and with edges $\{1 \xrightarrow{1} 2,\ 2 \xrightarrow{10} 0,\ 1 \xrightarrow{10} 3,\ 3 \xrightarrow{1} 0\}$ that are annotated with costs; clearly both of the two possible strategies are optimal.

2.3 Graph Transformation

We made the assumption that for each two vertices in the graph there is at most one edge connecting them. Let us show that we did not lose any generality by assuming this. For an active state u and for any $v \in V$ let us choose an edge leading from u to v with the smallest cost and discard all the other edges between u and v. For a passive state u replace the set of multiple edges D between u and v with one edge e such that $p(e) = \sum_{e' \in D} p(e')$ and $c(e) = (\sum_{e' \in D} p(e')c(e'))/p(e)$. This merging of multiple edges into a single edge does not change the expected cost of one step from u. The graph modifications have the following impact on LP. With removal of the edges exiting from active states we drop the corresponding redundant inequalities. The introduction of one edge for a passive state with changed c and p functions does not change the coefficients before $M[v]$ in LP in the inequality corresponding to passive states and therefore does not change the solution of LP.

2.4 Graph Compression

Every test graph is equivalent, as far as our optimization problems are concerned, to one in which no edge joins two passive vertices. The idea is to replace the edges leaving a passive vertex u in the following manner. Consider all paths emanating from u, passing through only passive vertices, but then ending at an active vertex or the goal vertex. Each such path has a probability, obtained by multiplying the probabilities of its edges, and it has a cost, obtained by adding the costs of its edges. Replace each such path by a single edge, from u to the final, active vertex in the path; give this new edge the same probability and cost that the path had. If this replacement process produces several edges joining the same pair of vertices, transform them to a single edge as in Subsection 2.3. The details of test graph compression are given in [7].

One may wonder if such a compression is worthwhile. The answer depends to a great extent on the topology of the test graph. It may sometimes pay off to apply the

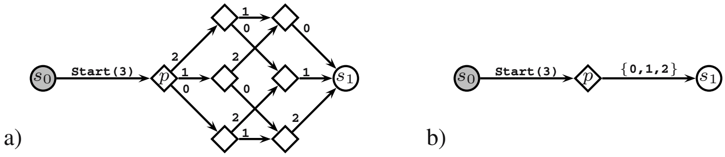


Fig. 2. a) Test subgraph obtained by exploring the extended chat model with 3 clients up to the posting phase; transitions from passive states (diamonds) are labeled by the respective client entering the session. b) Same subgraph after compression.

compression to certain subgraphs of the full test graph, rather than to the whole test graph. Let us illustrate a fairly common situation that arises in testing highly concurrent systems where the compression would reduce the number of states and edges. We revisit the chat model above and extend it as follows. There is an additional state variable `nClients` representing the number of clients entering in the chat session, so the state is given by the tuple `(nClients,clients,queue,recipients)`. There is a new *controllable* action `Start` that starts the entering phase of clients by updating `nClients`. There is also a new *observable* action `Enter` representing the event of a client entering the session. A client that has already entered the session cannot enter it again.

```
var nClients as Integer
Start(n as Integer)
  require nClients = 0 and n > 0
  nClients := n
Enter(c as Integer)
  require nClients > 0 and c in {0..nClients-1} - clients
  clients := clients + {c}
```

Assume also that the enabling condition (`require` clause) of the `Post` action is extended with the condition that the entering phase was started and that all clients have entered the session, i.e., `nClients > 0 **and** clients.Size = nClients`. So the "posting" phase is not started until all clients have entered the session. Suppose that the initial state s_0 is $(0, \emptyset, [], \emptyset)$. By generating the FSM from the model program with 3 clients, the initial part of the test graph up to the posting phase that starts in state s_1 is illustrated in Figure 2.a. The compression of the subgraph between the states s_0 and s_1 would yield the subgraph shown in Figure 2.b with a single passive state p and a transition from p to s_1 representing the composed event of all three clients having entered the session in some order.

The effect of the compression algorithm is in some cases, such as in this example, similar to partial order reduction. Obviously, reducing the size of the test graph improves feasibility of the linear programming approach. However, for large graphs we use the value iteration algorithm, described next. Due to the effectiveness of value iteration the immediate payoff of compression is not so clear, unless compression is simple and the number of states is reduced by an order of magnitude. We are still investigating the practicality of compression and it is not yet implemented in the Spec Explorer tool.

3 Value Iteration

Value iteration is the most widely used algorithm for solving discounted Markov decision problems (see e.g. [22]). Reachability games give rise to non-discounted Markov decision problems. Nevertheless the value iteration algorithm applies; this is a practical approach for computing strategies for transient test graphs.

Let $G = (V, E, V^{\mathrm{a}}, V^{\mathrm{p}}, g, p, c)$ be a test graph. The classical value iteration algorithm works as follows on G.

Value iteration. Let $n = 0$ and let M^0 be the zero vector with coordinates V so that every $M^0[u] = 0$. Given n and M^n, we compute M^{n+1} (and then increment n):

$$M^{n+1}[u] = \begin{cases} \min_{(u,v)\in E}\{c(u,v) + M^n[v]\}, & \text{if } u \in V^{\mathrm{a}}; \\ \sum_{(u,v)\in E} p(u,v)(c(u,v) + M^n[v]), & \text{if } u \in V^{\mathrm{p}}; \\ 0, & \text{if } u = 0. \end{cases} \tag{7}$$

Value iteration for negative MDPs with the expected total reward criterion, or *negative Markov decision problems* for short, does not in general converge to an optimal solution, even if one exists. However, if there exists a strategy for which the expected cost is finite for all states [22, Assumption 7.3.1], then value iteration *does* converge for negative Markov decision problems, see [22, Theorem 7.3.10] or [10, Theorem 10]. In light of lemmas 3 and 4, this implies that value iteration converges for transient test graphs. Let us make this more precise, as a corollary of Theorem 7.3.10 in [22] or Theorem 10 in [10].

Corollary 2. *Let G be a transient test graph as above. For any $\varepsilon > 0$, there exists N such that, for all $n \geq N$ and all states $u \in V$, $M^*[u] - M^n[u] < \varepsilon$, where M^* is the optimal cost vector.*

The iterative process, generally speaking, does not reach a fixed point in finitely many iterations. Consider the test graph in Figure 3. It is not difficult to calculate that the infinite sequence $(M^n[1])_{n=1}^{\infty}$ computed by (7) is

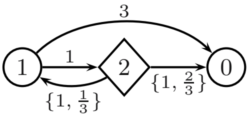


Fig. 3. Sample test graph; transitions from active states are labeled by their costs; transitions from passive states are labeled by their costs and probabilities

$$1,\ 2,\ 2\tfrac{1}{3}\ 2\tfrac{2}{3},\ 2\tfrac{7}{9},\ 2\tfrac{8}{9},\ 2\tfrac{25}{27},\ 2\tfrac{26}{27},\ \ldots,\ 2\tfrac{3^i-2}{3^i},\ 2\tfrac{3^i-1}{3^i},\ \ldots$$

that converges to $M^*[1] = 3$.

When should we terminate the iteration? Given a cost vector M let S_M denote any strategy defined so that equation (3) is satisfied for every active state u. Further, let

$S_n = S_{M^n}$. Observe that the total number of possible strategies is finite and that any non-optimal strategy occurs only finitely many time in the sequence $S_0, S_1, \ldots$. Thus, from some point on, every S_n is optimal. In reality, the desired n is typically not that large because the convergence of the computed costs towards the optimal costs is exponentially fast. For practical purposes, the iteration process halts when the additional gain is absorbed in rounding errors.

4 Experiments

We have conducted some experiments in Spec Explorer in order to evaluate which approach performs better on test graphs that arise from typical model programs, whether linear programming implemented by the simplex method or value iteration. Even for small examples the value iteration method has outperformed linear programming, and we are yet to find examples for which linear programming would be preferable. For some test graphs with several thousand vertices, the straightforward value iteration method as described above is also inadequate. We are currently investigating extensions of the value iteration method as well as the use of graph compression explained above. Table 1 compares the running times of the value iteration and simplex method for test graphs generated from the following two sample problems. Both examples are available in the Spec Explorer distribution [1].

Table 1. Comparison of value iteration and linear programming on sample problems

Sample problem	**Number of vertices in test graph**	**Value iteration running time**	**Simplex running time**	**Ratio simplex/ value iteration**
Chat	40	4ms	33ms	8
Chat	92	25ms	370ms	15
Chat	225	100ms	7000ms	70
Bag	128	19ms	635ms	34
Bag	277	135ms	5600ms	41

Chat. Described in Section 1.

Bag. A multi-threaded implementation of a bag (multiset). Several concurrent users are allowed to add, delete and lookup elements from a shared bag. The example is a variation of a case study used in [23].

The different test graph sizes for Chat and Bag arise by varying the exploration settings using the Spec Explorer tool in a way that was illustrated on a smaller scale with that Chat model in Section 1. To motivate the relevance of the different cases in the table let us take a closer look at the two test graphs generated from the Bag model.

In the case with 128 vertices, the number of users (that equals the number of concurrent threads in the implementation) is 2 and the maximum allowed size of the bag is 1. This configuration yields an exploration of the state space with all the possible interleavings of the bag operations performed by the users. The test suite generated from the test graph gives a global strategy to cover all the interleavings of the observable thread operations.

The case with 277 vertices corresponds to a situation when the maximum bag size is 2 elements and again there are 2 users. This provides a test suite that, in particular, covers the scenario when one of the users tries to remove all occurrences of a particular element and the other user tries to add that element to the bag.

The numbers 128 and 277 are in the range where the solutions are not obvious to the tester but the state graph is small enough for visual inspection and overview. The accepting state in both cases is the state where both users or threads are inactive.

The implementation of the bag example in the distribution [1] has a locking error that can only be discovered with at least two concurrent users accessing the bag.

5 Related Work

Extension of the FSM-based testing theory to nondeterministic and probabilistic FSMs got some attention a while ago [14, 21, 28]. The use of games for testing is pioneered in [2]. A recent overview of using games in testing is given in [27].

An implementation that conforms to the given specification can be viewed as a refinement of the specification. In study [11], based on [3], the game view is proposed as a general framework for dealing with refining and composing systems. Models with controllable and observable actions correspond to interface automata in [11].

Model-based testing allows one to test a software system using a specification (a.k.a. model) of the system under test [5]. There are other model-based testing tools [4, 17, 18, 19, 24]. To the best of our knowledge, Spec Explorer is currently alone in supporting the game approach to testing. Our models are Abstract State Machines [15]. In Spec Explorer, the user writes models in AsmL [16] or in Spec# [6]. The theoretical foundations of the tool are described in [8].

In [20] several algorithms are described that generate optimal length-bounded strategies from test graphs, where optimality is measured by minimizing the total cost while maximizing the probability of reaching the goal, if the goal is reachable. The problem of generating a strategy with optimal *expected* cost is stated as an open problem in [20].

Solving negative and positive Markov decision problems with the total reward criterion are studied in [10] to address the basic problems of computing the minimum and maximum probability or the minimum and maximum expected time to reach a target set in a probabilistic system. In particular, the problem of computing the minimum expected time can be reduced to the negative Markov decision problem with the total reward criterion. In this paper we used Alfaro's Theorem 9 [10], which shows that linear programming works for negative MDPs after eliminating vertices from which the target state is not reachable, and that the optimal solution of the LP is unique.

One may wonder how transient stochastic games [12, Section 4.2] are related to transient test graphs. A transient stochastic game is a game between two players that will stop with probability 1 no matter which strategies are used. This condition gives rise to a proper subclass of transient test graphs where *all* strategies are reasonable. Recall that a test graph is transient if and only if there *exists* a reasonable strategy. An unreasonable strategy is for example a strategy that takes you back and forth between two active states.

6 Conclusion

One of the main contributions of this paper is the identification of reachability games on transient test graphs as a fundamental notion in the area of testing of probabilistic systems. We show how known results, especially those on Markov decision process theory, can be used for a powerful effect in the context of testing. We provide some experimental results. And we worked out a careful self-contained exposition [7] of all the material. Finally let us note that this paper addresses a relatively easy case when all the states are known in advance. The more challenging (and important) case is on-the-fly or online testing where new states are discovered as you go [25]. In a sense, this paper is a warmup before tackling on-the-fly test strategy generation.

References

1. Spec Explorer. URL:http://research.microsoft.com/specexplorer, released January 2005.
2. R. Alur, C. Courcoubetis, and M. Yannakakis. Distinguishing tests for nondeterministic and probabilistic machines. In *Proc. 27th Ann. ACM Symp. Theory of Computing*, pages 363–372, 1995.
3. R. Alur, T. A. Henzinger, O. Kupferman, and M. Vardi. Alternating refinement relations. In *Proceedings of the Ninth International Conference on Concurrency Theory (CONCUR'98)*, volume 1466 of *LNCS*, pages 163–178. Springer, 1998.
4. C. Artho, D. Drusinsky, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, G. Rosu, and W. Visser. Experiments with test case generation and runtime analysis. In Börger, Gargantini, and Riccobene, editors, *Abstract State Machines 2003*, volume 2589 of *LNCS*, pages 87–107. Springer, 2003.
5. M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Towards a tool environment for model-based testing with AsmL. In Petrenko and Ulrich, editors, *Formal Approaches to Software Testing, FATES 2003*, volume 2931 of *LNCS*, pages 264–280. Springer, 2003.
6. M. Barnett, R. Leino, and W. Schulte. The Spec# programming system: An overview. In M. Huisman, editor, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: International Workshop, CASSIS 2004*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.
7. A. Blass, Y. Gurevich, L. Nachmanson, and M. Veanes. Play to test. Technical Report MSR-TR-2005-04, Microsoft Research, January 2005. Revised April 5, 2005.
8. C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Model-based testing of object-oriented reactive systems with Spec Explorer. Technical Report MSR-TR-2005-59, Microsoft Research, 2005.
9. K. Chatterjee, M. Jurdziński, and T. Henzinger. Simple stochastic parity games. In *CSL 03: Computer Science Logic*, Lecture Notes in Computer Science 2803, pages 100–113. Springer-Verlag, 2003.
10. L. de Alfaro. Computing minimum and maximum reachability times in probabilistic systems. In *International Conference on Concurrency Theory*, volume 1664 of *LNCS*, pages 66–81. Springer, 1999.

11. L. de Alfaro. Game models for open systems. In N. Dershowitz, editor, *Verification: Theory and Practice: Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772 of *LNCS*, pages 269 – 289. Springer, 2004.
12. J. Filar and K. Vrieze. *Competitive Markov decision processes*. Springer-Verlag New York, Inc., 1996.
13. W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *ISSTA'02*, volume 27 of *Software Engineering Notes*, pages 112–122. ACM, 2002.
14. S. Gujiwara and G. V. Bochman. Testing non-deterministic state machines with fault-coverage. In J. Kroon, R. Heijunk, and E. Brinksma, editors, *Protocol Test Systems*, pages 363–372, 1992.
15. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
16. Y. Gurevich, B. Rossman, and W. Schulte. Semantic essence of AsmL. *Theoretical Computer Science*, 2005. To appear in special issue dedicated to FMCO 2003, preliminary version available as Microsoft Research Technical Report MSR-TR-2004-27.
17. A. Hartman and K. Nagin. Model driven testing - AGEDIS architecture interfaces and tools. In *1st European Conference on Model Driven Software Engineering*, pages 1–11, Nuremberg, Germany, December 2003.
18. C. Jard and T. Jéron. TGV: theory, principles and algorithms. In *The Sixth World Conference on Integrated Design and Process Technology, IDPT'02*, Pasadena, California, June 2002.
19. V. V. Kuliamin, A. K. Petrenko, A. S. Kossatchev, and I. B. Bourdonov. UniTesK: Model based testing in industrial practice. In *1st European Conference on Model Driven Software Engineering*, pages 55–63, Nuremberg, Germany, December 2003.
20. L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann, and W. Grieskamp. Optimal strategies for testing nondeterministic systems. In *ISSTA'04*, volume 29 of *Software Engineering Notes*, pages 55–64. ACM, July 2004.
21. A. Petrenko, N. Yevtushenko, and G. v. Bochmann. Testing deterministic implementations from nondeterministic FSM specifications. In B. Baumgarten, H.-J. Burkhardt, and A. Giessler, editors, *IFIP TC6 9th International Workshop on Testing of Communicating Systems*, pages 125–140. Chapman & Hall, 1996.
22. M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Mathematical Statistics. A Wiley-Interscience, New York, 1994.
23. S. Tasiran and S. Qadeer. Runtime refinement checking of concurrent data structures. *Electronic Notes in Theoretical Computer Science*, 113:163–179, January 2005. Proceedings of the Fourth Workshop on Runtime Verification (RV 2004).
24. J. Tretmans and E. Brinksma. TorX: Automated model based testing. In *1st European Conference on Model Driven Software Engineering*, pages 31–43, Nuremberg, Germany, December 2003.
25. M. Veanes, C. Campbell, W. Schulte, and N. Tillmann. Online testing with model programs. In *ESEC/FSE'05*, 2005.
26. J. von Neumann and O. Morgenstern. *The Theory of Games and Economic Behavior*. Princeton University Press, 1944.
27. M. Yannakakis. Testing, optimization, and games. In *Proceedings of the Nineteenth Annual IEEE Symposium on Logic In Computer Science, LICS 2004*, pages 78–88. IEEE, 2004.
28. W. Yi and K. G. Larsen. Testing probabilistic and nondeterministic processes. In *Testing and Verification XII*, pages 347–61. North Holland, 1992.

A Note on an Anomaly in Black-Box Testing

Antti Huima

Conformiq Software Ltd.
antti.huima@conformiq.com

Abstract. Testing should not reduce confidence in the system under test – unless defects are found. We show that for a general class of finite-state systems this intuition is incorrect. We base our argument on the view of risk as a probability. We calculate the risk of having an invalid implementation, based on a concrete, believable fault model, and show that executing correct test runs can actually decrease confidence in the system under test. This anomaly is important as it explains some of the difficulty in establishing mathematical links between fault models and testing efficiency. The presented anomaly itself is claimed to be independent of the particular structure of systems. We provide critique of the result, and discuss the potential limits of the presented anomaly as well as ways to remedy it.

1 Introduction

Three years ago, we took part in an academic research project at a local university. The project was about formal black-box testing based on Petri nets. [9, 13] Quite early, the researchers from the university suggested that testing coverage could be measured by calculating the number of Petri net places or transitions that have been visited during testing. We asked from the researchers if there would be a mathematical way to argument for such coverage metrics. For example, could it be possible to prove mathematically that it is more beneficial to count transitions than places?

We found soon that such a quantitative analysis would not be possible without a fault model. We considered informally simple fault models such as the "disappearing" of pre-places from Petri net transitions with a uniform probability. But we run into problems like that multiple mutations or faults could cancel each others and result in a conforming system, making it very difficult to calculate even the a priori probability of a conformant implementation [7, 8].

When the project ended, no conclusion had been reached. We continued to study the issue, considering simple finite-state models and simple but reasonable fault models for them. In our view, a coverage metric would be better than another metric if its increase would correlate with a faster increase in confidence in the system under test than the other one. Eventually we found a strange phenomenon: it would be possible to test systems so that passing test runs would actually *decrease* confidence and *increase* the risk of an erroneous implementation. In this paper, we report our findings regarding this anomaly and provide a critical view on them.

W. Grieskamp and C. Weise (Eds.): FATES 2005, LNCS 3997, pp. 47–61, 2006.

Short Description of the Anomaly. For an informal description of the result, let us define the following observations:

$$\begin{aligned} T &: \text{A small amount of testing has been done} \\ T' &: \text{A large amount of testing has been done} \\ C &: \text{No defect has been detected during testing} \\ D &: \text{The SUT has potential to work incorrectly in general} \end{aligned}$$

The anomaly is that in the context of testing a black-box SUT, $\mathbf{P}(C|T) \geq \mathbf{P}(C|T')$ is in general true whereas $\mathbf{P}(D|C,T) \geq \mathbf{P}(D|C,T')$ is not. In words, the probability of having spotted a defect does not decrease when testing progresses (an obvious fact), but the probability that the SUT is actually incorrectly implemented may increase at the same time as testing progresses without finding defects at all. To paraphrase, it is possible to execute test runs that all terminate with "pass", and yet the risk that the system might work incorrectly increases.

This is not an anomaly in testing per se, neither a problem with mathematics. The point we want to make is that this anomaly exists in the conventional thinking about black-box testing. Even though it does not in any way diminish the value of the current large body of black-box testing theory and methodology, we believe that this anomaly should be reported and studied further.

Related Work. The main research topics related to this note are the testing of finite-state systems [3], in particular the **ioco**-theory [15, 16], and the research on fault models [10, 11]. As this is a technical note, we assume that the reader is familiar with these subjects. Further pointers to the literature are included in the text below.

2 Mathematical Model

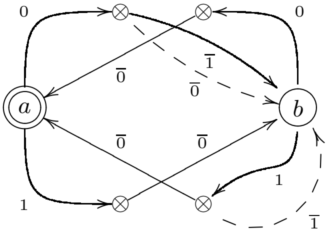
We will now proceed to establish a mathematical model in which to describe the anomaly in detail.

2.1 Bit Transducers

Let $\mathscr{B}$ denote the set $\{0,1\}$, i.e. the set of bits. We introduce first a simple model of nondeterministic finite-state systems, capable of input and output over the alphabet $\mathscr{B}$.

Definition 1 (Bit Transducers). *A bit transducer is a tuple $\langle Q, i, \Delta \rangle$ where (1) Q is a finite set of states, (2) $i \in Q$ is an initial state, and (3) $\Delta : Q \times \mathscr{B} \times \mathscr{B} \to Q \times \mathscr{B}$ is a transition function.*

A bit transducer is a device that implements a process that transforms a bit string into another bit string, i.e. provides mappings $\mathscr{B}^n \to \mathscr{B}^n$ for all $n \in \mathbb{N}$. The transition function of a bit transducer maps a present control state, an input bit, and a random bit into a next control state and an output bit. Thus,



The transducer on the left has two states: a and b, a being the initial state. The arcs from the state symbols to ⊗–nodes denote inputs, the ⊗–nodes random choices, and the arcs from ⊗–nodes the outputs and the next control states. The dashed arcs correspond to the random bit 1 and the solid arcs to 0. When only a solid arc is drawn out of a ⊗–node, the behavior does not depend on the random bit source.

Fig. 1. The bit transducer **T**

a bit transducer has access to a random bit source and by this source implements a probabilistic process. In other words, bit transducers are nondeterministic systems. This is reasonable because tested systems in practice are often nondeterministic, for example due to environmental factors.

Example 1. In Fig. 1, we present a simple bit transducer that we use as an example throughout this paper. We give to this transducer the name **T**. As a mathematical object, **T** can be characterized by $\mathbf{T} = \langle\{a, b\}, a, \Delta\rangle$ where

$$\begin{aligned}\Delta = [&\langle a,0,0\rangle \mapsto \langle b,\overline{1}\rangle, \langle a,0,1\rangle \mapsto \langle b,\overline{0}\rangle, \langle a,1,0\rangle \mapsto \langle b,\overline{0}\rangle, \langle a,1,1\rangle \mapsto \langle b,\overline{0}\rangle,\\ &\langle b,0,0\rangle \mapsto \langle a,\overline{0}\rangle, \langle b,0,1\rangle \mapsto \langle a,\overline{0}\rangle, \langle b,1,0\rangle \mapsto \langle a,\overline{0}\rangle, \langle b,1,1\rangle \mapsto \langle b,\overline{1}\rangle]. \qquad (1)\end{aligned}$$

We put a line over all output bits throughout this paper. This line carries no mathematical meaning but is provided purely as a visual hint to the reader.

The following definition describes the operation of a bit transducer.

Definition 2 (Execution of Bit Transducers). *Let $I, R \in \mathscr{B}^n$ for $n \in \mathbb{N}$ be two bit strings of length n and let $B = \langle Q, i, \Delta\rangle$ be a bit transducer. The output of B on $\langle I, R\rangle$ is the bit string $O \in \mathscr{B}^n$ if and only if there exists a sequence of states $q_0, \ldots, q_n$ such that*

1. *$q_0 = i$, and*
2. *$\forall k \in \{1, \ldots, n\} : \Delta(\langle q_{k-1}, I[k], R[k]\rangle) = \langle q_k, O[k]\rangle$.*

(Here $I[k]$ denotes the kth element of the bit string I, $I[1]$ being the first element.)

The idea of the random bit source of a bit transducer is that it is not visible. Hence, we assume a uniform distribution of random bit strings, and get the following definition:

Definition 3 (Probability of Output). *Let $I \in \mathscr{B}^n$ and let B be a bit transducer. Then B is said to produce the output $O \in \mathscr{B}^n$ on the input I with the probability*

$$p = \frac{|\{R \in \mathscr{B}^n \mid O \text{ is the output of } B \text{ on } \langle I, R\rangle\}|}{2^n}. \qquad (2)$$

We denote this probability p by $\mathbf{P}(O|I, B)$.

Definition 4 (Traces). *A* trace *is a pair $\langle I, O\rangle$ where $I, O \in \mathscr{B}^n$ for some $n \geq 0$. For a bit transducer B,* **tr** B *denotes the traces of B, defined as*

$$\mathbf{tr}\, B = \{\langle I, O\rangle \mid \mathbf{P}(O|I, B) > 0\}. \tag{3}$$

Example 2. Consider the transducer **T** in Fig. 1. On the input 000, the transducer produces the output $\overline{101}$ with probability $\frac{1}{4}$ and the output $\overline{111}$ with probability 0. Because $2^{-3} = \frac{1}{8}$, there are two distinct random bit strings on which the output $\overline{101}$ is produced. These are 101 and 111; namely, the transition back from b to a on input 0 does not depend on the random bit; technically both bit values map to the same output bit and destination state (see Ex. 1).

Rationale for the System Model. Bit transducers are simple finite-state systems. Finite-state systems have been examined extensively in the literature of formal testing theory.

Relation to IOTSes. In particular, an injective morphism into the domain of IOTSes [16] can be provided easily, as shown next.

Definition 5 (IOTSes). *An IOTS is a tuple $\langle Q, i, \Sigma_I, \Sigma_O, \Delta\rangle$ where Q is a set of states, $i \in Q$ is an initial state, Σ_I is an input alphabet, Σ_O is an output alphabet, the alphabets are disjoint and do not contain τ, and $\Delta \subseteq Q \times (\Sigma_I \cup \Sigma_U \cup \{\tau\}) \times Q$ is a transition relation such that $\forall q \in Q, a \in \Sigma_I : \exists k \in \mathbb{N}, q_1, \ldots, q_k \in Q : \langle q, \tau, q_1\rangle \in \Delta, \langle q_1, \tau, q_2\rangle \in \Delta, \cdots, \langle q_{k-1}, a, q\rangle \in \Delta$.*

Definition 6 (Path Notation). *If q is a state of an IOTS $\langle Q, i, \Sigma_I, \Sigma_O, \Delta\rangle$ and $w = w_1 \cdots w_k \in (\Sigma_I \cup \Sigma_O \cup \{\tau\})^*$, we write $q \xrightarrow{w} q'$ to denote*

$$\exists q_0, \ldots, q_k : q = q_0 \wedge q_k = q' \wedge \forall i \in \{1, \ldots, k\} : \langle q_{i-1}, w_i, q_i\rangle \in \Delta. \tag{4}$$

Furthermore, we write $q \xrightarrow{w}$ to denote $\exists q' : q \xrightarrow{w} q'$ and finally $L \xrightarrow{w}$ as a shorthand for $i \xrightarrow{w}$ when i is the initial state of L.

A bit transducer B can be mapped to a similar (in a sense shown below) IOTS L. We provide an intensional characterization of this mapping. We mark by $B \approx L$ the proposed similarity, defined as follows:

Definition 7 (Similarity Between IOTSes and Bit Transducers). *A bit transducer $B = \langle Q, i, \Delta\rangle$ and an IOTS $L = \langle Q', i', \Sigma_I, \Sigma_O, \Gamma\rangle$ are similar, denoted by $B \approx L$, iff (1) $\Sigma_I = \{0, 1\}$, (2) $\Sigma_O = \{\overline{0}, \overline{1}\}$, (3) $Q' = Q \times \{W, 0, 1\}$, (4) $i' = \langle i, W\rangle$, and (5) $\langle\langle q, b\rangle, \alpha, \langle q', b'\rangle\rangle \in \Gamma$ iff either*

$$\alpha \in \{0, 1\} \wedge b = W \wedge b' = \alpha \wedge q = q', \tag{5}$$

$$\alpha \in \{0, 1\} \wedge b \in \{0, 1\} \wedge b' = b \wedge q = q', \quad \textit{or} \tag{6}$$

$$\alpha \in \{\overline{0}, \overline{1}\} \wedge b \in \{0, 1\} \wedge b' = W \wedge \exists r : \Delta(\langle q, b, r\rangle) = \langle q', \alpha\rangle\,. \tag{7}$$

It now follows that if $B \approx L$ and B produces output $\overline{b}_1 \cdots \overline{b}_n$ on input $b_1 \cdots b_n$, then $L \xrightarrow{b_1\overline{b}_1 \cdots b_n\overline{b}_n}$. Similarly, if $L \xrightarrow{b_1\overline{b}_1 \cdots b_n\overline{b}_n}$, then B produces output $\overline{b}_1 \cdots \overline{b}_n$ on the input $b_1 \cdots b_n$ (straightforward proofs omitted). Hence, B and L can be seen to be similar systems. Thus, bit transducers form, at least behaviorally, a subclass of IOTSes. This should guarantee that the system model in itself is realistic.

Nondeterministic and probabilistic [1] systems are commonly discussed, and appear also in the practice. Often the nondeterminism of a system under test is caused by environmental factors that are not under the test system's control.

In order to be able to analyse nondeterministic systems with probability theory, nondeterministic choice points must be associated with a probabilistic model. We have fixed the probability $\frac{1}{2}$ for every nondeterministic choice (bit transducers have only binary branchings). Another option would have been to allow every branch point to be associated with real-valued branching probabilities, or to consider a more mathematically challenging class of Markov models. These have been studied widely, also in the context of testing [17]. However, because we eventually argue that the anomaly that we wish to present here is not strongly tied to the system model itself, we feel that it is acceptable to stick with this simple model of probabilistic choice.

2.2 Specifications

Because bit transducers only map bit strings to bit strings, there is no built-in notion of correctness for them. Therefore, we need also a domain of functional requirements or system specifications with which implementations can be compared. We use bit transducers also as system specifications in the following manner:

Definition 8 (Correct Traces). *Let B be a bit transducer. A trace $\langle I, O\rangle$ is correct with respect to B iff $\langle I, O\rangle \in \mathbf{tr}\, B$, i.e. iff $\mathbf{P}(O|I, B) > 0$.*

In other words, it is correct to produce O on input I according to a specification (which is also a bit transducer) B if there exists at least one execution in B that produces O from I. This definition avoids a probabilistic definition of correctness, which may be debatable. But in the current testing practice, functional testing requirements are usually non-probabilistic, and functional black-box testing is a vividly researched and practiced art. There are forms of testing that include probabilistic analysis such as load and scalability testing, but we do not consider them here.

Example 3. Based on the observations of Ex. 2, the trace $\langle 000, \overline{101}\rangle$ is correct with respect to $\mathbf{T}$ but the trace $\langle 000, \overline{111}\rangle$ is not.

We can now define what is a correct implementation of a specification:

Definition 9 (Correct Implementations). *Let B_S be a bit transducer (a specification) and B_I a bit transducer (an implementation). B_I is correct with respect to B_S if and only if*

$$\mathbf{P}(O|I, B_I) > 0 \implies \langle I, O\rangle \textit{ is correct w.r.t. } B_S \tag{8}$$

which yields a reflexive and transitive conformance relation (a preorder) that can be written in the form

$$B_I \preceq_{\mathbf{tr}} B_S \Longleftrightarrow_{\text{def}} \mathbf{tr}\ B_I \subseteq \mathbf{tr}\ B_S. \tag{9}$$

In other words, an implementation may implement a subset of the behavior of its specification but not a superset.

Rationale for the Specification Model. First, the idea of using the same formalism for specifications and systems is not new, and has been extensively used e.g. in the study of the famous **ioco**-theory [15, 16].

The idea that specifications should not be probabilistic can be also traced down to this same theory. Namely, in **ioco** every trace is either valid or invalid, and no probabilistic interpretation of implementations is a part of the standard theory.

In what follows we will find that our specifications are in a sense non-probabilistic but implementations are probabilistic. We strongly disagree with a view that this would diminish the importance of our result. First, we have not evidence that the reported anomaly could not occur in the context of deterministic implementations – it can be enough to have a probabilistic fault model. Second, all real-world implementations of nondeterministic systems are in practice probabilistic in one way or another. At the same time, functional specifications are seen often as non-probabilistic, and test runs yield non-probabilistic verdicts (pass, fail). Hence, if the disrepancy between non-probabilistic specifications and probabilistic implementations would turn out to be the cause for the reported anomaly, it should be clearly identified as a topic for future research.

2.3 Fault Model

We have now functional requirements and implementations. What is missing is the concept of a fault model. We take the following general view: a fault model maps a specification to a probability distribution of implementations. As the concrete fault model we choose the following: when a specification is implemented, there is a uniform probability for every single output bit being unwantedly toggled. To simplify, we let this probability by $\frac{1}{2}$. It is unrealistically high, but because our whole analysis is eventually qualitative, nothing is lost. Thus the following definition:

Definition 10. *Let $B_S = \langle Q, i, \Delta \rangle$ be a bit transducer (functioning as a system specification). Then $B_I = \langle Q, i, \Gamma \rangle$ is a potential implementation of B_S if and only if for every $q \in Q$ and $i, r \in \mathscr{B}$ it holds that*

$$\exists o, o' \in \mathscr{B}, q' \in Q : \Delta(\langle q, i, r \rangle) = \langle q', o \rangle \wedge \Gamma(\langle q, i, r \rangle) = \langle q', o' \rangle\,. \tag{10}$$

In the fault model, the probability of every such a potential implementation of B_S is $2^{-4|Q|}$. We denote this probability by $\mathbf{P}(B_I|B_S)$, and denote the set of potential implementations ("mutants") of B_S by **impl** B_S*. To simplify notation, we define that $\mathbf{P}(B|B_S) = 0$ for all $B \notin$* **impl** B_S*.*

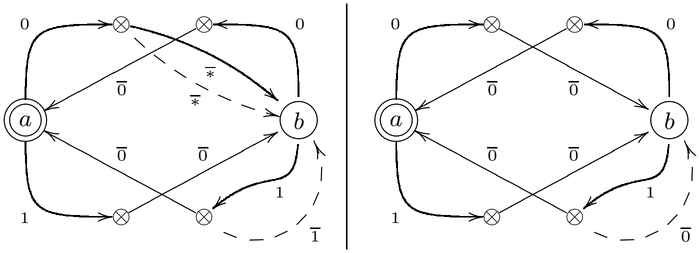


Fig. 2. Those potential implementations of **T** which are correct w.r.t. **T**. Four correct implementations follow the template on the left, where an instance of $\overline{*}$ can be replaced with either $\overline{0}$ or $\overline{1}$. One of these is the original specification **T** itself. The fifth correct implementation is the transducer on the right, which produces always a string containing only zeroes.

We are now ready to commence an analysis employing standard probability theory. First, we can calculate the a priori probability that a specification is correctly implemented:

Definition 11. *We denote by* $\mathbf{P}(\mathsf{correct}|B_S)$ *the a priori probability that* B_S *is correctly implemented, defined as*

$$\mathbf{P}(\mathsf{correct}|B_S) = \sum_{B_I : B_I \preceq_{\mathbf{tr}} B_S} \mathbf{P}(B_I|B_S) \tag{11}$$

Example 4. Consider the bit transducer **T**. There are 256 potential implementations of **T** ($2^{4\cdot 2}$); thus each one has a priori probability of 2^{-8}. Of these potential implementations, five are correct w.r.t. **T**. These are illustrated in Fig. 2.

Hence, the a priori probability for an implementation of **T** being correct is, under the given fault model, $\mathbf{P}(\mathsf{correct}|\mathbf{T}) = 5 \cdot 2^{-8} \approx 0.0195$.

Rationale for the Fault Model. According to Petrenko [10, 11], a fault model includes a specification, the conformance relation, and the set of possible implementations. It is not difficult to see that our approach is not fundamentally different, even though we have separated the conformance relation (9) from the fault model, and our fault model includes a known probability distribution on the implementations.

The structure of our particular fault model is that there is a certain, fixed probability on the mutation of a transition label. Thus the set of implementations in our fault model is finite. This general approach was used, for instance, as a basis for test derivation by Petrenko et al. [12]. The idea of a bit-flip fault is prevalent in the domain of hardware testing [14].

Another known fault model is to consider all finite state machines upto a certain size limit [4]. It can be argued that under such a fault model implementations bear no a priori link to their specifications – from some viewpoints an unrealistic assumption. It is unclear to us whether the presented anomaly can surface in the context of such fault models where there is no real link between implementations and specifications.

2.4 Analysis

We have now set the wheels in motion; probabilistic analysis will lead us to our goal. The idea is to compute the a posteriori probability that an implementation is correct, given a trace that has been observed to be produced by the implementation.

Thus, let B_S be a specification and let $\langle I, O\rangle$ be trace. If this is not correct w.r.t. B_S, then the probability that the implementation itself is correct must be zero. Hence, we consider now the case of a correct trace, which corresponds to the idea of a test run that has passed. Thus, assume $\langle I, O\rangle \in \mathbf{tr}\ B_S$.

The a priori probability for producing output O on the input I by an unknown implementation of B_S is given by

$$\mathbf{P}_{\text{spec}}(O|I, B_S) = \sum_{B_I} \mathbf{P}(B_I|B_S)\mathbf{P}(O|I, B_I). \tag{12}$$

Example 5. Consider the trace $\langle 0, 1\rangle$ which is correct w.r.t. $\mathbf{T}$. The a priori probability for this observation, given $\mathbf{T}$ as a specification, under the given fault model, is exactly $\frac{1}{2}$.

We have now calculated both the a priori probability for an implementation's correctness (11), as well as the a priori probability for a certain output on a certain input (12). What we need yet is the probability of an output given an input and the assumption that the implementation producing the output is in fact correct. This can be calculated as:

$$\mathbf{P}(O|I, B_S, \mathsf{correct}) = \frac{\sum_{B_I : B_I \preceq_{\mathbf{tr}} B_S} \mathbf{P}(B_I|B_S)\mathbf{P}(O|I, B_I)}{\sum_{B_I : B_I \preceq_{\mathbf{tr}} B_S} \mathbf{P}(B_I|B_S)} \tag{13}$$

where the denominator is actually $\mathbf{P}(\mathsf{correct}|B_S)$ (11).

Hence, we can invoke the Bayes' rule [6] and calculate the a posteriori probability that a particular implementation of the specification is actually correct. We denote this by $\mathbf{P}(\mathsf{correct}|O, I, B_S)$:

$$\mathbf{P}(\mathsf{correct}|O, I, B_S) = \frac{\mathbf{P}(\mathsf{correct}|B_S)\mathbf{P}(O|I, B_S, \mathsf{correct})}{\mathbf{P}_{\text{spec}}(O|I, B_S)}. \tag{14}$$

Example 6. Note that if $P(B_I|B_S)$ is constant for all B_I (all implementations are equiprobable, which is the case in our example), this can be simplified to

$$\frac{\sum_{B_I : B_I \preceq_{\mathbf{tr}} B_S} \mathbf{P}(O|I, B_I)}{\sum_{B_I} \mathbf{P}(O|I, B_I)}. \tag{15}$$

Example 7. Consider the trace $\left\langle 0, \overline{1}\right\rangle$ and the bit transducer $\mathbf{T}$. We know from Ex. 4 that an implementation of $\mathbf{T}$ is correct by a priori probability

$\mathbf{P}(\mathsf{correct}|\mathbf{T}) \approx 0.0195$. Furthermore, by Ex. 5 it is known that the trace $\langle 0, \overline{1}\rangle$ has a priori probability of $\mathbf{P}_{\mathrm{spec}}(\overline{1}|0, \mathbf{T}) = 0.5$ of being produced by an implementation of $\mathbf{T}$ (implementations distributed by the fault model).

Furthermore, we know that there are five correct potential implementations of $\mathbf{T}$ (see Fig. 2). We calculate the probability $\mathbf{P}(\overline{1}|0, \mathbf{T}, \mathsf{correct})$ as

$$\mathbf{P}(\overline{1}|0, \mathbf{T}, \mathsf{correct}) = \frac{2^{-8} \cdot (0 + 0.5 + 0.5 + 1 + 0)}{5 \cdot 2^{-8}} = 0.4. \tag{16}$$

Hence, we can invoke (14), obtaining

$$\mathbf{P}(\mathsf{correct}|\overline{1}, 0, \mathbf{T}) \approx \frac{0.0195 \cdot 0.4}{0.5} = 0.0156. \tag{17}$$

But note that this probability is less than 0.0195, the a priori probability for the implementation's correctness (which corresponds also to the a posteriori correctness probability after empty input and output – it is naturally the same).

The same effect occurs also with longer inputs and outputs, for example $\langle 001100, \overline{100100}\rangle$ is correct w.r.t. $\mathbf{T}$, yet

$$\mathbf{P}(\mathsf{correct}|\overline{100100}, 001100, \mathbf{T}) \approx 0.083 \tag{18}$$

which is significantly less than

$$\mathbf{P}(\mathsf{correct}|\overline{10010}, 00110, \mathbf{T}) \approx 0.125. \tag{19}$$

Example 8. Let us now consider the infinite sequence of inputs and outputs

$$i_1 = 01, o_1 = \overline{10}, i_2 = 0101, o_2 = \overline{1010}, i_3 = 010101, o_3 = \overline{101010}, \ldots \tag{20}$$

For all $k > 0$, let A_k denote the probability $\mathbf{P}(o_k|i_k, \mathbf{T})$ and C_k the probability $\mathbf{P}(o_k|i_k, \mathbf{T}, \mathsf{correct})$.

At the initial state a, correct mutants will produce the output $\overline{1}$ on the input 0 with the probability $\frac{2}{5}$. There are only three correct mutants that are capable of producing $\overline{1}$ first; all those mutants will produce $\overline{0}$ with the probability $\frac{1}{2}$ on the input 1 in state b, returning back to state a. On the third input bit, which is 0, there are three correct mutants still possible. Of these, the one having $\overline{1}$ on both of the transitions from a on input 0 (call it M_1) is now relatively as probable as the other two together. Hence, the probability that $\overline{1}$ will be now outputted in response to the input 0 is $\frac{3}{4}$. After two "rounds", the Bayesian probability of the mutant M_1 is twice as high as that of the other two; hence the probability of the output bit $\overline{1}$ is $\frac{2}{3} + \frac{1}{3} \cdot \frac{1}{2} = \frac{5}{6}$. On the next round, M_1 is four times as probable as the remaining mutants, then eight times, and so on. This yields the formula

$$C_k = \frac{4}{10 \cdot 2^k} \prod_{2 \le i \le k} \left(\frac{2^{i-2} + \frac{1}{2}}{2^{i-2} + 1} \right) \qquad k > 1 \tag{21}$$

for the probabilities C_k. The analysis for A_k involves all the mutants and is slightly more complex. We give only the resulting formula, which we have verified also by numerical computation. Denoting

$$\alpha(n) = \frac{1}{4} \prod_{2 \le i \le n} \frac{2^{i-2} + \frac{1}{2}}{2(2^{i-2}+1)}, \qquad \beta(n) = \frac{1}{4} \left(\prod_{2 \le i \le n} \frac{2^{i-2} + \frac{1}{2}}{2^{i-2}+1} \right)^2 \tag{22}$$

we have

$$A_k = \frac{1}{4} \left[3\alpha(k) + \alpha(1)\beta(k-1) + \left(\sum_{1 \le i < k} \alpha(i)\beta(k-i) \right) \right] \qquad k > 1. \tag{23}$$

If we now denote the product over i in (21) by K, we get

$$C_k = \frac{4}{10 \cdot 2^k} K, \qquad A_k = \frac{1}{4} \left[\frac{3}{4} \frac{1}{2^{k-1}} K + \frac{1}{16} K^2 \left(\frac{2^{k-2}+1}{2^{k-2}+\frac{1}{2}} \right)^2 + R \right] \tag{24}$$

where $R \ge 0$. Hence we get

$$\frac{C_k}{A_k} \le \frac{16}{10 \left[\frac{3}{2} + \frac{1}{16} \left(\frac{6}{4}\right)^k \left(\frac{2^{k-2}+1}{2^{k-2}+\frac{1}{2}} \right)^2 \right]} \qquad k > 1 \tag{25}$$

where we have applied the easily verifiable fact that for $k > 1$, $K > \left(\frac{3}{4}\right)^k$. Now the denominator on the right grows without limits when $k \to \infty$. Hence $\lim_{k \to \infty} C_k / A_k = 0$, and

$$\lim_{k \to \infty} \mathbf{P}(\mathsf{correct} | o_k, i_k, \mathbf{T}) = \lim_{k \to \infty} \frac{\mathbf{P}(\mathsf{correct} | \mathbf{T}) C_k}{A_k} = \frac{5}{256} \lim_{k \to \infty} \frac{C_k}{A_k} = 0. \tag{26}$$

Structural Explanation of the Phenomenon. This effect is caused ultimately by the structure of $\mathbf{T}$. The fifth mutant of $\mathbf{T}$ that produces always a sequence of zeroes adds extra probability mass on the answer $\overline{0}$ for the input 0. The mutation of the transition $\langle b, 1, 0 \rangle \mapsto \langle b, \overline{1} \rangle$ to $\langle b, 1, 0 \rangle \mapsto \langle b, \overline{0} \rangle$ alone creates an invalid mutant: this mutant is capable of producing the trace $\langle 1110, \overline{0001} \rangle$, which is not legal according to the specification $\mathbf{T}$. However, combined with the mutation from $\langle a, 0, 0 \rangle \mapsto \langle b, \overline{1} \rangle$ to $\langle a, 0, 0 \rangle \mapsto \langle b, \overline{0} \rangle$, a valid mutant results. Therefore, the set of valid mutants overemphasizes the answer $\overline{0}$ for the input 0, as mentioned. Hence, observing $\overline{1}$ after the input 0 suggests an invalid implementation, even though the response in itself is valid.

This explains why the trace $\langle 0, \overline{1} \rangle$ yields a negative effect on the a posteriori probability that the system under test is correct. The analysis of the infinite sequence (20) is complicated by the need to take into account also the effect of those mutants that take the self-loop transition at b with mutated output bit ($\overline{0}$), but the basic idea remains the same.

We can wrap up the previous discussion in the following theorem about our bit transducers and our failure model, which is a description of the anomaly itself in this context:

Theorem 1 (Main Result). *There exists a bit transducer B, and an infinite sequence of inputs $i_1 \prec i_2 \prec \cdots$ ($\prec$ is the strict prefix relation) and an infinite sequence of outputs $o_1 \prec o_2 \prec \cdots$, such that for all k, $\langle i_k, o_k\rangle$ is correct w.r.t. B and $\mathbf{P}(\mathsf{correct}|o_k, i_k, B) < \mathbf{P}(\mathsf{correct}|o_{k-1}, i_{k-1}, B)$ (the probability of a correct implementation decreases). Furthermore, the probability $\mathbf{P}(\mathsf{correct}|o_k, i_k, B)$ approaches zero in the limit $k \to \infty$.*

Proof. Let B be $\mathbf{T}$ and see Ex. 8, where such a sequence has been constructed.

3 Discussion

We defined a universe of system models (Def. 1), bit transducers, which are nondeterministic finite-state machines processing bits. We decided that – for the purpose of black-box testing bit transducers – systems could be specified as bit transducers also. Our conformance relation was based on a standard view of observational containment: a system conforms to its specification if and only if its behavior (the set of producible traces) is a subset of the behavior of its specification. (Note that all bit transducers are always ready to accept all inputs; hence, there are no partial specifications and thus it is correct to use the subset relation.)

We then defined a fault model, which described the potential implementations for a given specification. Our fault model was probabilistic but attached a uniform probability to all transition label mutations so that, in fact, all possible implementations were equiprobable.

We then showed that there exists a simple bit transducer, which we named $\mathbf{T}$, such that testing a black-box implementation of $\mathbf{T}$, there exists an infinite sequence of longer and longer test runs such that (1) the test runs are correct with respect to the conformance relation we defined, but (2) the probability-theoretic risk of actually having an *invalid* implementation increases and approaches one in the limit. (This was the essence of the concluding Theorem 1 – the particular sequence was discussed in Ex. 8.)

The resulting anomaly is that running one more test run that yields “pass” (only conforming behavior has been observed) actually increases the risk of having a malfunctioning system. This means that from the viewpoint of the expected utility theory [5], the utility of the system under test has decreased (if we attach a negative utility on the case of having a malfunctioning system). Hence, a test run ending with “pass” produced negative value. This is counterintuitive; yet, the result is a mathematical fact, derived via universally accepted theories about probability and rational decision [6, 5].

3.1 Underlying Assumptions

Let us now attempt to enumerate the underlying assumptions behind our result.

We assumed that *conformance is a set-theoretic relation* – an implementation is either conformant or not, but it cannot be “maybe-conforming” (which would be a view towards fuzzy logic). Having a set-theoretic binary conformance relation is common, and is e.g. the case in the **ioco**-theory [16].

We assumed that *every individiual observation is either correct (conforming) or not.* Again, this follows the thinking in the **ioco**-theory. This is also reflected in practice: for example, the principal verdicts available in TTCN-3 are "pass" and "fail" [2].

We assumed that *implementations can be nondeterministic.* We claim this is reasonable, because real-world systems tend to be nondeterministic, although the reasons vary: some systems make explicit calls to random number generation; some systems are nondeterministic because underlying services (e.g. operating systems) are nondeterministic from the viewpoint of the system itself (e.g. thread scheduling); and importantly, many systems are nondeterministic from testing point of view due to environmental factors that are not accounted for in testing (e.g. points of control and observation not visible to a test harness).

We assumed that *it is possible to attach probabilities on nondeterministic branches.* We believe this makes sense, because by the law of large numbers, unrelated phenomena eventually become stochastically modelable.

We assumed that there *exists a probability distribution of multiple potential implementations of a system specification, and that the specification affects the structure of the distribution.* The fact that there exists a probability distribution is probably uncontestable (what is true is that the distribution is usually unknown to us because the development of a particular system is not an experiment repeated many times – but this has not precluded the use of hypothetical fault models as analytical devices).

We claim that the resulting anomaly is mostly a direct consequence of these assumptions. Next, we describe why we believe this is the case.

3.2 Arguments for and Against

We believe that this anomaly is ultimately caused by the commonly agreed view that observed behavior is either acceptable or not. This causes loss of information. An acceptable behavior is a behavior that could have been produced by at least one potential implementation that has a non-zero probability and that is correct with respect to the system specification. Thus, there is no distinction between behaviors that have different likelihoods of being produced by correct systems; in common thought, all behaviors with non-zero such a likelihood are collected into the same opaque class of conforming behaviors.

If the "pass" verdict is seen as meaning "there is yet a non-zero probability of having a correct system", and nothing else, there is in fact no anomaly. However, people tend to reason like this: "if I test more, there is a higher probability of finding a defect; hence, if no defect is found, I can trust the system more because I have tested it more". Our results show that the last part of this sentence is not mathematically true – only the first part is.

An observation can increase – from the viewpoint of Bayesian deduction – the probability that the system under test is in fact one of the faulty implementations; yet the observation cannot necessarily be used to prove that beyond doubt. In all fault models where this is true, the presented anomaly can exist.

One of the main contributions of this note is that we have constructed *a realistic system model and a realistic fault model within which this anomaly has been described and reported in detail.*

We can also imagine arguments against this note. Let us attempt to counter some of these hypothetical objections.

"It is not an anomaly, because it is based on a calculation that yields no contradiction." Certainly, the calculation itself is not contradictory, in the same sense as mathematics are not self-contradictory. The anomaly lies in that the theory of test observations and verdicts is not completely aligned with the theory of rational decision. Yet, rational decision theory is closely linked with the notions of risk and confidence, and risk and confidence is what testing is all about.

"There is no anomaly, because the probability of producing the presented sequence of inputs and outputs approaches zero when the length of the sequence increases." It is true that the elements of the sequence (20) become less and less probable when executed against any of the valid mutants. However, every element has a positive probability, and in practice testing must stop after a certain number of steps. At this point, there has been a positive probability of producing a prefix of the sequence. The quantitative value of that probability is not important, because it depends on the chosen system and fault models.

"The anomaly is caused by having a non-probabilistic specification and a probabilistic implementation." This claim could have truth in it – more research is needed to find out if the same result can be produced with strictly deterministic systems. However, what is very important is that the **ioco** testing theory is about non-probabilistic specifications; yet it can be used to test nondeterministic (i.e. probabilistic) black boxes. Hence, this kind of an argument would not be targeted towards this note alone, but a large body of current testing theory. (Of course, it could be worthfile to present that argument anyway.)

3.3 Avoiding the Anomaly

In this section we present some conceivable ways to remedy this anomaly.

Probabilistic conformance. One way to avoid the anomaly would be to dispense with the idea of a binary set-theoretic conformance relation, and to replace it with a probabilistic one. In practice this would mean also dispensing with the traditional verdict "pass". The verdict "fail" could remain, because it corresponds to the impossibility of having a correct implementation (a defect has been surely detected).

Controlled fault models. Another option would be to consider not all fault models but only those fault models where the anomaly does not surface. For example, we conjecture that this anomaly is not present in general in fault models where there is no relationship between specifications and implementations, but the distribution of possible implementations is the same for all specifications. We leave this as a subject for further study.

Stochastic view on the whole testing process. The anomaly presented here is that a certain test sequence yields negative contribution to confidence in the correctness of the system under test, yet the sequence in itself is correct according to the specification. However, it is plausible that in the example we presented, the *expected* effect on the confidence over all different test input sequences could be actually uniformly positive. Taking this view, one could argument that the anomaly is not important because it disappears “in the long run”. We leave this also as a subject for further study.

3.4 Pragmatic Consequences

One last thing to discuss is whether the presented result can contribute in any way to the practice of black-box testing – especially with formal methods.

It is quite clear that the calculations in this paper are practically impossible to be carried out on a system of any realistic size. Hence, we can reasonably assume that the current practices (pass & fail, set-theoretic conformance relations) will continue their happy life. In the context of this assumption, the following insights can be seen as practical consequences of our result.

Specifications that allow for multiple correct behaviors cause problems. Our analysis confirms the understanding that a system under test should be as little nondeterministic as possible: more control over the environment of an SUT and more strict specifications result in more efficient testing.

There can be really bad testing strategies. The testing strategy constructed in this paper yields in the limit probability 0 for the correctness of a black box that is still actually correct! What this tells most about is perhaps the constructed strategy itself. It is a testing strategy that destroyes value (in terms of increasing risk) with a positive probability. The pragmatic implication is that the quality of testing strategies can wary wildly, and that a way to assess their quality in quantitative terms can prove out to be a powerful test generation heuristic.

4 Conclusions and Future Work

In this note, we described an anomaly in the relationship between the mathematical definition of confidence and the common thinking about black-box testing: it is possible to execute tests that get the label “pass”, yet the confidence in the system under test decreases. We claimed that this is a general phenomenon and provided analysis of the result from multiple perspectives.

Many questions have been left open by this note. We suggest these as topics for future research: Is it possible to find an example of the phenomenon in the context of transfer (as opposed to output) faults? Does this anomaly exist in the context of strictly deterministic implementations? Can the anomaly be shown to exist only in fault models where implementation probabilities depend on the specification? Does the anomaly disappear if only the expected increase of confidence over all uniformly distributed test input and the related output sequences is considered? Can a deeper analysis of the phenomenon lead to advances in test design heuristics?

Acknowledgements are due to the anonymous referees for both FATES ’03 and FATES ’05.

References

[1] R. Alur, C. Courcoubetis, and M. Yannakakis. Distinguishing tests for nondeterministic and probabilistic machines. In *Proc. 27th ACM Symposium on Theory of Computing*, pages 363–372, 1995.

[2] European Telecommunications Standards Institute (ETSI). The testing and test control notation version 3; part 1: TTCN-3 core language. ETSI ES 201 873-1, V2.2.1, Feb. 2003.

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

[4] G. Luo, G. von Bochmann, and A. Petrenko. Test selection based on communicating nondeterministic finite state machines using a generalized wp-method. *IEEE Transactions on Software Engineering*, SE-20(2):149–162, 1994.

[5] A. Mas-Colell, M. D. Whinston, and J. R. Green. *Microeconomic Theory.* Oxford University Press, 1995.

[6] J. S. Milton and J. C. Arnold. *Introduction to probability and statistics.* McGraw–Hill, 1995.

[7] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology*, 1(1):5–20, Jan. 1992.

[8] A. J. Offutt and S. Lee. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, May 1994.

[9] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall PTR, 1981.

[10] A. Petrenko. Fault model-driven test derivation from finite state models: Annotated bibliography. In F. Cassez, C. Jard, B. Rozoy, and M. D. Ryan, editors, *MOVEP*, volume 2067 of *Lecture Notes in Computer Science*, pages 196–205. Springer, 2000.

[11] A. Petrenko, N. Yevtushenko, and G. V. Bochman. Fault models for testing in context. In *FORTE '96*, pages 163–178, 1996.

[12] A. Petrenko, N. Yevtushenko, and J. L. Huo. Testing transition systems with input and output tester. In *TestCom 2003*. Springer-Verlag, 2003.

[13] T. Pyhälä and K. Heljanko. Specification coverage aided test selection. In J. Lilius, F. Balarin, and R. J. Machado, editors, *Proceeding of the 3rd International Conference on Application of Concurrency to System Design (ACSD'2003)*, pages 187–195, Guimaraes, Portugal, June 2003. IEEE Computer Society.

[14] M. Rimen, J. Ohlsson, and J. Torin. On microprocessor error behavior modeling. In *24th IEEE International Symposium on Fault-Tolerant Computing*, 1994.

[15] J. Tretmans. A formal approach to conformance testing. In *Proc. 6th International Workshop on Protocols Test Systems*, number C-19 in IFIP Transactions, pages 257–276, 1994.

[16] M. van der Bijl, A. Rensink, and J. Tretmans. Compositional testing with IOCO. In *Formal Approaches to Software Testing: Third Internal Workshop, FATES 2003*, pages 86–100, 2004.

[17] J. Whittaker and M. Thomason. A markov chain model for statistical software testing. *IEEE Transactions on Software Engineering*, 20(10):812–824, Oct. 1994.

A Novel Test Coverage Metric for Concurrently-Accessed Software Components
(A Work-in-Progress Paper)

Serdar Tasiran, Tayfun Elmas, Guven Bolukbasi, and M. Erkan Keremoglu

Koç University, Istanbul, Turkey
{stasiran, telmas, gbolukbasi, mkeremoglu}@ku.edu.tr

Abstract. We propose a novel, practical coverage metric called "location pairs" (LP) for concurrently-accessed software components. The LP metric captures well common concurrency errors that lead to atomicity or refinement violations. We describe a software tool for measuring LP coverage and outline an inexpensive application of predicate abstraction and model checking for ruling out infeasible coverage targets.

1 Introduction

Verification and testing of concurrently-accessed software components is particularly challenging because the interleaving of concurrently executed threads compounds the program state space. Validation methods must both store information and have control of thread scheduling. As a result, exhaustive testing or model checking are prohibitively costly for realistic configurations of industrial-scale concurrent programs. This motivates the study of methods that combine verification and testing approaches in order to strike a compromise between computational cost and exhaustiveness.

We are exploring several hybrid techniques in which coverage metrics serve as the link between model checking and testing tools and enable us to (i) quantify the adequacy testing/verification performed, (ii) communicate partial results and testing/verification goals between tools, (iii) direct testing effort towards unexplored, quantitatively distinct executions of a program that are interesting for a particular purpose.

In this paper, we propose a coverage metric, called "location pairs" (LP) focused on the concurrency aspects of test executions. The LP metric was inspired by a pattern common to atomicity and refinement violations that were found in industrial software examples and in the literature [2, 3, 5]. These concurrency errors are triggered whenever an instance of the following scenario takes place: A thread $\mathbf{t}_1$ executes a particular line of code ln_1, then a context-switch occurs and another thread $\mathbf{t}_2$ starts execution at another line of code ln_2. This sequence of events, and the existence of a dependency between $\mathbf{t}_1$'s execution of the block of code preceding ln_1 and $\mathbf{t}_2$'s execution of the block of code following ln_2 guarantee that the error will occur regardless of the program state and other concurrent threads. This scenario is different from a race condition: $\mathbf{t}_1$ and $\mathbf{t}_2$ may protect

W. Grieskamp and C. Weise (Eds.): FATES 2005, LNCS 3997, pp. 62–71, 2006.

all global variables they access using the proper locks, but the interleaving described may still cause an atomicity or refinement violation. The LP metric is associated with higher-level concurrency errors similar to those in [1, 2, 5].

The fact that the LP metric seems to correspond well with certain types of concurrency errors makes it a promising tool for guiding manual validation effort and catching errors beyond the reach of methods based on state-space traversal. A particular concurrency error may not be possible with small program states or few threads because certain conflicts or resource contention are required to trigger it. It may also be unlikely because it requires the external events to be timed and threads interleaved a certain way. The LP metric captures errors caused by such unforeseen interleavings. The oversight typically occurs because of erroneous assumptions about the environment or the belief that a synchronization mechanism makes a certain interleaving impossible. Given an unexamined but apparently reachable location pair as a target, the programmer can reason about conditions that need to be set up to reach the target or provide the coverage analysis tool hints that help it prove that the target location pair is unreachable. In this way, the LP metric helps test writers explore qualitatively distinct and error-prone scenarios. Further, if they believe a certain scenario is not possible, it provides a tool for them to make explicit and check the justification for their belief.

The use of the LP metric also makes possible practical automatic techniques to be provided for the tasks of measuring coverage, ruling out infeasible coverage targets, and providing abstract traces to help with test input generation. We are developing a software tool that measures test coverage according to the LP metric for Java programs. We also propose the lightweight use of formal verification tools (a combination of predicate abstraction and model checking) to rule out an efficiently-computable set of unreachable location pairs. By avoiding exhaustive exploration of possible program states and thread interleavings and precise determination of reachable location pairs, we keep the computational burden of our method low. Since the metric involves consideration of pairs of method bodies, it does not lead to a combinatorial blow-up in the number of coverage targets.

Section 2 presents preliminaries required to state our coverage metric precisely. A motivating example from Java class libraries is provided in Section 3. Section 4 describes the LP coverage metric. Section 5 outlines the method used to compute a reduced set of coverage targets. Section 6 describes how the proposed metric successfully captures the concurrency errors in the examples studied. Section 7 outlines our future research.

2 Preliminaries

2.1 Concurrent Programs: Syntax

In this paper, we focus on realizations of concurrently accessible data structures written in object-oriented languages. For ease of exposition, we use a simple

P	::=	*defn** e	(program)
defn	::=	*class cn body*	*(class declaration)*
body	::=	*extends* c { *field** *method** }	*(class body)*
field	::=	*t fn* = *e*	*(field declaration)*
method	::=	*t mn(arg***)* {*e*}	*(method declaration)*
arg	::=	*t x*	*(variable declaration)*
s,t	::=	*c* | *int* | *boolean* | ...	*(type)*
c	::=	*cn* | *Object*	*(class type)*
e	::=	*new c*	*(allocate***)*
		| *x*	*(variable***)*
		| *e.fd*	*(field access***)*
		| *e.fd:=e*	*(field assignment***)*
		| $e == e \mid e < e$	*(comparison expression***)*
		| $e \vee e \mid e \wedge e \mid \neg\ e$	*(boolean expression***)*
		| *e.mn(e***)*	*(method call)*
		| *synchronized e in e*	*(synchronization)*
		| *fork e*	*(fork)*
		| `if`(`Bexp`?s : s)	*(if statement)*
		| `while`(`Bexp`, s)	*(while statement)*
		| `pure`(s)	*(purity annotation)*
		| `atomic`(s)	*(atomicity annotation)*
cn	$\in$	*ClassNames*	
fn	$\in$	*FieldNames*	
mn	$\in$	*MethodNames*	
x,y	$\in$	*VariableNames*	

Fig. 1. The grammar for CJ

language that we call CJ. The syntax of CJ is given in Fig. 1. A data structure $\mathcal{D}$ is an instantiation of a CJ class $\mathcal{C}$. The set of $\mathcal{C}$'s methods are denoted by $\mathcal{M}_{\mathcal{C}}$.

`atomic`(s) marks s as an *atomic* statement block: a sequence of actions, which can be shown to be atomic by using commutativity and reduction arguments. `pure`(s) marks s as a *pure* statement block as defined in [4]. Roughly speaking, a pure block does not modify the program state unless it terminates exceptionally. The concepts of purity and atomicity are used in defining the LP metric in order to reduce the number of qualitatively distinct scenarios that need to be explored.

2.2 Concurrent Programs: Semantics

The semantics of a data structure written as a CJ class is given by a state transition graph. Each *state* of the program is a unique assignment of values to program variables. *Global variables* $\mathcal{V}_G$ are variables in the representation of $\mathcal{D}$ that are accessible by all methods of $\mathcal{C}$. Each multi-threaded execution also uniquely defines a set of *local variables* $\mathcal{V}_L$ which are variables accessible by an individual thread only. Local variables correspond to method-local variables in CJ.

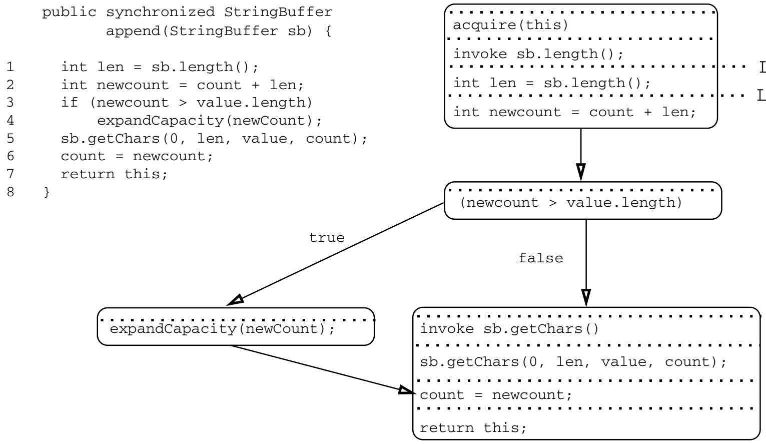
Each state transition corresponds to an atomic update of a program variable, called an *action*. For ease of exposition, we assume that the types of expressions shown with an asterisk (*) in Fig. 1 are executed atomically as well as method invocations, returns, lock acquisitions and releases.

Control Flow Graphs: With each method $\mu \in \mathcal{M}$ we associate a control flow graph CFG_μ obtained from the CJ code for the method. A control flow graph $CFG_\mu = \langle V, E, \lambda_v, \lambda_e \rangle$ is a directed graph. Vertices of a CFG are partitioned into two: $V = V_{ctrl} \cup V_{exec}$, where V_{ctrl} is the set of *branching vertices* corresponding to "if" and "while" statements, and V_{exec} is the set of *execution vertices*. Each branching vertex $v \in V_{ctrl}$ is labeled with a single atomically-evaluated expression $\lambda_v(v)$. The two outgoing edges representing the two branches are labeled by the corresponding boolean value of $\lambda_v(v)$. Each execution vertex $v \in V_{exec}$ is labeled by a sequence of actions $\lambda_e(v)$.

A *location* in the CFG_μ is like a program counter – it indicates at what point of the code the execution is. More precisely, a CFG_μ has associated with it a set of locations L_μ where each $l \in L_\mu$ is identified by either a pair of actions (α_i, α_{i+1}) where α_i and α_{i+1} are two consecutive actions in the label of a vertex in the CFG, or represents the entry point of a vertex in the CFG and corresponds to the case where execution of actions labeling that vertex has not started yet. The action immediately following a location l is denoted by $\alpha(l)$.

3 Motivating Example: `java.lang.StringBuffer`

The control flow graph for the `append` method of an older version of `StringBuffer` is given in Fig. 2. This version of the `append` implementation has a concurrency error due to the fact that the method argument, `sb`, is not locked throughout the method [5].



```
public synchronized StringBuffer
       append(StringBuffer sb) {

  int len = sb.length();
  int newcount = count + len;
  if (newcount > value.length)
      expandCapacity(newCount);
  sb.getChars(0, len, value, count);
  count = newcount;
  return this;
}
```

Fig. 2. The CFG for `StringBuffer.append()`

During an execution of `append` by a thread $\mathtt{t}_1$, the state of `sb` can be modified by another thread $\mathtt{t}_2$ between the invocation of the (synchronized) method `sb.length()` in line 1 of the `append` method and the invocation of `sb.getChars()` in line 5. For example, if `sb`'s length is more than 0 and $\mathtt{t}_2$ executes `sb.setLength(0)` between these two invocations, an atomicity violation occurs. `setLength(0)` invokes the following line

```
count = newLength;
```

where the value of `newLength` is 0. Let "`L3 -> L4`" denote this action in `setLength()`. Since this action occurs before `sb`'s contents are appended to `this`, in fact, an exception is thrown during `getChars`.

Observe that this error occurs whenever line 1 of `append` is followed by the line `count = newLength;` in `sb.setLength`. We found that this pattern of two particular consecutive dependent actions from two method bodies explains all atomicity and refinement violations we encountered. The LP metric makes precise and captures this intuition in the form of a coverage metric. Since the metric requires pairwise consideration of method bodies, it does not lead to a combinatorial blow-up in the number of coverage targets.

4 The "Location Pairs" Coverage Metric

We formulate our coverage metric as a requirement that certain states and certain transitions of a set of coverage finite-state machines (FSM) be traversed during test executions of the program. A coverage FSM $\mathcal{F}^{1,2}$ is defined for each pair of methods (μ_1, μ_2). A fragment of the coverage FSM for with μ_1 chosen to be `StringBuffer.append()` and μ_2 = `setLength()` is given in Fig. 3. The `L3 -> L4` transition represents the line `count = newLength` in `setLength()`. The states and transitions of the coverage FSM $\mathcal{F}^{1,2}$ are described below.

A state s of $\mathcal{F}^{1,2}$ is a tuple $s = \langle l^1, \mathtt{pend}^1, l^2, \mathtt{pend}^2, \mathtt{depdt}\rangle$ where

- l^1 and l^2 are locations in the CFG's of μ_1 and μ_2, respectively,
- `depdt` is a boolean variable indicating whether the pair of actions $\alpha(l^1)$ and $\alpha(l^2)$ immediately following locations l^1 and l^2 are *dependent* as defined

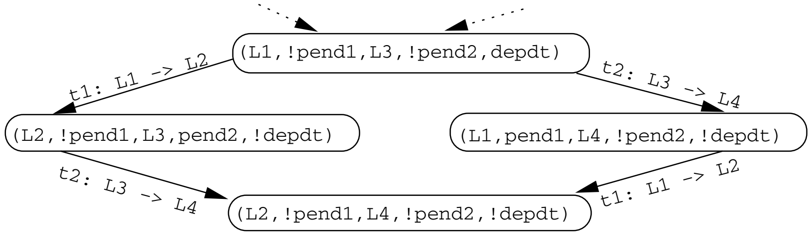

(L1,!pend1,L3,!pend2,depdt)
t1: L1 -> L2
t2: L3 -> L4
(L2,!pend1,L3,pend2,!depdt)
(L1,pend1,L4,!pend2,!depdt)
t2: L3 -> L4
t1: L1 -> L2
(L2,!pend1,L4,!pend2,!depdt)

Fig. 3. A fragment of the coverage state machine for a concurrent execution of `StringBuffer.append()` and `StringBuffer.setLength()`

in [9], i.e., at least one of them modifies a variable that the other one reads or writes a different value to, and
– $\texttt{pend}^1$ (respectively, $\texttt{pend}^2$) is a boolean variable that has the value `true` iff the current coverage state was reached by taking an action in μ_1 (respectively, μ_2) from a previous coverage state where `depdt` was `true`.

There is a transition in the coverage FSM whenever one location can be followed by another. This will be reduced later to approximate what can happen. There is a transition in the coverage FSM from state p to state q

$$p = \langle l_p^1, \texttt{pend}_p^1, l_p^2, \texttt{pend}_p^2, \texttt{depdt}_p \rangle \longrightarrow q = \langle l_q^1, \texttt{pend}_q^1, l_q^2, \texttt{pend}_q^2, \texttt{depdt}_q \rangle$$

iff $\texttt{depdt}_q$ is a legal value for whether actions $\alpha(l_q^1)$ and $\alpha(l_q^2)$ are dependent, and one of the followings holds:

(i) the execution of method μ_1 can move from l_p^1 to l_q^1, and $l_p^2 = l_q^2$, or
(ii) the execution of method μ_2 can move from l_p^2 to l_q^2, and $l_p^1 = l_q^1$.

The $\texttt{pend}^1$ and $\texttt{pend}^2$ bits are updated as follows:

– If $\texttt{depdt}_p$ is `false` then both $\texttt{pend}_q^1$ and $\texttt{pend}_q^2$ are assigned to `false`.
– If $\texttt{depdt}_p$ is `true` and case (i) above was applied, then $\texttt{pend}_q^2$ is assigned to `true` and $\texttt{pend}_p^1$ is assigned to `false`.
– If $\texttt{depdt}_p$ is `true` and case (ii) above was applied, then $\texttt{pend}_q^1$ is assigned to `true` and $\texttt{pend}_p^2$ is assigned to `false`.

Our coverage metric requires that all reachable transitions of the coverage FSM of the following two forms be traversed:

$$p = \langle l_p^1, \texttt{true}, l^2, \texttt{pend}_p^2, \texttt{depdt}_p \rangle \xrightarrow{\alpha(l^1)} q = \langle l_q^1, \texttt{pend}_q^1, l^2, \texttt{pend}_q^2, \texttt{depdt}_q \rangle$$

$$p = \langle l^1, \texttt{pend}_p^1, l_p^2, \texttt{true}, \texttt{depdt}_p \rangle \xrightarrow{\alpha(l^2)} q = \langle l^1, \texttt{pend}_q^1, l_q^2, \texttt{pend}_q^2, \texttt{depdt}_q \rangle$$

During an execution of a multi-threaded program, several different pairs of threads can be in the process of executing μ_1 and μ_2 concurrently. Each such pair of threads is at a particular, possibly different state of the coverage FSM $\mathcal{F}^{1,2}$.

The following reductions are used to obtain a more compact CFG and fewer locations from a method without eliminating any interesting interleavings:

- We lump together a basic block (i.e. uninterrupted by branching statements) of *method-local actions* – actions that do not modify any global variables, and consider it as a single action. Any method-local action is independent from any other action by another thread, therefore, its interleavings do not produce qualitatively distinct scenarios.
- A statement block s marked as "pure" ($\texttt{pure}(s)$) is partitioned into a control flow graph of atomic actions as described above. A pure execution of such a block is interpreted as no action having been taken at all, since a pure execution does not modify any global variables (see [4]). This results in

an important reduction in the number of target interleavings, since pure executions typically perform only lock acquisitions and releases and without this optimization, they can lead to a large number of possible, equivalent interleavings.

- During coverage analysis, we model atomic executions of lock-protected blocks marked `atomic`$(\alpha_s; \ldots ; \alpha_t)$ as a pair of consecutive actions $(\alpha^{\llbracket}, \alpha^{\rrbracket})$. $\alpha^{\llbracket}$ is an aggregate action that acquires all the locks in the beginning of the atomic block and $\alpha^{\rrbracket}$ releases the locks acquired previously as well as achieving the composed effect of $\alpha_s, \ldots, \alpha_t$. This is done in order to incorporate into the model a possible violation of the claimed atomicity: If a thread gets interleaved in between the pair of actions and modifies a global variable that was supposed to have been protected by the locks of the atomic block, an error is signaled.

Even though applying the reductions above yields a smaller, more usable coverage FSM, an exact determination of the reachable set of states and transitions of $\mathcal{F}^{1,2}$ remains undecidable. To have a practical method, we instead employ an inexpensive technique that uses predicate abstraction and model checking to rule out a subset of unreachable states and the transitions of $\mathcal{F}^{1,2}$. Due to space limitations, a detailed description is deferred to the Appendix.

5 Measuring Coverage

While we make conservative simplifying assumptions while reducing the coverage FSM, during actual coverage measurement, no such approximation is needed. Whether a pair of actions executed one after the other are dependent can be easily and exactly determined at run-time by examining the types and parameters of the actions by the coverage tool.

If at any point in the execution of a multi-threaded program, some thread $\mathbf{t}_1$ starts executing μ_1 while another thread $\mathbf{t}_2$ is executing μ_2 (or $\mathbf{t}_2$ starts executing μ_2 while $\mathbf{t}_1$ is executing μ_1), the coverage tool creates a new instance $\mathcal{F}_i^{(1,2)}$ of the class representing the coverage FSM $\mathcal{F}^{1,2}$ and starts it at a state corresponding to the pair of locations that $\mathbf{t}_1$ and $\mathbf{t}_2$ are in. From then on $\mathcal{F}_i^{(1,2)}$ takes transitions triggered by the actions of $\mathbf{t}_1$ and $\mathbf{t}_2$ as described in Section 4 until either $\mathbf{t}_1$ exits that particular execution of μ_1 or $\mathbf{t}_2$ exits μ_2, whichever comes earlier. The coverage tool keeps an instance $\mathcal{F}_{rec}^{(1,2)}$ of the coverage FSM $\mathcal{F}^{1,2}$ for record keeping. All edges of $\mathcal{F}^{1,2}$ visited by any instance of $\mathcal{F}^{1,2}$ created during execution are recorded in $\mathcal{F}_{rec}^{(1,2)}$. Note that different edges of $\mathcal{F}^{1,2}$ may be covered by different pairs of threads.

The feedback provided to the programmer after running a test suite is a list of unexamined pairs of locations that the analysis described above has not been able to rule out as a possibility. At this point, the programmer can either identify this as a true coverage gap and write test programs aimed at exercising those scenarios, or, if he believes this is an unreachable pair of locations, he can provide his reasoning in the form of additional predicates to the coverage feasibility

```
1 public synchronized void addElement(Object obj) {
2    modCount++;
3    ensureCapacityHelper(elementCount + 1);
4    elementData[elementCount] = obj;
5    elementCount++;
6 }
```

```
1 public int lastIndexOf(Object elem) {
2    int count = Read(elementCount) - 1;
3    return lastIndexOf(elem, count);
4 }
```

Fig. 4. Code fragments illustrating the error in `java.util.Vector`

analysis in order to rule out the fictitious coverage gap (see the Appendix). Note that the programmer has to provide his reasoning rather than simply turning off the warning from the coverage tool.

6 Empirical Evidence for the Metric

This section describes how the LP coverage metric captures concurrency errors from the literature and errors in industrial examples we studied. Each error scenario as described is easily expressed as one of the required edges for the coverage FSM for the pair of methods referred to for each example.

`java.util.StringBuffer`: This example, the concurrency error associated with it and how the location pairs metric captures it were discussed in Section 3.

`java.util.Vector`: The code for this example is modified to contain one atomic action per line for illustration purposes (see Fig. 4). If line 3 in `lastIndexOf` is followed by line 5 in line 4 in `addElement`, this leads to an atomicity violation, which was previously discovered by [6]. In particular, if `elem == obj`, this would have led to an incorrect return value for `lastIndexOf`, which is a refinement violation [2].

Cache Module of Boxwood: The error, explained in more detail in [2], involves a cache block in a data structure to be flushed to the next level of the storage hierarchy while it is being overwritten by another thread. This corresponds to line 3 in the `CpToCache` method in Fig. 5 being executed right after line 3 in the `Flush` method, representing flush midway through the copying of buffer `buf` to the cache.

The "Scan" File System: The error in this system, as documented in [10], is very similar in spirit to the scenario in the Boxwood cache. While the file

```
1 private static void CpToCache(byte[] buf,
        CacheEntry te, int lsn, Handle h) {
2    for(int i = 0; i < buf.Length; i++) {
3          te.data[i] = buf[i];
4    }
5      te.lsn = lsn;
6 }
```

```
1 public static void Flush(int lsn) {
...
2 lock(clean) {
3    BoxMain.alloc.Write(h, te.data, te.data.Length,
                         0, 0, WRITE_TYPE.RAW);
4 }
...
```

Fig. 5. Buggy code fragment from an earlier version of the Boxwood `Cache` module

system cache is being written to the disk, after a block gets copied to disk and gets marked “clean”, it gets overwritten by another file system thread.

The Concurrency Error Categories in [3]: The LP metric can express as a coverage goal all error-prone scenarios that are described in this work. The errors in the category “Code Assumed to Be Protected” of [3] are particularly relevant for atomicity and refinement violations.

7 Ongoing Work

We are implementing a software tool written in Java that measures LP metric coverage attained during testing. We instrument the byte-code of the program under test by inserting notification calls from the tested program to the coverage tool after each code block interpreted as an atomic action as described above. To minimize impact of online coverage analysis on the concurrency behavior of the original program, in a later version of the tool, we intend to instrument the program being tested in order to generate per-thread logs of actions relevant to the coverage metric. The coverage tool will then take only the logs as its input and will not affect the execution of the program being tested.

We also intend to study bug databases of other large multi-threaded designs, such as web servers, and determine to what extent the proposed metric captures the bugs documented. Future research includes an implementation of the coverage FSM reduction technique described in the Appendix.

References

1. C. Artho, K. Havelund, and A. Biere. High-Level Data Races. In VVEIS ’03, Int’l Workshop on Verification and Validation of Enterprise Information Systems.
2. T. Elmas, S. Tasiran, and S. Qadeer. Vyrd: Verifying concurrent programs by runtime refinement-violation detection. In *Proc. ACM SIGPLAN 2005 Conf. on Programming Language Design and Implementation, PLDI 2005.* ACM Press, June 2005.
3. Eitan Farchi, Yarden Nir, Shmuel Ur. Concurrent Bug Patterns and How to Test Them. 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, page 286.
4. C. Flanagan, S. Freund, and S. Qadeer. Exploiting purity for atomicity. In *Proc. Intl. Symposium on Software Testing and Analysis (ISSTA 2004).* ACM Press, 2004.
5. C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proc. 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256–267, 2004.
6. C. Flanagan and S. Qadeer. A type and effect system for atomicity. In Proceedings of the ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation, PLDI ’03.
7. B. Long, P. Strooper, and L. Wildman. A method for verifying concurrent java components based on an analysis of concurrency failures. *Concurrency and Computation: Practice and Experience*, 00(1-7):1–13, 2005.

8. B. Long and P. A. Strooper. A classification of concurrency failures in java components. 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France., page 287.
9. P.Godefroid. Partial order methods for the verification of concurrent systems-an approach to the state-explosion problem. In *Lecture Notes in Computer Science*, volume 1032. Springer-Verlag, 1996.
10. S. Tasiran, A. Bogdanov, and M. Ji. Detecting concurrency errors in file systems by runtime refinement checking. Tech. Report HPL-2004-177, HP Labs, 2004.
11. R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. *IEEE Trans. Softw. Eng.*, 18(3):206–215, 1992.

Adaptive Random Testing by Bisection and Localization

Johannes Mayer

University of Ulm,
Dept. of Applied Information Processing,
89069 Ulm, Germany
johannes.mayer@uni-ulm.de

Abstract. Adaptive Random Testing (ART) denotes a family of test case generation algorithms that are designed to detect common failure patterns better than pure Random Testing. The best known ART algorithms, however, use many distance computations. Therefore, these algorithms are quite inefficient regarding runtime. New algorithms combining Adaptive Random Testing by Bisection and the principle of localization are presented. These algorithms heavily reduce the amount of distance computation while exhibiting very good performance measured in terms of the number of test cases necessary to detect the first failure.

1 Introduction

Software testing is the process of executing a program with the intention to uncover bugs [1]—an old and still important definition. It is usually quite time consuming to produce a significant number of test cases. Therefore, Random Testing [2, 3, 4, 5, 6], i. e. the random generation of test cases, has gained much importance. Another black box strategy, namely partition testing [7], i. e. the division of the input domain into partitions and the generation of test cases from each partition, has also attracted much attention. There have been several investigations comparing Random Testing and partition testing [7, 8, 9, 10, 11]. An advantage of Random Testing over partition testing is that it is able to deliver reliability predictions [4, 12, 13]. However, the fact that Random Testing does not use information about the program under test has been criticized [1] and led to criteria on when partition testing performs better than Random Testing [7, 9]. Adaptive Random Testing (ART) [10] has been designed to detect common failure patterns better than pure Random Testing. This aim is achieved through wide spread test cases. The best ART algorithms (D-ART [10] and RRT [14, 15, 16]), however, require a huge amount of distance computations. Recently, ART algorithms inspired by partition testing have be published [17]. These algorithms do not require distance computations. Their performance measures in terms of the number of test cases necessary to detect the first failure are better than that of pure Random Testing, but not nearly as good as the best ART algorithms. The present contribution presents two algorithms which combine D-ART resp. RRT with dynamic partitioning using the principle of localization.

W. Grieskamp and C. Weise (Eds.): FATES 2005, LNCS 3997, pp. 72–86, 2006.

Whereas in [18] this was done for ART by random partitioning, the present contribution combines D-ART resp. RRT with ART by bisection.

The following section presents preliminaries regarding failure patterns, notation, and common ART methods. The proposed algorithms are detailed in Section 3. An empirical evaluation of the proposed algorithms is described and discussed in Section 4, followed by a conclusion.

2 Preliminaries

2.1 Notation

The input domain is assumed to be bounded. The failure rate, i. e. the percentage of failure-causing inputs, is denoted θ. For a finite input domain of size d with m failure-causing inputs, $\theta = m/d$.

The *F-measure* is the number of test cases necessary to detect the first failure. This is a very natural measure for the performance of a testing strategy, since often testing is stopped when the first failure is detected. The F-measure has been used in all publications on ART. It is therefore ideal for comparison purposes.

For Random Testing with uniform input profile and replacement, the theoretical mean F-measure is equal to $1/\theta$.[1] For example, for a failure rate of $\theta = 0.01$, the theoretical mean F-measure of random testing with replacement is 100.

2.2 Failure Patterns

Chan et al. [8] observed and described three typical patterns of failure-causing inputs within the input domain of real programs they examined (cf. Figure 1). The block pattern (cf. Figure 1a) describes the situation where the failure-causing inputs are located next to each other within a small region of the input domain. The strip pattern (cf. Figure 1b) is achieved if the failure-causing inputs form a narrow strip within the input domain. Finally, the situation when there are

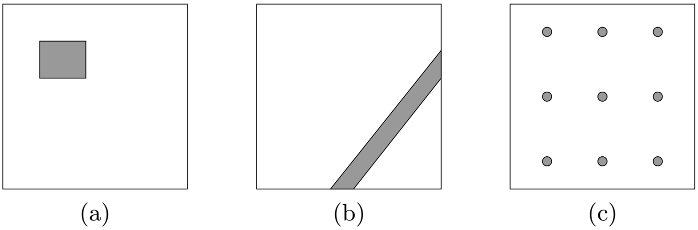


Fig. 1. Block, strip, and point patterns within a two-dimensional input domain

[1] Let F_{RT} denote the theoretical mean F-measure of Random Testing with replacement for a given failure rate θ. This notation is used in the tables with the simulation results.

many wide spread failure-causing inputs or small clusters of such inputs is described by the point pattern (cf. Figure 1c). According to Chan et al. [8], the block and the strip failure pattern are the most common. These patterns support the intuition of ART that wide spread test cases have a higher probability of earlier detecting failures.

2.3 Some ART Algorithms

The first ART algorithm was distance-based ART (D-ART) [10, 19] with fixed sized candidate set. This method chooses the first test case purely randomly. Thereafter, a set of test case candidates—each chosen purely randomly—of size k is computed in each iteration. The test case from the candidate set with the greatest minimal distance to the already executed test cases is chosen as the next test case. In the following iteration, a new candidate set is chosen and the procedure is repeated until a failure is detected (or the resources for testing are exhausted). The size $k = 10$ of the candidate set has been recommended.

Another ART algorithm is based on restriction, namely Restricted Random Testing (RRT) [14, 15, 16]. The first test case is again chosen purely random. In each iteration a disc with exclusion radius $r := \sqrt{A \cdot R/(\pi \cdot n)}$ is located around each previously executed test case that did not exhibit a failure. A denotes the area of the input domain, R denotes the coverage ratio (of the exclusion zone relative to the area of the input domain), and n is the number of previously executed test cases not causing a failure. Each randomly generated test case that falls within one of these discs is rejected. The first test candidate that does not fall within the exclusion zone is chosen as the next test case and the procedure is repeated. A coverage ratio $R = 1.5$ is given as a recommendation.

D-ART and RRT require many distance computations. Therefore, ART by bisection has been presented in [17] along with ART by random partitioning. These methods have the advantage that they do not need distance computations. ART by bisection chooses the first test case purely randomly. Thereafter, the input domain is bisected. One of the two resulting partitions contains the already executed test case and the other does not. The "empty" partitions that do not contain a previously executed test case are selected in a random order and a test case is generated purely randomly within each such region. As soon as all partitions contain previously executed test cases, all partitions are bisected along the larger side. The generation of test cases within the "empty" partitions and the bisection of all partitions is repeated in each iteration. Some steps of the execution of this algorithm are illustrated in Figure 2. The "non-empty" partitions are shown hatched. Figure 2a shows the first iteration where one test case has been selected randomly. After the bisection at the end of the first iteration there remains only one "empty" partition for the second iteration. Figure 2b shows the generation of a test case within this region. After further bisection at the end of the second iteration, Test 3 is chosen within an "empty" region in iteration three (cf. Figure 2c). Finally, Figure 2d shows one step of the fourth iteration. The partitions have again been bisected after the third iteration.

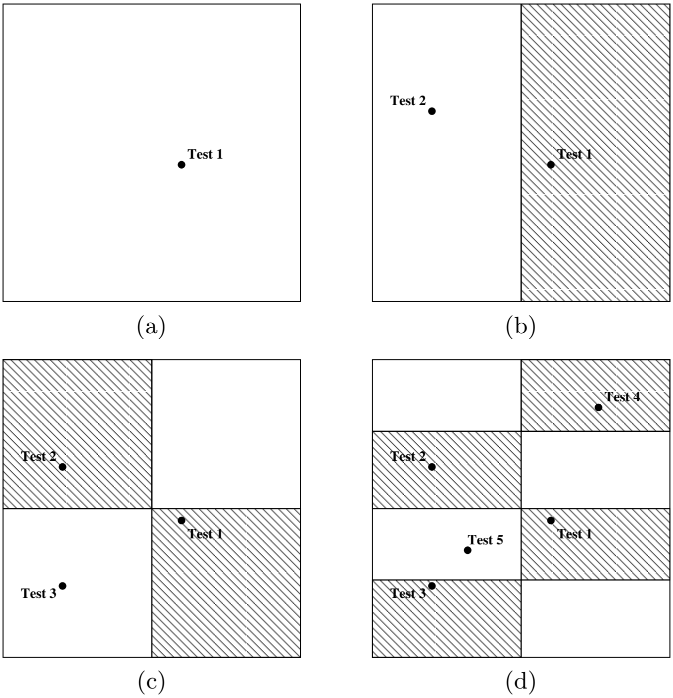


Fig. 2. Adaptive Random Testing by Bisection: Some steps of the algorithm

3 The ART by Bisection and Localization Algorithms

Whereas D-ART and RRT extensively use distance computations, ART by bisection does not need any distance computation. However, ART by bisection performs significantly worse than D-ART and RRT in terms of the F-measure. The combination of ART by random partitioning and D-ART resp. RRT in [18] is based on the principle of localization. This means that only "nearby" previously executed test cases are used for the distance computations to reduce the effort of D-ART and RRT. The novel ART algorithms are also based on the principle of localization and combine ART by bisection with D-ART resp. RRT. The selection of the next test case is done using D-ART resp. RRT, where the distance computation is only performed with "neighboring" previously executed test cases.

It is assumed that the two-dimensional input domain is rectangular with lower left corner $(x_{\min}, y_{\min})$ and upper right corner $(x_{\max}, y_{\max})$. Therefore, the inputs are two-dimensional vectors (x, y) of real values with $x_{\min} \leq x \leq x_{\max}$ and $y_{\min} \leq y \leq y_{\max}$. It can trivially be adapted to a bounded region of integers or higher dimensional input domains.

The coverage ratio R for the first algorithm must be within $[0, 0.7]$.

Algorithm 1: Adaptive Random Testing by Bisection and Localization with RRT

```
1. Initialize the list of untested regions $L_{\text{untested}} := \{\{(x_{\min}, y_{\min})(x_{\max}, y_{\max})\}\}$,
   the list of tested regions $L_{\text{tested}}$ with the empty list, and the target exclusion
   area $A_{excl} := \mathrm{R} \cdot (x_{\max} - x_{\min}) \cdot (y_{\max} - y_{\min})$.
2. While $L_{\text{untested}}$ is not empty:
   (a) Set the exclusion radius $r := \sqrt{A_{excl}/(\pi \cdot \#(L_{\text{tested}}))}$.[2]
   (b) Randomly select a test region $T = \{(x_0, y_0)(x_1, y_1)\}$ from $L_{\text{untested}}$ and
       do *not* remove it.
   (c) Randomly select a point $(x, y)$ from within the test region $T$.
   (d) For each neighbor $T'$ of $T$ with $(T', (x', y'))$ in $L_{\text{tested}}$:
       If $\text{dist}((x, y), (x', y')) \leq r$ then reject $(x, y)$ and go back to step 2.
   (e) If the point $(x, y)$ is a failure-causing input, report failure and terminate.
   (f) Otherwise append $(T, (x, y))$ to $L_{\text{tested}}$ and remove $T$ from $L_{\text{untested}}$.
3. Initialize $L_{\text{temp}}$ with the empty list.
4. For each element $(\{(x_0, y_0)(x_1, y_1)\}, (x, y))$ from $L_{\text{tested}}$:
   (a) Let $T := \{(x_0, y_0)(x_1, y_1)\}$. Furthermore, let $w := x_1 - x_0$ be the width
       of $T$ and $h := y_1 - y_0$ be the height of $T$.
   (b) If $w \geq h$, divide $T$ into the two regions $T_1 := \{(x_0, y_0)(x_0 + w/2, y_1)\}$ and
       $T_2 := \{(x_0 + w/2, y_0)(x_1, y_1)\}$. Otherwise divide $T$ into the two regions
       $T_1 := \{(x_0, y_0)(x_1, y_0 + h/2)\}$ and $T_2 := \{(x_0, y_0 + h/2)(x_1, y_1)\}$.
   (c) If $(x, y) \in T_2$, exchange $T_1$ and $T_2$.
   (d) Add $(T_1, (x, y))$ to $L_{\text{temp}}$ and $T_2$ to $L_{\text{untested}}$.
5. Copy $L_{\text{temp}}$ into $L_{\text{tested}}$ and proceed with step 2.
```

Within each step, an "empty" region is chosen and a test case candidate is randomly selected within this region. If the candidate is rejected (i. e. the distance to one of its neighboring[3] regions is not greater than the exclusion radius r), the process of choosing an "empty" region and selecting a test case candidate is repeated. Otherwise, the candidate is executed as the next test case and the region is marked "non-empty". Figure 3 illustrates a possible situation within the execution of the algorithm. Thereafter the algorithm proceeds as illustrated in Figure 4. It first selects the lower left "empty" region and a test case candidate within it (cf. Figure 4a). Since the candidate is within the exclusion area determined by the discs around the neighboring previously executed test case, it is rejected. Thereafter, the upper left "empty" region is chosen and a test case candidate is selected which is not rejected (cf. Figure 4b), since it is not within the discs.

The following algorithm has the size k of the candidate set as its parameter.

[2] $\#(L_{\text{tested}})$ denotes the size of the list L_{tested}.

[3] A region has at most four neighboring regions: The left, right, upper, and lower neighbor.

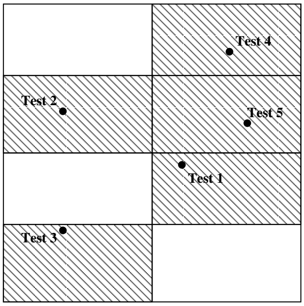
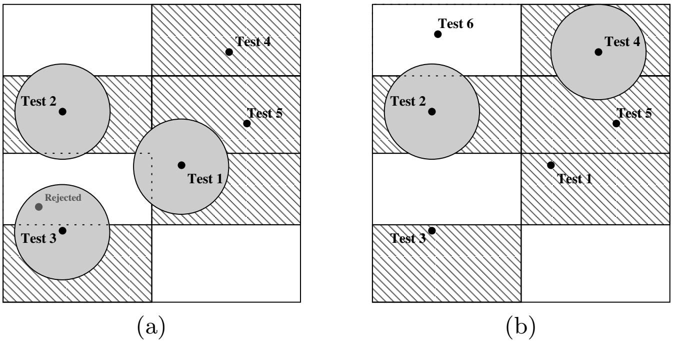


Fig. 3. Adaptive Random Testing by Bisection and Localization: The previously executed test cases, "empty" and "non-empty" regions



(a) (b)

Fig. 4. Adaptive Random Testing by Bisection and Localization with RRT

Algorithm 2: Adaptive Random Testing by Bisection and Localization with D-ART

```
1. Initialize the list of untested regions $L_{\text{untested}} := \{\{(x_{\min}, y_{\min})(x_{\max}, y_{\max})\}\}$
   and the list of tested regions $L_{\text{tested}}$ with the empty list.
2. While $L_{\text{untested}}$ is not empty:
   (a) Initialize the candidate set $C$ with the empty set.
   (b) Repeat $k$ times:
       i. Randomly select a test region $T = \{(x_0, y_0)(x_1, y_1)\}$ from $L_{\text{untested}}$
          and do *not* remove it.
       ii. Randomly select a point $(x, y)$ from within the test region $T$.
       iii. Initialize the neigbor set $N$ with the empty set.
       iv. For each neighbor $T'$ of $T$ with $(T', (x', y'))$ in $L_{\text{tested}}$:
           Add $(x', y')$ to $N$.
       v. Let $d$ be the minimal distance between $(x, y)$ and the set of neighbors
          $N$.
       vi. Add $((x, y), T, d)$ to $C$.
   (c) Select the element $((x, y), T, d)$ from $C$ with maximal distance $d$.
```

   (d) If the point (x, y) is a failure-causing input, report failure and terminate.
   (e) Otherwise append $(T, (x, y))$ to L_{tested} and remove T from L_{untested}.
3. Initialize L_{temp} with the empty list.
4. For each element $(\{(x_0, y_0)(x_1, y_1)\}, (x, y))$ from L_{tested}:
   (a) Let $T := \{(x_0, y_0)(x_1, y_1)\}$. Furthermore, let $w := x_1 - x_0$ be the width of T and $h := y_1 - y_0$ be the height of T.
   (b) If $w \geq h$, divide T into the two regions $T_1 := \{(x_0, y_0)(x_0 + w/2, y_1)\}$ and $T_2 := \{(x_0 + w/2, y_0)(x_1, y_1)\}$. Otherwise divide T into the two regions $T_1 := \{(x_0, y_0)(x_1, y_0 + h/2)\}$ and $T_2 := \{(x_0, y_0 + h/2)(x_1, y_1)\}$.
   (c) If $(x, y) \in T_2$, exchange T_1 and T_2.
   (d) Add $(T_1, (x, y))$ to L_{temp} and T_2 to L_{untested}.
5. Copy L_{temp} into L_{tested} and proceed with step 2.

In each step, k times an "empty" region is chosen and a test case candidate is selected. For each test case candidate the minimal distance to its neighboring previously executed test cases is computed. The candidate which maximizes this distance is then chosen as the next test case and the process is repeated. For the previous situation as in Figure 3, the situation after the selection of all test case candidates is illustrated in Figure 5. The distance computations are illustrated through dashed lines. Since the candidate in the upper left region maximizes the minimal distance to its neighboring previously executed test cases, it is chosen as the next test case.

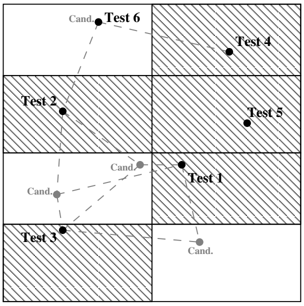


Fig. 5. Adaptive Random Testing by Bisection and Localization with D-ART

Since there are at most four neighbors, the total number of distance computations is

$$\sum_{i=1}^{F} 4k = 4kF$$

for a run of the D-ART variant with F-measure F. This can also be assumed the total number of distance computations for the RRT variant if k is seen as the mean number of necessary test case candidates in each iteration. Thus, for

ART by bisection and localization, the total number of distance computation is *linear* in the F-measure, whereas it is quadratic for D-ART and RRT.[4]

4 Simulation Study

To compare the novel algorithms with other ART methods, a simulation study has been performed and the F-measure has been determined.

4.1 The Simulation Design

For the first part of the simulations, the sample size n was chosen 50000, i. e. the algorithm was run with 50000 randomly chosen failure patterns. For the second part of the simulation the sample size was $n = 5000$. The confidence level $1 - \alpha$ was chosen 0.99. In a table one can look up $\Phi^{-1}(0.995) \approx 2.58$—a quantile of the normal distribution. Therefore, the accuracy is

$$|\overline{X_n} - \mu| \leq \frac{S_n}{\sqrt{50000}} \cdot 2.58 \approx 0.01154 \cdot S_n$$

and

$$|\overline{X_n} - \mu| \leq \frac{S_n}{\sqrt{5000}} \cdot 2.58 \approx 0.03649 \cdot S_n$$

on confidence level 99% using the central limit theorem, where X_n are the F-measures of the sample, $\overline{X_n}$ is the sample mean of the F-measures, μ is the true mean of the F-measure, and S_n is the sample standard deviation of the F-measures. μ is the F-measure to be determined. $\overline{X_n}$ are the simulation results. The above formulae, thus, yield the accuracy of the simulation results.

The failure pattern was randomly generated. The area θA of the failure pattern was determined by the failure rate θ and the area A of the input domain. For the block pattern, a square was chosen randomly totally within the input domain. For the strip pattern, two adjacent sides and two points on these sides were chosen randomly. The strip was then constructed centered on the line connecting these points and its width was computed so that the strip had the desired area θA. Points near the corners were rejected to avoid overly wide strips. For the point pattern, 50 and alternatively 10 non-overlapping discs with equal radius being totally within the input domain were randomly generated to achieve the total area θA.

The first part of the simulations were to investigate the performance of the proposed ART algorithms and to find suitable values for the parameters R and k. These simulation were done with $n = 50000$ samples for the following

- failure rates: $0.01, 0.005, 0.002, 0.001, 0.0005$
- failure patterns: block, strip, and point (with 50 discs)

and various coverage ratios R and candidate set sizes k.

[4] One has to replace k by $(i - 1)$ within the sum in the above formula for D-ART and RRT, which results in $4F(F + 1)/2 = 2F^2 + 2F$.

The second part of the simulations was performed in order to compare the novel ART algorithms with related ART algorithms. In this case, only $n = 5000$ samples were used (due to the runtime of D-ART and RRT). The parameters of the various ART methods were as follows: RRT ($R = 1.5$), D-ART ($k = 10$), ART by random partitioning with localization and RRT ($R = 0.4$) resp. D-ART ($k = 3$), and ART by bisection with localization and RRT ($R = 0.7$) resp. D-ART ($k = 13$). (The last two parameters have been determined by the first part of the simulations.) In this case, the above failure rates and patterns were also used complemented by the point pattern with 10 discs.

For the simulations with equal failure rate and failure pattern, the same generated failure pattern has been used for each ART method.

4.2 Results and Discussion

The results of the first part of the simulations are shown in Tables 1–5 for the block pattern. Each table contains the relative[5] mean F-measure (in the third and fourth column) along with its accuracy (with 99% confidence) in parentheses below and the relative standard deviation of the F-measure (in the fifth and sixth column). The first two columns contain the parameters of the algorithms. The minimum of the third and fourth column is in bold face.

Table 1. Improvement of ART by bisection with localization with block pattern and failure rate 0.01

		F/F_{RT}		Std.Dev.$/F_{RT}$	
R	k	RRT	D-ART	RRT	D-ART
0.0	1	0.741 (±0.007)	0.734 (±0.007)	0.610	0.609
0.1	3	0.732 (±0.007)	0.678 (±0.006)	0.603	0.525
0.2	5	0.721 (±0.007)	0.669 (±0.006)	0.594	0.505
0.3	7	0.713 (±0.007)	0.668 (±0.006)	0.588	0.492
0.4	9	0.708 (±0.007)	0.663 (±0.006)	0.580	0.485
0.5	11	0.693 (±0.006)	0.668 (±0.006)	0.560	0.485
0.6	13	0.691 (±0.006)	**0.662** (±0.006)	0.556	0.479
0.7	15	**0.681** (±0.006)	0.667 (±0.006)	0.547	0.478

Table 2. Improvement of ART by bisection with localization with block pattern and failure rate 0.005

		F/F_{RT}		Std.Dev.$/F_{RT}$	
R	k	RRT	D-ART	RRT	D-ART
0.0	1	0.742 (±0.007)	0.738 (±0.007)	0.629	0.631
0.1	3	0.733 (±0.007)	0.664 (±0.006)	0.626	0.526
0.2	5	0.723 (±0.007)	0.657 (±0.006)	0.612	0.504
0.3	7	0.711 (±0.007)	0.650 (±0.006)	0.598	0.493
0.4	9	0.701 (±0.007)	0.645 (±0.006)	0.589	0.482
0.5	11	0.689 (±0.007)	0.649 (±0.006)	0.571	0.486
0.6	13	0.681 (±0.006)	**0.644** (±0.006)	0.559	0.478
0.7	15	**0.675** (±0.006)	0.650 (±0.006)	0.550	0.482

[5] Relative to the theoretical mean F-measure of Random Testing.

Table 3. Improvement of ART by bisection with localization with block pattern and failure rate 0.002

		F/F_{RT}		Std.Dev./F_{RT}	
R	k	RRT	D-ART	RRT	D-ART
0.0	1	0.731 (±0.007)	0.732 (±0.007)	0.608	0.609
0.1	3	0.730 (±0.007)	0.660 (±0.006)	0.607	0.517
0.2	5	0.713 (±0.007)	0.649 (±0.006)	0.594	0.496
0.3	7	0.704 (±0.007)	0.637 (±0.006)	0.584	0.484
0.4	9	0.700 (±0.007)	0.639 (±0.006)	0.574	0.482
0.5	11	0.686 (±0.006)	**0.634** (±0.005)	0.560	0.475
0.6	13	0.675 (±0.006)	0.635 (±0.005)	0.547	0.475
0.7	15	**0.663** (±0.006)	0.635 (±0.005)	0.537	0.473

Table 4. Improvement of ART by bisection with localization with block pattern and failure rate 0.001

		F/F_{RT}		Std.Dev./F_{RT}	
R	k	RRT	D-ART	RRT	D-ART
0.0	1	0.739 (±0.007)	0.737 (±0.007)	0.633	0.636
0.1	3	0.732 (±0.007)	0.661 (±0.006)	0.627	0.527
0.2	5	0.722 (±0.007)	0.644 (±0.006)	0.612	0.501
0.3	7	0.708 (±0.007)	0.639 (±0.006)	0.601	0.489
0.4	9	0.699 (±0.007)	0.632 (±0.006)	0.588	0.482
0.5	11	0.687 (±0.007)	0.629 (±0.005)	0.574	0.477
0.6	13	0.679 (±0.006)	0.631 (±0.005)	0.561	0.476
0.7	15	**0.671** (±0.006)	**0.629** (±0.005)	0.549	0.473

The simulation results for ART by bisection and localization with RRT and the block pattern are best for $R = 0.7$. For each failure rate, the relative mean F-measure decreases with increasing R. The case $R = 0.0$ stands for the original ART by bisection (without localization). The improvement of ART by bisection with localization and RRT with $R = 0.7$ over the original ART by bisection is between 0.06 and 0.07. The relative mean F-measure for ART by bisection with localization and RRT with $R = 0.7$ is between 0.663 and 0.681 for the block pattern. There is a tendency that the F-measure decreases for decreasing failure rate—as with the original ART by bisection.

For ART by bisection and localization with D-ART, the simulation results indicate that a choice of $k = 13$ seems to be optimal. As above, the choice $k = 1$ stands for the original ART by bisection. One can observe that the relative mean F-measure decreases from $k = 1$ to $k = 13$ by between 0.07 and 0.10. The relative mean F-measure for $k = 13$ is between 0.631 and 0.662.

Therefore, D-ART seems to be the better choice than RRT for ART by bisection and localization in case of the block pattern.

Similar simulations have also been made for the strip pattern and the point pattern (with 50 discs). The results, however, could not be included due to space limitations.

For the strip pattern and ART by bisection and localization with RRT, a choice of $R = 0.5$ seems to be optimal. However there is at most a difference of 0.01 between the relative mean F-measure for $R = 0.5$ and $R = 0.7$. For D-ART, $k = 7$ seems to be optimal. But the difference between the relative mean F-measures for $k = 7$ and $k = 13$ is always below 0.007.

Table 5. Improvement of ART by bisection with localization with block pattern and failure rate 0.0005

		F/F_{RT}		Std.Dev./F_{RT}	
R	k	RRT	D-ART	RRT	D-ART
0.0	1	0.733 (±0.007)	0.730 (±0.007)	0.608	0.608
0.1	3	0.727 (±0.007)	0.653 (±0.006)	0.608	0.518
0.2	5	0.718 (±0.007)	0.641 (±0.006)	0.598	0.494
0.3	7	0.709 (±0.007)	0.633 (±0.006)	0.588	0.483
0.4	9	0.687 (±0.007)	0.634 (±0.006)	0.571	0.481
0.5	11	0.684 (±0.006)	0.630 (±0.005)	0.557	0.475
0.6	13	0.675 (±0.006)	0.631 (±0.005)	0.552	0.474
0.7	15	**0.667** (±0.006)	**0.628** (±0.005)	0.537	0.467

For the point pattern with 50 discs, the optimal mean values are $R = 0.3$ and $k = 3$. However, the relative mean F-measure differs only up to 0.01 between the optimal parameter and $R = 0.7$ resp. $k = 13$.

Altogether, it seems to be justified to choose $R = 0.7$ and $k = 13$, since these parameters have the greatest influence for the block failure pattern.

The results of the second part of the simulation study is shown in Tables 6–9. In the second study, the novel ART methods (with parameters $R = 0.7$ and

Table 6. The mean F-measure of the respective ART method related to the mean F-measure of Random Testing for the block failure pattern

	$F -$ measure$/F_{RT}$							
	RRT	D-ART	ART-RP	ART-Bi.	ART-RP Loc.		ART-Bi. Loc.	
Failure Rate					RRT	D-ART	RRT	D-ART
$\theta = 0.0100$	0.652 (±0.017)	0.672 (±0.018)	0.758 (±0.024)	0.748 (±0.023)	0.671 (±0.020)	0.728 (±0.022)	0.681 (±0.020)	0.665 (±0.017)
$\theta = 0.0050$	0.632 (±0.016)	0.654 (±0.018)	0.780 (±0.025)	0.744 (±0.023)	0.679 (±0.021)	0.712 (±0.022)	0.686 (±0.020)	0.649 (±0.017)
$\theta = 0.0020$	0.609 (±0.016)	0.650 (±0.018)	0.786 (±0.024)	0.741 (±0.023)	0.675 (±0.020)	0.720 (±0.022)	0.667 (±0.020)	0.638 (±0.017)
$\theta = 0.0010$	0.593 (±0.016)	0.638 (±0.018)	0.788 (±0.025)	0.743 (±0.023)	0.692 (±0.021)	0.721 (±0.022)	0.663 (±0.020)	0.636 (±0.017)
$\theta = 0.0005$	0.594 (±0.015)	0.642 (±0.018)	0.813 (±0.026)	0.724 (±0.022)	0.688 (±0.020)	0.730 (±0.023)	0.674 (±0.020)	0.617 (±0.017)

Table 7. The mean F-measure of the respective ART method related to the mean F-measure of Random Testing for the strip failure pattern

	$F-\text{measure}/F_{RT}$							
	RRT	D-ART	ART-RP	ART-Bi.	ART-RP Loc.		ART-Bi. Loc.	
Failure Rate					RRT	D-ART	RRT	D-ART
$\theta = 0.0100$	0.854 (±0.030)	0.854 (±0.030)	0.954 (±0.034)	0.933 (±0.032)	0.902 (±0.031)	0.934 (±0.033)	0.889 (±0.031)	0.878 (±0.031)
$\theta = 0.0050$	0.876 (±0.031)	0.910 (±0.032)	0.954 (±0.034)	0.961 (±0.034)	0.937 (±0.033)	0.952 (±0.033)	0.918 (±0.032)	0.928 (±0.032)
$\theta = 0.0020$	0.908 (±0.032)	0.933 (±0.034)	1.011 (±0.036)	0.967 (±0.035)	0.938 (±0.033)	0.973 (±0.034)	0.959 (±0.034)	0.949 (±0.034)
$\theta = 0.0010$	0.947 (±0.035)	0.958 (±0.035)	0.982 (±0.035)	0.994 (±0.036)	0.966 (±0.034)	0.957 (±0.034)	0.976 (±0.035)	0.985 (±0.036)
$\theta = 0.0005$	0.955 (±0.034)	0.952 (±0.035)	0.994 (±0.035)	0.962 (±0.035)	0.982 (±0.035)	0.980 (±0.035)	0.960 (±0.035)	0.982 (±0.035)

Table 8. The mean F-measure of the respective ART method related to the mean F-measure of Random Testing for the point failure pattern with 10 discs

	$F-\text{measure}/F_{RT}$							
	RRT	D-ART	ART-RP	ART-Bi.	ART-RP Loc.		ART-Bi. Loc.	
Failure Rate					RRT	D-ART	RRT	D-ART
$\theta = 0.0100$	0.995 (±0.033)	0.958 (±0.032)	0.948 (±0.033)	0.939 (±0.032)	0.925 (±0.031)	0.938 (±0.032)	0.932 (±0.031)	0.951 (±0.031)
$\theta = 0.0050$	0.970 (±0.032)	0.946 (±0.031)	0.956 (±0.034)	0.956 (±0.033)	0.934 (±0.032)	0.935 (±0.032)	0.936 (±0.032)	0.935 (±0.031)
$\theta = 0.0020$	0.937 (±0.030)	0.904 (±0.030)	0.943 (±0.033)	0.928 (±0.031)	0.934 (±0.033)	0.918 (±0.031)	0.920 (±0.031)	0.953 (±0.032)
$\theta = 0.0010$	0.927 (±0.030)	0.924 (±0.031)	0.963 (±0.034)	0.943 (±0.033)	0.928 (±0.032)	0.915 (±0.031)	0.897 (±0.030)	0.946 (±0.032)
$\theta = 0.0005$	0.921 (±0.030)	0.927 (±0.031)	0.942 (±0.033)	0.950 (±0.033)	0.959 (±0.032)	0.918 (±0.032)	0.901 (±0.030)	0.927 (±0.031)

$k = 13$) are compared to several other ART methods (also with recommended parameters as listed in the previous section) using some common failure patterns and various failure rates. For this study, the sample size was $n = 5000$ due to the huge runtime of RRT and D-ART for small failure rates. The following abbreviation where chosen in the tables: ART by random partitioning (ART-RP), ART by bisection (ART-Bi.), ART by random partitioning and localization (ART-RP Loc.) combined with either RRT or D-ART, and finally ART by bisection and localization (ART-Bi. Loc.) also combine with either RRT or D-ART.

For the block failure pattern, the following observations can be made: The best known ART methods are D-ART and RRT. ART by bisection and localization with D-ART ($k = 13$) is better than D-ART and has only a slightly higher relative mean F-measure (by about between 0.01 and 0.02) than RRT (with

Table 9. The mean F-measure of the respective ART method related to the mean F-measure of Random Testing for the point failure pattern with 50 discs

	$F -$ measure$/F_{RT}$							
	RRT	D-ART	ART-RP	ART-Bi.	ART-RP Loc.		ART-Bi. Loc.	
Failure Rate					RRT	D-ART	RRT	D-ART
$\theta = 0.0100$	1.032 (±0.036)	0.998 (±0.036)	0.972 (±0.035)	0.982 (±0.035)	0.999 (±0.037)	0.979 (±0.035)	0.970 (±0.035)	1.026 (±0.036)
$\theta = 0.0050$	1.030 (±0.036)	1.006 (±0.037)	0.989 (±0.035)	0.964 (±0.034)	1.005 (±0.036)	0.978 (±0.035)	1.002 (±0.036)	0.990 (±0.036)
$\theta = 0.0020$	0.999 (±0.036)	1.016 (±0.036)	0.972 (±0.034)	0.989 (±0.036)	0.991 (±0.036)	0.970 (±0.035)	0.977 (±0.035)	0.985 (±0.035)
$\theta = 0.0010$	0.984 (±0.035)	0.994 (±0.035)	0.995 (±0.036)	0.967 (±0.035)	0.980 (±0.035)	0.974 (±0.035)	1.003 (±0.036)	0.997 (±0.036)
$\theta = 0.0005$	0.973 (±0.035)	0.981 (±0.035)	0.967 (±0.035)	0.983 (±0.036)	0.968 (±0.034)	0.970 (±0.036)	1.000 (±0.036)	1.003 (±0.036)

$k = 10$). It is also better than ART by random partitioning and localization [18] and ART through dynamic partitioning [17]. RT by bisection and localization with RRT ($R = 0.7$) is a litte bit worse than the D-ART variant, but always better than ART through dynamic partitioning and mostly than ART by random partitioning and localization.

For the strip failure pattern the following can be said: As for the block pattern, RRT and D-ART are the best known ART methods. ART by bisection and localization with D-ART is again (mostly) better than ART through dynamic partitioning and ART by random partitioning. Compared to RRT, it is by at most 0.04 to 0.05 worse. Both variants of ART by bisection and localization cannot really be compared due to the accuracy.

For point patterns with 10 discs, ART by bisection and localization with RRT seems to be the best ART method. However for point patterns with 50 discs, ART by random partitioning and localization with D-ART performs best. The novel ART algorithms are at least not worse than Random Testing.

Summarizing, the ART by bisection and localization algorithm combined with D-ART performs very good for the block pattern—better than D-ART and nearly as good as RRT. This proposed algorithm is for the strip pattern also the ART method with computational efficiency and best performance among all efficient ART algorithms. For point pattern with 10 discs, the RRT variant of the proposed algorithm is (in most cases) the best ART algorithm. And for point patterns with 50 discs the presented algorithms are at least noworse than Random Testing.

5 Conclusion

Two innovative ART algorithms combining ART by bisection and the concept of localization have been presented. These algorithms need only a linear number of distance computations in terms of the F-measure—the number of test cases necessary to detect the first failure-causing input—, whereas the D-ART and RRT algorithms need a quadratic number. Furthermore, the new ART algorithms perform better in terms of the F-measure than ART by dynamic partitioning and ART by random partitioning and localization for block, strip, and point patterns (with 10 discs). And the proposed ART algorithms are not worse than Random Testing for the point pattern with 50 discs. For the block pattern, the D-ART variant of the new method even outperforms D-ART. This variant also performs quite good for the strip pattern. For the point pattern with 10 discs, the RRT variant is even the best ART method.

It has to be pointed out that in the literature various methods for the simulation have been used. Also the point pattern is sometimes simulated by 10 discs and sometimes by 50 discs. Through a huge comparative simulation the present paper provides a uniform comparison of the discussed ART methods. This simulation study is based on artificial failure patterns. A similar study with mutated programs should be performed to evaluate the algorithms with real examples. Furthermore, it would also be a good idea to compare the ART methods not only by their F-measure, but also by their runtime. This will be done in an upcoming study.

The algorithms have been described for two-dimensional bounded input domains of numbers. It can be generalized to similar domains in higher dimensions. However, it seems not to be trivial to generalize to other types of inputs, since suitable metric spaces have to be identified.

As shared by all Random Testing and ART methods, there is the need for a test oracle. If there is no oracle, Random Testing can not be applied. However if such an oracle is accessible and Random Testing is considered, it is worth to try the presented algorithms, since their achieved F-measure is comparable to that of D-ART and RRT, and they are much more efficient regarding runtime than D-ART and RRT. In most cases, the presented algorithms seem to be the ART algorithms with the lowest F-measure among all efficient ART algorithms.

Acknowledgement

The author is grateful to T.Y. Chen for discussions and help regarding the simulation of failure patterns.

References

1. Myers, G.J.: The Art of Software Testing. Wiley, New York (1979)
2. Agrawal, V.D.: When to use random testing. IEEE Transactions on Computers **27** (1978) 1054–1055
3. Duran, J.W., Ntafos, S.C.: An evaluation of random testing. IEEE Transactions on Software Engineering **10** (1984) 438–444
4. Hamlet, R.: Random testing. In: Encylopedia of Software Engineering. Wiley (1994) 970–978
5. Loo, P.S., Tsai, W.K.: Random testing revisited. Information and Software Technology **30** (1988) 402–417

6. Schneck, P.B.: Comment on "when to use random testing". IEEE Transactions on Computers **28** (1979) 580–581
7. Weyuker, E.J., Jeng, B.: Analysing partition testing strategies. IEEE Transactions on Software Engineering **17** (1991) 703–711
8. Chan, F.T., Chen, T.Y., Mak, I.K., Yu, Y.T.: Proportional sampling strategy: Guidelines for software testing practitioners. Information and Software Technology **38** (1996) 775–782
9. Chen, T.Y., Yu, Y.T.: On the relationship between partition and random testing. IEEE Transactions on Software Engineering **20** (1994) 977–980
10. Chen, T.Y., Tse, T.H., Yu, Y.T.: Proportional sampling strategy: A compendium and some insights. The Journal of Systems and Software **58** (2001) 65–81
11. Hamlet, R.G., Taylor, R.: Partition testing does not inspire confidence. IEEE Transactions on Software Engineering **16** (1990) 1402–1411
12. Frankl, P.G., Hamlet, R.G., Littlewood, B., Strigini, L.: Evaluating testing methods by delivered reliability. IEEE Transactions on Software Engineering **24** (1998) 586–601
13. Frankl, P.G., Hamlet, R.G., Littlewood, B., Strigini, L.: Correction to: Evaluating testing methods by delivered reliability. IEEE Transactions on Software Engineering **25** (1999) 286
14. Chan, K.P., Chen, T.Y., Towey, D.: Restricted random testing. In Kontio, J., Conradi, R., eds.: Proceedings of the 7th European Conference on Software Quality (ECSQ 2002). Volume 2349 of Lecture Notes in Computer Science., Springer (2002) 321–330
15. Chan, K.P., Chen, T.Y., Towey, D.: Normalized restricted random testing. In: Proceedings of the 18th Ada-Europe International Conference on Reliable Software Technologies. Volume 2655 of Lecture Notes in Computer Science., Springer (2003) 368–381
16. Chan, K.P., Chen, T.Y., Kuo, F.C., Towey, D.: A revisit of adaptive random testing by restriction. In: Proceedings of the 28th International Computer Software and Applications Conference (COMPSAC 2004), IEEE Computer Society (2004) 78–85
17. Chen, T.Y., Eddy, G., Merkel, R., Wong, P.K.: Adaptive random testing through dynamic partitioning. In: Proceedings of the 4th International Conference on Quality Software (QSIC 2004), IEEE Computer Society (2004) 79–86
18. Chen, T.Y., Huang, D.: Adaptive random testing by localization. In: Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC 2004), IEEE Computer Society (2004) 292–298
19. Chen, T.Y., Leung, H., Mak, I.K.: Adaptive random testing. In Maher, M.J., ed.: Proceedings of the 9th Asian Computing Science Conference (ASIAN 2004). Volume 3321 of Lecture Notes in Computer Science., Springer (2004) 320–329

Interactive Testing with HOL-TestGen

Achim D. Brucker and Burkhart Wolff

Information Security, ETH Zürich, ETH Zentrum, CH-8092 Zürich, Switzerland
{brucker, bwolff}@inf.ethz.ch

Abstract. HOL-TestGen is a test environment for specification-based unit testing build upon the proof assistant Isabelle/HOL. While there is considerable skepticism with regard to interactive theorem provers in testing communities, we argue that they are a natural choice for (automated) symbolic computations underlying systematic tests. This holds in particular for the development on non-trivial formal test plans of complex software, where some parts of the overall activity require inherently guidance by a test engineer. In this paper, we present the underlying methods for both black box and white box testing in interactive unit test scenarios. HOL-TestGen can also be understood as a unifying technical and conceptual framework for presenting and investigating the variety of unit test techniques in a logically consistent way.

Keywords: symbolic test case generations, black box testing, white box testing, theorem proving, interactive testing.

1 Introduction

HOL-TestGen [1, 5, 6] is a test environment for unit testing based on the proof assistant Isabelle/HOL. Its design rationale is remarkably different from the mainstream of other symbolic testing tools which are designed to be fully automatic: In our view, the development of tests is an *interactive* activity, where the form of test specifications, the abstraction levels used in a test, the solution of generated logical constraints (for path-conditions, etc.), and the parameters of the test data selection must be experimented with and adopted up to the point where the generated tests are sufficiently "good" with respect to an underlying test adequacy criteria.

Aiming at a fully automatic tool for specification-based test has a number of consequences: be it for the specification language and for the degree of abstraction of test specifications, be it for the theories of the underlying data structures, and be it, last but not least, on the way how not automatically resolvable logical constraints of a test are finally treated. It may be the case that most of the generated constraints can (and, of course, should!) be solved by an automated theorem prover or constraint solver, however, what happens to the remaining rest? Raising this question and putting the interactive element from the periphery into the center leads in our experience to different answers with respect to the usable specification language, to the design of the system architecture as well as to the testing methodology.

W. Grieskamp and C. Weise (Eds.): FATES 2005, LNCS 3997, pp. 87–102, 2006.

This paper focuses on the latter issue. HOL-TestGen has been used to find non-trivial bugs in "real" software based on highly automated symbolic computation processes [6], where theory and implementation are described. But what is the underlying methodology leading to this result? And are there other ways to profit from the inactive potentials of HOL-TestGen? We answer these questions in the main sections of this paper: in Sec. 4, we describe the "best practices" developed in previous specification-based black box tests. In particular, we show how the highly automated standard workflow for generating test data can be enhanced by mixing it with more or less ingenious intermediate theorem proving steps. In Sec. 5, we will exploit the underlying generality of Isabelle for a different testing technique in the style of Pathfinder [11], SpecExplorer [8], and Korat [4]. The approach is based on a suitable semantic presentation of a programming language (a "logical embedding"), which can be used to both derive semantic constraints underlying a test as well as solving them in an integrated way. Moreover, our approach allows for logging explicit test hypotheses, a concept developed by the authors [6].

We would like to emphasize that all our symbolic computations are based entirely on conservative theory extensions of HOL and derived rules from them, such that HOL-TestGen is in fact a proven correct tool (assuming the consistency of HOL and its correct implementation in Isabelle). We believe that proving the crucial rules helps to develop a simple, semantically clean and integrated support of testing techniques.

The contributions and the plan of this paper are as follows: First, we outline an overall system presentation of HOL-TestGen (Sec. 3), second, we will develop the interactive methodology of generating specification-based black box tests (Sec. 4), and finally, we develop a proof of concept for white box testing in our framework. In particular, we show how our concept of explicit test hypotheses can be applied in this context (Sec. 5).

2 Foundations

2.1 Isabelle

Isabelle [10] is a *generic* theorem prover. New object logics can be introduced by specifying their syntax and inference rules. Among other logics, Isabelle supports first order logic, Zermelo-Fränkel set theory and HOL, which we choose as framework for HOL-TestGen.

While Isabelle/HOL is usually coined as "proof assistant", we use it as symbolic computation environment. Implementations on Isabelle/HOL can re-use existing powerful deduction mechanisms such as higher-order resolution and rewriting, and the overall environment provides a large collection of components ranging from documentation generators and code-generators to (generic) decision procedures for datatypes and Presburger Arithmetic.

Isabelle can easily be controlled by a programming interface on its implementation level in SML in a logically safe way, as well as in the Isar level, i.e., a tactic proof language in which interactive and automated proofs can be mixed arbitrarily. Documents in the Isar format, enriched by the commands provided by HOL-TestGen, can be processed incrementally within Proof General (see Sec. 3) as well as in batch mode. These documents can be seen as formal and technically checked test plan of a program under test.

2.2 Higher-Order Logic

Higher-order logic (HOL) [7, 3] is a classical logic with equality enriched by total polymorphic[1] higher-order functions. It is more expressive than first-order logic, since e.g., induction schemes can be expressed inside the logic. Pragmatically, HOL can be viewed as a combination of a typed functional programming language like SML or Haskell extended by logical quantifiers. Thus, it often allows a very natural way of specification.

Isabelle/HOL provides also a large collection of theories like sets, lists, multisets, orderings, and various arithmetic theories. Furthermore, it provides the means for defining data types and recursive function definitions over them in a style similar to a functional programming language.

Isabelle/HOL processes rules and theorems of the form $A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow A_{n+1}$, also denoted as $[\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1}$. They can be understood as a rule of the form "from assumptions A_1 to A_n, infer conclusion A_{n+1}". In particular, the presentation of sub-goals uses this format. We will refer to assumptions also as *constraints* in this paper.

3 The HOL-TestGen System

HOL-TestGen is an *interactive* (semi-automated) test tool for specification based unit tests. Its theory and implementation has been described in [5], here, we briefly review main concepts and outline the standard workflow. The latter is divided into four phases: writing the *test specification*, generation of *test cases* along with a *test theorem*, generation of *test data* (TD), and the *test execution* (*result verification*) phase involving runs of the "real code" of the program under test. Once a test theory is completed, documents can be generated that represent a formal test plan. See Fig. 1 for the overall workflow.

The properties of *program under test* are specified in HOL in the *test specification* (TS). The system will decompose the test specification in the test case generation phase into a semantically equivalent *test theorem* which has the form:

$$[\![\mathsf{TD}_1; \ldots; \mathsf{TD}_n; \mathsf{THYP}\ \mathsf{H}_1; \ldots; \mathsf{THYP}\ \mathsf{H}_m]\!] \Longrightarrow \mathsf{TS}$$

where THYP is a constant (semantically: an identity) used to mark the *test hypotheses* that are underlying this test. At present, HOL-TestGen uses only uniformity and regularity Hypothesis; for example, a uniformity hypothesis means informally "if the program conforms to one instance of a case to TS, it conforms

[1] To be more specific: *parametric polymorphism.*

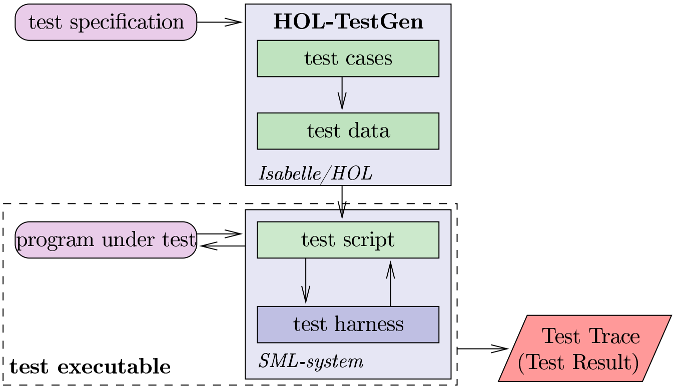
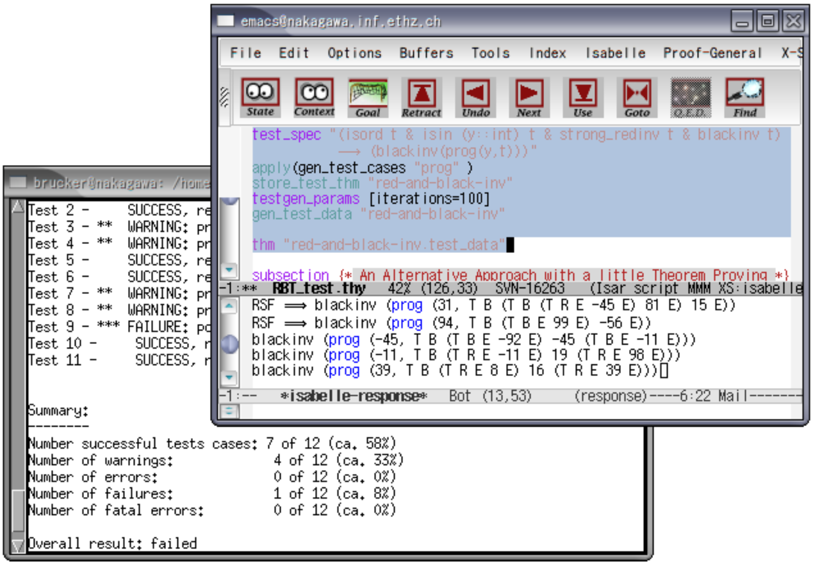


Fig. 1. Overview of the Standard Workflow of HOL-TestGen

to all instances of this case to TS" (see Sec.4.2 for a formal presentation). Thus, a test theorem has the following meaning:

> *If the program under test passes the tests for all* TD_i *successfully, and if it satisfies all test hypothesis, it conforms to the test specification.*

In this sense, a test theorem bridges the gap between test and verification. h The theory containing test theory, test specifications, configurations of the test data and test script generation, possibly extended by proofs for rules that support


emacs@nakagawa.inf.ethz.ch
File Edit Options Buffers Tools Index Isabelle Proof-General X-S
State Context Goal Retract Undo Next Use Goto Q.E.D. Find
brucker@nakagawa: /home

```
test_spec "(isord t & isin (y::int) t & strong_redinv t & blackinv t)
        ⟶ (blackinv(prog(y,t)))"
apply(gen_test_cases "prog" )
store_test_thm "red-and-black-inv"
testgen_params [iterations=100]
gen_test_data "red-and-black-inv"

thm "red-and-black-inv.test_data"

subsection {* An Alternative Approach with a little Theorem Proving *}
-1:** RBT_test.thy   42% (126,33)   SVN-16263   (Isar script MMM XS:isabelle
```

```
RSF ⟹ blackinv (prog (31, T B (T B (T R E -45 E) 81 E) 15 E))
RSF ⟹ blackinv (prog (94, T B (T B E 99 E) -56 E))
blackinv (prog (-45, T B (T B E -92 E) -45 (T B E -11 E)))
blackinv (prog (-11, T B (T R E -11 E) 19 (T R E 98 E)))
blackinv (prog (39, T B (T R E 8 E) 16 (T R E 39 E)))
-1:--   *isabelle-response*   Bot  (13,53)     (response)----6:22 Mail-------
```

```
Test 2 -      SUCCESS, re
Test 3 - **   WARNING: pr
Test 4 - **   WARNING: pr
Test 5 -      SUCCESS, re
Test 6 -      SUCCESS, re
Test 7 - **   WARNING: pr
Test 8 - **   WARNING: pr
Test 9 - ***  FAILURE: po
Test 10 -      SUCCESS, r
Test 11 -      SUCCESS, r


Summary:
--------
Number successful tests cases: 7 of 12 (ca. 58%)
Number of warnings:             4 of 12 (ca. 33%)
Number of errors:               0 of 12 (ca. 0%)
Number of failures:             1 of 12 (ca. 8%)
Number of fatal errors:         0 of 12 (ca. 0%)

Overall result: failed
```

Fig. 2. A HOL-TestGen Session Using Proof General

the overall process, is written in an extension of the Isar language [12]. It can be processed in batch mode, but also using the Proof General interface interactively, see Fig. 2. This interface allows for interactively stepping through a test theory (in the upper sub-window) and the sub-window below shows the corresponding system state. A system state may be a proof state in a test theorem development, or the result of inspections of generated test data or a list of test hypothesis.

After test data generation, HOL-TestGen can produces a *test script* driving the test using the provided *test harness*. The test script together with the test harness stimulate the code for the program under test built into the *test executable*. Executing the *test executable* runs the test and results in a *test trace* showing possible errors in the implementation (see lower window in Fig. 2).

4 Interactive Black Box Testing

In this section we present the method for the current main application of HOL-TestGen: generating test data for black box testing of side-effect free programs. As running example we chose the red-black trees already used in [5] to find an error in the "real" sml/NJ library. However, this time we will show *how* errors were found and how test data can be generated that actually explores the program under test to a satisfactory degree.

4.1 The Test Specification

Red-black trees store the balancing information in one additional bit per node, which is called the "color of a node". This is either red or black. A valid (balanced) red-black tree must fulfill the following three invariants:

1. *Red Invariant:* each red node has a black parent.
2. *Strong Red Invariant:* the root is red *and* the red invariant holds.
3. *Black Invariant:* each path from the root to a leaf has the same number of black nodes.

An invariant can be represented as recursive predicate; for the red invariant this looks as follows:

```
types     'a item = "'a :: ord_key"
datatype  color = R | B
datatype 'a tree = E | T color "'a tree" "'a item" "'a tree"


consts redinv :: "'a tree ⇒ bool"
recdef redinv "measure (λt. (size t))"
              "redinv E = True"
              "redinv (T B a y b) = (redinv a ∧ redinv b)"
              "redinv (T R (T R a x b) y c) = False"
              "redinv (T R a x (T R b y c)) = False"
              "redinv (T R a x b) = (redinv a ∧ redinv b)"
```

Assume we want to test that insertion or deletion (summarized by the placeholder prog) fulfill the black invariant. Hence, we are searching for test data fulfilling the premise of the following test specification:

```
test_spec "(isord t∧isin y t∧strong_redinv t∧ blackinv t)⟶ (blackinv(prog(y,t)))"
```

which we found after some experimenting (weaker preconditions lead to understandable exceptions of the program under test).

4.2 First Attempt: The "Standard Workflow"

Test Case Generation. Now we can automatically generate *test cases* in a model checking-like fashion by applying the gen_test_cases method. The method generates data-structures (here: trees) up to a certain depth and performs case splitting over all possible cases; remaining constraints are simplified. The default depth-parameter of the method is set to 3. Finally, the resulting test theorem is stored in a *test environment*:

```
  apply( gen_test_cases "prog")
store_test_thm " red_and_black_inv"
```

This fairly simple setup generates already 25 subgoals containing 12 test cases, altogether with non-trivial constraints, among them:

1. $[\![\ x_1{=}x_2\ ]\!] \Longrightarrow \mathsf{blackinv}\ (\mathsf{prog}\ (x_1,\ \mathsf{T\ B\ E}\ x_2\ \mathsf{E}))$
2. $[\![\ x_1{=}x_6;\ \mathsf{max\_B\_height}\ (\mathsf{T}\ x_5\ x_4\ x_3\ x_2) = 0;\ \mathsf{blackinv}\ x_2;$
   $\mathsf{max\_B\_height}\ x_4 = \mathsf{max\_B\_height}\ x_2;\ \mathsf{blackinv}\ x_4;\ \mathsf{redinv}\ (\mathsf{T}\ x_5\ x_4\ x_3\ x_2);$
   $\forall x.\ (x{=}\ x_3 \longrightarrow x_6 < x) \wedge (\mathsf{isin}\ x\ x_4 \longrightarrow x_6 < x) \wedge (\mathsf{isin}\ x\ x_2 \longrightarrow x_6 < x);$
   $\forall x.\ \mathsf{isin}\ x\ x_2 \longrightarrow x_3 < x;\ \forall x.\ \mathsf{isin}\ x\ x_4 \longrightarrow x < x_3;\ \mathsf{isord}\ x_2;\ \mathsf{isord}\ x_4]\!]$
   $\Longrightarrow \mathsf{blackinv}\ (\mathsf{prog}\ (x_1,\ \mathsf{T\ B\ E}\ x_6\ (\mathsf{T}\ x_5\ x_4\ x_3\ x_2)))$

An example for a generated uniformity test hypothesis is:

$$\mathsf{THYP}\ ((\exists x\ xa.\ x = xa \longrightarrow \mathsf{blackinv}\ (\mathsf{prog}\ (x,\ \mathsf{T\ B\ E}\ xa\ \mathsf{E}))) \longrightarrow$$
$$(\forall x\ xa.\ x = xa \longrightarrow \mathsf{blackinv}\ (\mathsf{prog}\ (x,\ \mathsf{T\ B\ E}\ xa\ \mathsf{E}))));$$

Test Data Generation. Generating concrete test data already takes a remarkable length of time, as it's quite unlikely that the random solver generates values that fulfill these ordering constraints. Therefore we restrict the attempts (*iterations*) the random solver takes for solving a single test case to 40

```
testgen_params [iterations =40]
gen_test_data " red_and_black_inv"
```

which is sadly not sufficient to solve all conditions, e.g., we obtain test cases like

```
RSF ⟶ blackinv (prog (100, T B E 7 E))
RSF ⟶ blackinv (prog (83, T B (T B (T B E −8 E) 57 (T R E 13 E)) −62 E))
blackinv (prog (−91, T B (T R E −91 E) 5 E))
RSF ⟶ blackinv (prog (−33, T B (T R E −2 E) 37 E))
```

were RSF marks unsolved cases. Analyzing the generated test data reveals that only very few had been resolved and therefore lead to inconclusive tests. To compute more conclusive test data, we can interactively increase the number of iterations, which reveals that we need to set iterations to more than 100 to find a suitable set of test data reliably.

Test Script Generation. Now we generate the test script for PUT being implemented by wrapper.del:

```
gen_test_script " rbt_script .sml" " red_and_black_inv" "PUT" "wrapper.del"
```

In principle, any SML-system should be able to run the provided test-harness and generated test-script. Using their specific facilities for calling foreign code, testing of non-SML implementations, e.g., Java, or C, is supported. Depending on the SML-system, the test execution can be done within an interpreter or using a compiled test executable. Testing implementations written in SML is straightforward. For testing non-SML implementations it is in most cases sufficient to provide a quite simple "wrapper" doing some datatype conversion.

Test Result Verification. Running the test executable for red_and_black_inv results in an output similar to Tab. 1, showing successful test cases, failures (i.e., the implementation violates the post condition) and warning caused by unresolved cased (where the random solver returns RSF as pre-condition). In the latter case, the PUT is still executed (and throws in our example an exception). Already in this highly automatic set-up, we were able to produce the reported error in the SML library. However, based on the test depth 3 (which represents the limit of the standard approach if we restrict ourselves to a time investment of 10 minutes for the overall run) we cannot have trees with more than three nodes on the level of the test case generation. Of course, random solving increases the

Table 1. RBT Test trace

```
Test Results:
 Test  0 -     SUCCESS, result: E
 Test  1 -     SUCCESS, result: T(R,E,67,E)
 Test  2 -     SUCCESS, result: T(B,E,~88,E)
 Test  3 - **  WARNING: pre cond. false (exception during post cond.)
 Test  4 - **  WARNING: pre cond. false (exception during post cond.)
 Test  5 -     SUCCESS, result: T(R,E,30,E)
 Test  6 -     SUCCESS, result: T(B,E,73,E)
 Test  7 - **  WARNING: pre cond. false (exception during post cond.)
 Test  8 - **  WARNING: pre cond. false (exception during post cond.)
 Test  9 - *** FAILURE: post cond. false, result: T(B,T(B,E,~92,E),~11,E)
 Test 10 -     SUCCESS, result: T(B,E,19,T(R,E,98,E))
 Test 11 -     SUCCESS, result: T(B,T(R,E,8,E),16,E)

Summary:
 Number successful tests cases: 7 of 12 (ca. 58%)
 Number of warnings:            4 of 12 (ca. 33%)
 Number of errors:              0 of 12 (ca. 0%)
 Number of failures:            1 of 12 (ca. 8%)
 Number of fatal errors:        0 of 12 (ca. 0%)
```

depth of the trees sporadically, as can be seen from the test result, but in an unsystematic way. Thus, the program under test has obviously not been tested satisfactorily, and we need means to treat test data sets with higher depth.

4.3 Second Attempt: Using "Abstract Test Data"

By inspection of the constraints of the test theorem, one immediately identifies predicates for which solutions are difficult to find by a random process (a measure for this difficulty could be the percentage of trees up to depth k, that make this predicate valid. One can easily convince oneself, that this percentage is decreasing asymptotically).

Repeatedly, ground instances were needed for terms of the form:

1. max_B_height x = 0
2. max_B_height y = max_B_height z
3. blackinv x
4. redinv x
5. isord x

How can the constraint resolution be helped by user guidance? The idea is to establish ground instances by hand and to feed them into the resolution process as *abstract test cases*, see [6]. But which of the patterns should we choose? It turns out that max_B_height X = 0 (is the number of black nodes on any path 0?) has many candidates, but after depth 2 they are all ruled out by redinv). Thus, we picked redinv and provided ground instances for it by hand: redinv E, redinv (T R E (5::int) E), redinv (T B E (5::int) E), redinv (T R E 2 (T B E (5::int) E)), redinv ((T R (T B E (5::int) E) 6 E)), redinv(T R (T B E 3 E)4 (T B E (5::int) E)), etc. Each of these ground instances is in fact established by an automatic proof:

```
lemma redinv_6[test "red_and_black_inv"]:
    "redinv(T R (T B E 3 E) 4 (T B E (5::int) E))"  by auto
```

The pragma [test "red_and_black_inv"] is used to associate this theorem as abstract test data to the data generation

```
gen_test_data "red_and_black_inv"
```

An analysis of the test results (omitted here for space reasons) reveals that the tests are now a more complete set of trees of depth 4.

Note, however, that the "samples" of abstract data had been chosen with hindsight to the overall test: they all represent ordered trees that happen to fulfill the black invariant, generated within the time frame of 10 minutes as in the previous run. Abstract test data that do not fulfill all the other possible constraints represent dead ends and are no help for the constraint solving phase.

4.4 Third Approach: Using a Little Theorem Proving

The question arises how this problematic aspect of ingeniously added abstract test data can be overcome and be systematized for our example. One answer is a characterization theorem of redinv:

```
lemma redinv_enumerate:
  "redinv x =((x = E)
               ∨ (∃ a y b. x = T B a y b ∧redinv a ∧ redinv b)
               ∨ (∃ y. x = T R E y E)
               ∨ (∃ y am an ao. x = T R E y (T B am an ao) ∧
                                redinv (T B am an ao))
               ∨ (∃ ae af ag y. x = (T R (T B ae af ag) y E)
                                 ∧ redinv (T B ae af ag))
               ∨ (∃ ae af ag y T B bg bh bi.
                               x = (T R (T B ae af ag) y (T B bg bh bi)) ∧
                               (redinv (T B ae af ag) ∧ redinv (T B bg bh bi))))"
```

The precise form of this lemma can be inferred when inspecting the rule set generated by Isabelle from the redinv-definition. The proof is a routine induction proof which nevertheless needs knowledge about theorem proving in general and Isabelle in particular. This lemma is used to improve the form of the test theorem. To be a bit more precise, we insert after the test case generation a sequence of Isar-methods that resolve in any constraint of the form redinv x the above lemma, recomputes the TNF and repeats this process once. The resulting test is now of depth 5 and constitutes now a quite extensive test of our program (again in the time-frame of 10 minutes for a complete run).

4.5 Summing Up

In our experience, increasing the number of iterations also increases remarkably the time needed for test data generation. On the other side, this underpins the usual criticism with respect to random testing: deeply nested (either in the sense of data or execution paths) program structures cannot be tested seriously using pure random tests; guidance generated by test cases is crucially needed.

Further, our results show that highly automated approaches yield useful "first shots" but heavily profit from more or less ingenious user interaction. A trade-off must be made here between the time needed to run a test (including generation), the quality of the test and the time and experience needed in advanced techniques such as Isabelle theorem proving.

5 Imperative White Box Tests

Our framework is not restricted to black box test of side-effect free programs. Using a *logical embedding* (a representation in HOL comprising syntax and semantics) for an imperative language, it can be used to implement and analyze various white-box test techniques.

5.1 The Language IMP: An Overview

The Isabelle distribution comes already with various logical embeddings: IMP, IMPP, NanoJava, or MicroJava, and more are available in the literature. For the sake of this presentation, we chose the simplest one, IMP, which is intended as formalization of a textbook on programming language semantics [13, 9], and provides as such a particularly clean and complete collection of several semantics of IMP (natural semantics, transition semantics, denotational semantics, axiomatic semantics), proofs of their relations (e.g., denotational is equivalent to natural) and proofs of crucial meta-properties (axiomatic semantics is sound and relative complete).

The basic concepts of IMP are *values* val (just natural numbers, for example), and *states* state = loc ⇒ val. *Boolean expressions* bexp and *atomic expressions* (aexp) are represented as functions from state to val or bool. Thus, IMP has in fact no syntax of its own, but just inherits the expression language of HOL at this place[2]. The syntax of IMP commands com is then defined as data type:

```
datatype com = SKIP
             | ":==" loc aexp          ( infixl 60)
             | Semi com com            ("_ ; _" [60, 60] 10)
             | Cond bexp com com       (" IF _ THEN _ ELSE _" 60)
             | While bexp com          (" WHILE _ DO _" 60)
```

where the text in the parenthesis are just pragmas for the powerful Isabelle syntax engine to allow the usual infix/mixfix notation.

One of the operational semantics of IMP is a relation of triples evalc :: (com ×state × state) set ((cm,s,s') ∈ evalc is denoted $\langle \mathsf{cm,s}\rangle \xrightarrow[c]{} \mathsf{s'}$) which is inductively defined as follows:

inductive evalc **intros**

"$\langle \mathtt{SKIP}, s\rangle \xrightarrow[c]{} s$"
"$\langle x :== a, s\rangle \xrightarrow[c]{} s[x{:=}(a\ s)]$"
"$[\![\ \langle c_0, s\rangle \xrightarrow[c]{} s_1;\ \langle cs_1, s_1\rangle \xrightarrow[c]{} s_2\ ]\!] \Longrightarrow \langle c_0; cs_1,\ s\rangle \xrightarrow[c]{} s_2$"
"$[\![\ b\ s;\ \langle c_0, s\rangle \xrightarrow[c]{} s_1\ ]\!] \Longrightarrow \langle\ \mathtt{IF}\ \ b\ \mathtt{THEN}\ c_0\ \mathtt{ELSE}\ c\_1,\ s\rangle \xrightarrow[c]{} s_1$"
"$[\![\ \neg b\ s;\ \langle c_1, s\rangle \xrightarrow[c]{} s_1\ ]\!] \Longrightarrow \langle\ \mathtt{IF}\ \ b\ \mathtt{THEN}\ c_0\ \mathtt{ELSE}\ c_1,\ s\rangle \xrightarrow[c]{} s_1$"
"$[\![\neg b\ s]\!] \Longrightarrow \langle\ \mathtt{WHILE}\ \ b\ \mathtt{DO}\ \ c,\ s\rangle \xrightarrow[c]{} s$"
"$[\![\ b\ s;\ \langle c, s\rangle \xrightarrow[c]{} s_1;\ \langle\ \mathtt{WHILE}\ \ b\ \mathtt{DO}\ \ c, s_1\rangle \xrightarrow[c]{} s_2]\!] \Longrightarrow \langle\ \mathtt{WHILE}\ \ b\ \mathtt{DO}\ \ c,\ s\rangle \xrightarrow[c]{} s_2$"

The usual notation s[x:=v] is defined by λ y. if y=x then v else s y. For these inductive rules, an alternative rule set is derived that can be processed by the efficient Isabelle rewriter directly:

$\langle\ \mathtt{SKIP}, s\rangle \xrightarrow[c]{} s' = (s' = s)$
$\langle\ x :== a,\ s\rangle \xrightarrow[c]{} s' = (s' = s[x := a\ s])$
$b\ s \Longrightarrow \langle\ \ \mathtt{IF}\ \ b\ \mathtt{THEN}\ d\ \mathtt{ELSE}\ e, s\rangle \xrightarrow[c]{} s' = \langle\ d, s\rangle \xrightarrow[c]{} s'$
. . .

We omit the definition of the denotational semantics reflecting the partial correctness C :: com ⇒(state × state) set (see [2] for details), but it is linked to the operational semantics via the theorem $((s,\ t) \in C\ c) = \langle c,s\rangle \xrightarrow[c]{} t$. On the denotational level, program transformation rules relevant for the next section can be shown easily:

[2] This technique is also called a “shallow embedding”.

```
C(SKIP;c) = C(c)          C(c;SKIP) = C(c)        C((c;d);e) = C(c;(d;e))
C((IF b THEN c ELSE d);e) = C(IF b THEN c;e ELSE d;e)
C(WHILE b DO c)           = C(IF b THEN c; WHILE b DO c ELSE SKIP)
```

On the level of the denotational semantics, the usual notion of "valid Hoare triple" is formalized as:

```
|= {P} c {Q} ≡ ∀s t. (s, t) ∈ C c ⟶ P s ⟶ Q t
```

where P, Q are *assertions*, i.e., functions from state to bool.

5.2 Unwinding IMP Programs

To perform white box tests in the style of Pathfinder [11], SpecExplorer [8], or Korat [4], it is necessary to make the program paths explicit in the program representation and amenable to the rules of the operational semantics. Therefore, a pre-processing step is necessary that unfolds all WHILE-loops up to a certain limit, the *unwind-factor k*. This principle can also be applied in a language extension with procedure calls such as IMPP, also available in the Isabelle distribution. Additionally, the program should be transformed into a certain normal form to be efficiently processed (left associative sequential compositions must be avoided since they lead to an existentially quantified intermediate states which are more difficult to process in the symbolic computation). We define two recursive functions on com-terms that perform both these normalizations as well as the unwinding up to k. Note, that we will not program this function outside the logic as (tactic), i.e., a control program in SML, but inside HOL, such that we can also prove its correctness with respect to the IMP semantics:

```
consts "@@" :: "[com,com] ⇒com" (infixr 70)
primrec "SKIP @@ c = c"
        "(x:== E) @@ c = ((x:== E); c)"
        "(c;d) @@ e = (c; d @@ e)"
        "( IF b THEN c ELSE d) @@ e = ( IF b THEN c @@ e ELSE d @@ e)"
        "( WHILE b DO c) @@ e = ((WHILE b DO c);e)"


consts unwind :: "nat ×com ⇒com"
recdef unwind "less_than <*lex*> measure(λ s. size s)"
  "unwind(n, SKIP)      = SKIP"
  "unwind(n, a :== E) = (a :== E)"
  "unwind(n, IF b THEN c ELSE d) = IF b THEN unwind(n,c) ELSE unwind(n,d)"
  "unwind(n, WHILE b DO c) =
        (if 0 < n
         then IF b THEN unwind(n,c)@@unwind(n− 1,WHILE b DO c) ELSE SKIP
         else WHILE b DO unwind(0, c))"
  "unwind(n, SKIP ; c) = unwind(n, c)"
  "unwind(n, c ; SKIP) = unwind(n, c)"
  "unwind(n, ( IF b THEN c ELSE d) ; e) =
             ( IF b THEN (unwind(n,c;e)) ELSE (unwind(n,d;e)))"
  "unwind(n, (c ; d); e) = (unwind(n, c;d))@@(unwind(n,e))"
  "unwind(n, c ; d) = (unwind(n, c))@@(unwind(n, d))"
```

The primitive recursive auxiliary function c@@d appends a command d to the last command in c that is reachable from the root via sequential composition modes. The more tricky unwind function unfolds WHILE-loops as long as the unwind factor is positive and performs the program normal form computation along the program equivalences as discussed in Sec. 5.1.

The Isabelle Recursion Package adopts a "First Fit" pattern matching strategy (similar to SML). This means that in overlapping cases, the first is taken into account with higher priority—this is reflected on the level of the rewrite rule set generated from this definition. Thus, the last equation in the recursive definition is a catch-all rule for sequential composition.

Now we derived the following facts over these definitions:

Lemma 1 (Termination). *Both functions terminate.*

Proof. In the case of @@ this is trivial due to machine checked primitive recursion; in case of unwind a proof has to be performed that the lexicographic composition of the standard ordering $\_ < \_$ and the standard term ordering is well-founded and respected by the inner calls in this recursive definition. This proof is done fully automatically.

Lemma 2 (Correctness). *C(c @@ d)= C(c;d) and C(unwind(n,c)) = C(c)*

Proof. For @@, a straight-forward induction suffices. As for unwind, the proof is non-trivial, but routine (generalization over n, induction over c, intricate case splitting, application of semantic equivalences of Sec. 5.1).

5.3 Generating Path Conditions

As example program, we chose a little program that computes the square-root of a natural number. In Isabelle/IMP syntax, we can define it as follows:

```
constdefs squareroot :: "[loc, loc, loc, loc] ⇒ com"
          "squareroot tm sum i a ≡ ((  tm   :== λs. 1);
                                   (( sum  :== λs. 1);
                                    ((i    :== λs. 0);
                                      WHILE  λs. (s sum) <= (s a) DO
                                       ((  i    :== λs. (s i) + 1);
                                        ((tm   :== λs. (s tm) + 2);
                                         (sum  :== λs. (s tm) + (s sum)))))))
```

where the locations (references) are the input into the program to express semantically constraints on them, as we will see later. The shallow embedding of the expressions has the consequence that program variable accesses must be represented as explicit application of the state s (at this program point) to a location representing this variable. Hence, we implicitly require a pre-parser that makes these bindings of program variables explicit.

We need one further derived rule If_split, which is necessary to expand the case splits produced for each path:

$$[\![\mathsf{b\ s} \Longrightarrow \langle \mathsf{c,s}\rangle \xrightarrow[c]{} \mathsf{s'};\ \neg\, \mathsf{b\ s} \Longrightarrow \langle \mathsf{d,s}\rangle \xrightarrow[c]{} \mathsf{s'}]\!] \Longrightarrow \langle\ \mathtt{IF}\ \ \mathsf{b}\ \mathtt{THEN}\ \mathsf{c}\ \mathtt{ELSE}\ \mathsf{d,s}\rangle \xrightarrow[c]{} \mathsf{s'}$$

Putting everything together, we can now formulate the generation of symbolic states for the program squareroot as follows:

lemma derive_test_cases : **assumes** no_alias : . . .
 shows "$\langle \mathsf{unwind(3,\ squareroot\ \ tm\ sum\ i\ a),\ \ s} \rangle \xrightarrow[c]{} \mathsf{s'}$"

where the omitted technical side-condition no_alias specifies that the locations tm,sum,i,a are pairwise disjoint. Now, the canonical tactic script:

```
apply(simp add: squareroot_def )
apply( rule  If_split , simp_all add: update_def no_alias )+
```

unfolds the definition of squareroot, and then enters in a loop that performs the computation of unwind (including path normalization), the case splitting along the If_split rule discussed above, the evaluation of state constraints and the simplification of the arithmetic constraints until no further changes can be achieved. The resulting proof-state consists of the following goals:[3]

$$\begin{array}{lll}
1. & 9 \le \mathsf{s\ a} \Longrightarrow \langle\ \texttt{WHILE}\ \ \lambda \mathsf{s.\ s\ sum} \le \mathsf{s\ a} & \\
 & \qquad\qquad \texttt{DO}\ \ \mathsf{i} :== \lambda \mathsf{s.\ Suc\ (s\ \ i)\ ;} & \\
 & \qquad\qquad\quad \mathsf{(tm} :== \lambda \mathsf{s.\ Suc\ (Suc\ (s\ tm))\ ;} & \\
 & \qquad\qquad\qquad \mathsf{sum} :== \lambda \mathsf{s.\ s\ tm} + \mathsf{s\ sum\ \ ),} & \\
 & \qquad\qquad \mathsf{s(\,i\ := 3,\ \ tm := 7,\ sum := 16)} \rangle \xrightarrow[c]{} \mathsf{s'} & \\
2. & [\![4 \le \mathsf{s\ \ a;}\ \ 8 < \mathsf{s\ a}\ ]\!] & \Longrightarrow \mathsf{s'\ = s\ (\,i\ := 2,\ \ tm := 5,\ sum := 9)} \\
3. & [\![\ 1 \le \mathsf{s\ \ a;\ \ s\ \ a} < 4]\!] & \Longrightarrow \mathsf{s'\ = s\ (\,i\ := 1,\ \ tm := 3,\ sum := 4)} \\
4. & \mathsf{s\ \ a} = 0 & \Longrightarrow \mathsf{s'\ = s(tm := 1,\ sum := 1,\ \ i\ := 0)}
\end{array}$$

The resulting proof state enumerates the possible symbolic states including their path conditions.[4]

5.4 Treating Assertions and Test Hypothesises

Traditional pre and post conditions can be expressed via the validity relation for Hoare Triple, e.g.: $\models \{\mathsf{pre}\}$ squareroot tm sum i a $\{\mathsf{post\ \ a\ \ i}\}$ where pre is just λx. True and post a i is λ s. $\mathsf{(s\ \ i){*}(s\ \ i)} \le \mathsf{(s\ a)} \wedge \mathsf{s\ \ a} < \mathsf{(s\ \ i\ } + 1\mathsf{){*}(s\ \ i\ } + 1\mathsf{)}$.

The setup of a specification based white box test is now produced by the derived rule:

$$\models \{\mathsf{P}\}\ \mathsf{c}\ \{\mathsf{Q}\} = \forall \mathsf{s\ t.}\ \ \langle \mathsf{unwind\ (n,\ \ c\,),s} \rangle \xrightarrow[c]{} \mathsf{t} \longrightarrow \mathsf{P\ s} \longrightarrow \mathsf{Q\ t}$$

The result of this rule application is piped into the previous process which conjoins the preconditions with the path conditions and attempts to solve them; the post condition is then constructed over the post state constructed by the natural semantics.

Assertions can be introduced into our language as follows: First, we declare an uninterpreted constant STOP as command of the language. Then, a construct like ASSERT b c can be introduced as abbreviation for ASSERT b c ≡ IF b THEN c ELSE STOP, and further constructs like an annotated while loop AWHILE b inv c are introduced analogously.

[3] The presentation has been slightly syntactically simplified.

[4] The computing time for unwind-factor 10 based on this simplistic tactic remains under a few seconds, including pretty-printing.

It remains to show how white box testing fits *methodically* into our framework, where we try to generate *test hypothesis* that make the “logical difference” between a test and the verification of the test specification explicit. Obviously, the only new element related to white box test is the unwinding parameter; if exhausted, this leads to program fragments that represent the “set of untested execution paths” of a program under test. In our running example, this lead to the first sub-goal in the final proof state. Turned into an explicit *unwinding k test hypothesis*, this condition for resulting from the test theorem: $\models$\{pre\} squareroot tm sum i a \{post a i\} looks as follows:

```
1. THYP(9 ≤s a ⟶ ⟨ WHILE  λs. s sum ≤s a
                    DO    i :== λs. Suc (s i) ;
                       (tm :== λs. Suc (Suc (s tm)) ;
                       sum :== λs. s tm + s sum ),
                   s(i := 3, tm := 7, sum := 16)⟩ ⟶c s'
                   ∧ post a i s')
```

Testing a program in this setting means that all symbolic state transitions including their path conditions must satisfy the post condition whenever the pre condition holds. This is the case in our example, and the system will find the satisfiability of the generated constraints without need for random solving in this case. The only remaining assumption is the test hypothesis shown above which reflects that we have tested the program and not verified it.

To sum up, we described a symbolic computation process for white box tests in the language IMP, that generates from a given, potentially annotated program a test theorem including the test hypotheses automatically. This test theorem can be fed into the test data generation phase to find ground instances for particular paths as before.

5.5 Blowing Up IMP

The reader might object that the language IMP, having only Boolean and arithmetic side-effect free expressions and non-recursive, macro-like procedures, is too academic to be of practical importance. In contrast, we argue that IMP is a reasonable core language which can be “blown-up” fairly easy to larger languages, in large parts without adding further complexity to the symbolic computation process presented so far.

We discuss three extensions of IMP, two more straight-forward, one more involved, to give an impression over the potential of our approach:

1. *Mutual recursion:* Just apply our approach to embedding IMPP.
2. *Arbitrary expressions:* exchange val in the IMP semantics by a universe which is a sum of the HOL data types.
3. *Objects:* Extend our approach to an embedding like NanoJava.

In more detail, extension 2 requires that program variables must be presented as triples $(\mathsf{loc},\mathsf{emb}::\alpha\longrightarrow\mathsf{val},\mathsf{proj}::\mathsf{val}\longrightarrow\alpha)$ consisting of the traditional location, and a pair of functions (representing the typing of the program and variable) that allow for injection and projection of HOL-values into the `val` universe of IMP. Program variable accesses, which has been encoded by `s a` so far, will be `s !a` where `s!(a,emb,prj)` is defined by `prj(s a)`. The assignment semantics of IMP must be adopted analogously. This technique paves the way for lists, options, strings, and further user-defined data types. Expressions over user-defined HOL data-types can now be processed by the `gen_test_cases`-method which is at the heart of HOL-TestGen. As a result of these extensions, we have an SML-like language with data-types and HOL-expressions inside.

The extension 3 involves sub-classing, method calls with late-binding and object creation; as such, a lot more machinery is therefore involved whose tactical control will be feasible in our opinion, but require substantial more work.

6 Conclusion

We have shown the pragmatics of our Isabelle/HOL-based testing tool HOL-TestGen [1, 5] gained from previous experiences for specification based black box tests. While some aspects of the symbolic computations are fully automatic (like data separation lemma generation, generation of test hypothesis, TNF-computations, test data generations and solving), other aspects like constraint solving may profit from some theorem proving and experiments with "appropriate" formulations of test specifications/test theorems. We have also developed a method to use HOL-TestGen for specification based white box tests.

The symbolic computation process is fully presented inside HOL, so no tool integration and conversion issues are involved which may be critical both for correctness and efficiency. Since the necessary symbolic transformation processes can be based on derived rules,[5] HOL-TestGen can be used as a tool for a seamless *conceptual study* of these techniques including formal correctness proofs, their *prototypical implementation* and even their *industry strength implementation.* The latter, will require substantial effort in tactic programming and tool integration.

Although the example for imperative white-box test is based on a conceptual language and therefore merely a proof of concept than a proof of technology, we believe that the approach can scale up with respect to size of the supported language while maintaining reasonable efficiency of the underlying symbolic computations. Thus, we believe that HOL-TestGen can be seen as unifying framework in which a wide range of unit test techniques can be presented in a mathematically clean way.

References

[1] HOL-TestGen. `http://www.brucker.ch/projects/hol-testgen/`.

[2] IMP. `http://isabelle.in.tum.de/library/HOL/IMP/Denotation.html`.

[3] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press, Orlando, May 1986.

[5] Inserting such rules as *axioms* is trivial, but endangers correctness, of course.

[4] 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*, pages 123–133, 2002.

[5] A. D. Brucker and B. Wolff. HOL-TestGen 1.0.0 user guide. Technical Report 482, ETH Zrich, Apr. 2005.

[6] A. D. Brucker and B. Wolff. Symbolic test case generation for primitive recursive functions. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Testing of Software*, number 3395 in Lecture Notes in Computer Science, pages 16–32. Springer-Verlag, Linz, 2005.

[7] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[8] W. Grieskamp, N. Tillmann, and M. Veanes. Instrumenting scenarios in a model-driven development environment. *Information and Software Technology*, 46(15):1027–1036, 2004.

[9] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.

[10] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[11] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engg.*, 10(2):203–232, 2003.

[12] M. M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents.* PhD thesis, TU München, München, Feb. 2002.

[13] G. Winskel. *The Formal Semantics of Programming Languages.* MIT Press, Cambridge, Massachusetts, 1993.

Conformance Testing Relations for Timed Systems[⋆]

Manuel Núñez and Ismael Rodríguez

Dept. Sistemas Informáticos y Programación,
Universidad Complutense de Madrid, E-28040 Madrid, Spain
{mn, isrodrig}@sip.ucm.es

Abstract. This paper presents a formal framework to test both the functional and temporal behaviors in systems where temporal aspects are critical. Different implementation relations, depending on both the interpretation of time and on the (non-)determinism of specifications and/or implementations, are presented and related. We also study how tests cases are defined and applied to implementations. A test derivation algorithm, producing sound and complete test suites, is presented.

1 Introduction

The complexity of current systems necessarily leads to a higher relevance of testing issues during the development project. The scale and heterogeneity of present projects makes it impossible for developers to have an overall view of the system. Thus, it is difficult to foresee those errors that are either critical or more probable. Since the construction of a system requires to use several components, developed by different teams, reliability of these components is a must. This is a requirement not only for final customers but also for developers. In this context, *formal testing techniques* provide systematic procedures to check implementations in such a way that the coverage of critical parts/aspects of the system depends less on the intuition of the tester.

The application of formal testing techniques to check the correctness of a system requires to identify the *critical* aspects of the system, that is, those aspects that will make the difference between correct and incorrect behavior. While the relevant aspects of some systems only concern *what* they do, in some other systems it is equally relevant *how* they do what they do. For instance, the probability of an event to happen may be considered critical in a non-deterministic system. If a system runs in an environment where computational resources are shared by several systems, the consumption of resources may be relevant as well. Similarly, the time consumed by each operation should be considered critical in

[⋆] Research partially supported by the Spanish *Ministerio de Ciencia y Tecnología* project MASTER (TIC2003-07848-C02-01), the *Junta de Comunidades de Castilla-La Mancha* project DISMEF (PAC-03-001), and the Marie Curie Research and Training Network *TAROT* (MRTN-CT-2003-505121).

W. Grieskamp and C. Weise (Eds.): FATES 2005, LNCS 3997, pp. 103–117, 2006.

a real-time system. Actually, some operations that are concluded after a given deadline could be useless or unacceptable for the user of the system.

In this paper we present a formal testing methodology where the temporal behavior of systems is considered. A simple extension of the classical concept of *Extended Finite State Machine* will allow a specifier to explicitly denote temporal requirements for each action of a system. We study *conformance testing* relations to relate implementations, belonging to a given set $\mathtt{Imp}$, with specifications, taken from another set $\mathtt{Spec}$. Our study considers time extensions of the relation $\mathtt{conf}_{nt}$ [NR02], which is based on the update of $\mathtt{conf}$ [BSS86] to deal with inputs and outputs: $\mathtt{ioco}$ [Tre96, Tre99]. In order to cope with time, we do not take into account only that a system may perform a given action but we also record the amount of time that the system needs to do so. Unfortunately, conformance testing relations for timed systems have not been yet extensively studied, and only very recently some work has been performed in this line (e.g. [NR02, BB04, LMN04]). We propose five timed conformance relations according to the interpretation of *good* implementation for a given specification. Regarding our relations, time aspects add some extra complexity. For example, even though an implementation I had the same traces as a specification S, we should not consider that I conforms to S if the implementation is always *slower* than the specification. Moreover, it can be the case that a system performs the same sequence of actions for different times. These facts motivate the definition of several conformance relations. For example, it can be said that an implementation conforms to a specification if the implementation is always *faster*, or if the implementation is at least as *fast* as the worst case of the specification. We think that the relations that we introduce in this paper can be useful for the study of conformance for other models of timed systems. For example, the definitions can be easily adapted to timed automata [AD94]. Other definitions of timed I/O automata (e.g. [HNTC99, SVD01]) are restricted to deterministic (regarding actions) behavior. In this case, some of our relations will be equivalent among them (i.e. they will relate the same automata).

Regarding the application of testing to timed systems, several proposals have appeared in the literature (e.g. [MMM95, CL97, HNTC99, SVD01, EDK02, ED03]). Our proposal differs from these ones in several points, mainly because the treatment of time is different. We do not have a notion of clock(s) together with time constraints; we associate time to the execution of actions (representing the time that it takes for a system to perform an action). Besides, the time that a transition needs to be performed is not fixed (e.g. the transition t takes 3 units of time). This time depends on the values of the variables. In fact, since those values may change after each transition, it may happen that if we perform two (or more) times a transition then each performance takes a different amount of time. With respect to the application of tests to implementations, the above mentioned non-deterministic temporal behavior of specifications and/or implementations requires that tests work in a specific manner. For example, if we apply a test and we observe that the implementation takes less time than the one required by the specification, then this single application of the test allows us to know that the implementation *may* be faster than the specification, but not that it *must* be so.

The rest of the paper is organized as follows. In Section 2 we present our model to represent timed systems. In Section 3 we study implementation relations for our framework where temporal behavior is taken into account. We also relate these implementation relations. In Section 4 we show how test cases are defined and describe how to apply them to implementations. In Section 5 we introduce a test derivation algorithm to produce sound and complete, with respect to three of our conformance relations, test suites. Next, in Section 6, we consider two additional relations that can be used when implementations show non-deterministic behavior. We also relate these new relations with the previous ones. Finally, in Section 7 we present our conclusions and some directions for further research.

2 A Timed Extension of the EFSM Model

In this section we introduce our timed extension of the classical extended finite state machine model. The main difference with respect to usual EFSMs consists in the addition of *time*. In order to represent our timed EFSMs we consider that the number of different variables is equal to m. We will assume that each variable x_i belongs to the domain D_i. Thus, the values of all the variables at a given point of time can be represented by a tuple belonging to the cartesian product $D_1 \times D_2 \times \cdots \times D_m$. Regarding the domain to represent time, we consider that time values belong to a certain domain $\mathtt{Time}$ (e.g. we may take a continuous domain such as $\mathbb{R}_+$ or a discrete domain such as $\mathbb{N}$).

Definition 1. Let $\mathtt{Time}$ be the domain to define time values, $D_1, \ldots, D_m$ be sets of values, and let us consider $\mathcal{D} = D_1 \times D_2 \times \cdots \times D_m$. A *Timed Extended Finite State Machine*, in the following TEFSM, is a tuple $M = (S, I, O, Tr, s_{in}, \bar{y})$ where S is a finite set of states, I is the set of input actions, O is the set of output actions, Tr is the set of transitions, s_{in} is the initial state, and $\bar{y} \in \mathcal{D}$ is a tuple of variables.

Each transition $t \in Tr$ is a tuple $t = (s, s', i, o, Q, Z, C)$ where $s, s' \in S$ are the initial and final states of the transition, $i \in I$ and $o \in O$ are the input and output actions, respectively, associated with the transition, $Q : \mathcal{D} \longrightarrow \mathtt{Bool}$ is a predicate on the set of variables, $Z : \mathcal{D} \longrightarrow \mathcal{D}$ is a transformation over the current variables, and $C : \mathcal{D} \longrightarrow \mathtt{Time}$ is the time that the transition needs to be completed.

A *configuration* in M is a pair $(s, \bar{x})$ where $s \in S$ is the current state and $\bar{x} \in \mathcal{D}$ is the tuple containing the current value of the variables.

We say that $tr = (s, s', (i_1/o_1, \ldots, i_r/o_r), Q, Z, C)$ is a (timed) *trace* of M if there exist transitions $t_1, \ldots, t_r \in Tr$ such that $t_1 = (s, s_1, i_1, o_1, Q_1, Z_1, C_1)$,. . ., $t_r = (s_{r-1}, s', i_r, o_r, Q_r, Z_r, C_r)$, the predicate Q is defined such that it holds $Q(\bar{x}) = (Q_1(\bar{x}) \ \wedge \ Q_2(Z_1(\bar{x})) \ \wedge \ \ldots \ \wedge \ Q_r(Z_{r-1}(\ldots(Z_1(\bar{x}))\ldots)))$, the transformation Z is defined as $Z(\bar{x}) = Z_r(Z_{r-1}(\ldots(Z_1(\bar{x}))\ldots))$, and C is defined as $C(\bar{x}) = C_1(\bar{x}) + C_2(Z_1(\bar{x})) + \cdots + C_r(Z_{r-1}(\ldots(Z_1(\bar{x}))\ldots))$.

We say that $i_1/o_1, \ldots, i_r/o_r$ is a *non-timed evolution*, or simply *evolution*, of M if there exists a trace $(s_{in}, s', (i_1/o_1, \ldots, i_r/o_r), Q, Z, C)$ of M such that $Q(\bar{y})$ holds. We denote by $\texttt{NTEvol}(M)$ the set of non-timed evolutions of M. We say that the pair $((i_1/o_1, \ldots, i_r/o_r), v)$ is a *timed evolution* of M if there exists a trace $(s_{in}, s', (i_1/o_1, \ldots, i_r/o_r), Q, Z, C)$ of M such that $Q(\bar{y})$ holds and $v = C(\bar{y})$. We denote by $\texttt{TEvol}(M)$ the set of timed evolutions of M. Let $e \in \texttt{NTEvol}(M)$ and $v \in \texttt{Time}$ be such that $(e, v) \in \texttt{TEvol}(M)$. We say that (e, v) is an *instance* of e.

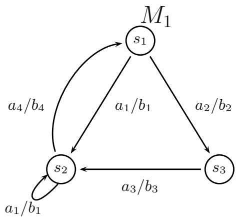
We say that M presents *non-observable non-deterministic* behavior if there exist $s, s_1, s_2 \in S$, $i \in I$, $o \in O$, $Q_1, Q_2 : \mathcal{D} \longrightarrow \texttt{Bool}$, $Z_1, Z_2 : \mathcal{D} \longrightarrow \mathcal{D}$, and $C_1, C_2 : \mathcal{D} \longrightarrow \texttt{Time}$ such that $(s, s_1, i, o, Q_1, Z_1, C_1), (s, s_2, i, o, Q_2, Z_2, C_2) \in Tr$. □

Intuitively, for a configuration $(s, \bar{x})$, a transition $t = (s, s', i, o, Q, Z, C)$ indicates that if the machine is in the state s, receives the input i, and the predicate Q holds for $\bar{x}$, then after $C(\bar{x})$ units of time the machine emits the output o and the values of the variables are transformed according to Z. Timed traces are defined as sequences of transitions. In this case, the predicate, the transformation function, and the time associated with the trace are computed from the ones corresponding to each transition belonging to the sequence. Let us note that different instances of the same evolution may appear in a specification as result of the different configurations produced after traversing the corresponding TEFSM. Finally, let us remark that the notion of non-observable non-determinism is less restrictive than the notion of observable non-determinism. For example, it allows to have both the transitions $(s, s_1, i, o_1, Q_1, Z_1, C_1)$ and $(s, s_2, i, o_2, Q_2, Z_2, C_2)$, as long as $o_1 \neq o_2$.

Example 1. In Figure 1 we present two TEFSMs. For example, let us suppose that the initial value of variables is $\bar{x} = (2, 0, 0, 2)$ and that the initial state of M_1 is s_1. Then, the transition t_{12} can be performed and it will take time $\frac{1}{2}$. After that, the value of the variables will be given by the tuple $(3, 0, 0, 1)$. □

3 (Timed) Implementation Relations

In this section we introduce our implementation relations. These relations are appropriate timed extensions of the relation $\texttt{conf}_{nt}$ [NR02]. All of them follow the same pattern: An implementation I *conforms* to a specification S if for any possible evolution of S the outputs that the implementation I may perform after a given input are a subset of those for the specification. This pattern is borrowed from ioco [Tre96, Tre99] but we do not consider *quiescent* states (that is, states where no external outputs are available). In addition to the non-timed conformance of the implementation, we require some time conditions to hold (this is a major difference with respect to ioco where time is not considered). For example, we may ask an implementation to be always faster than the time constraints imposed by the specification. The different considerations of time produce that there is not a unique way to define an implementation relation.



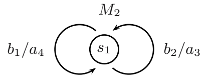
$I = \{a_1, a_2, a_3, a_4\}, O = \{b_1, b_2, b_3, b_4\}$

$$\begin{aligned}
t_{12} &= (s_1, s_2, a_1, b_1, Q_1, Z_1, C_1)\\
t_{13} &= (s_1, s_3, a_2, b_2, Q_2, Z_2, C_2)\\
t_{32} &= (s_2, s_3, a_3, b_3, Q_3, Z_3, C_3)\\
t_{21} &= (s_2, s_1, a_4, b_4, Q_4, Z_4, C_4)\\
t_{22} &= (s_2, s_2, a_1, b_1, Q_5, Z_5, C_5)
\end{aligned}$$

$$Z_i(\bar{x}) = \bar{x} + \begin{cases} (1, 0, 0, -1) \text{ if } i \in \{1, 2, 5\} \\ (0, 1, -1, 0) \text{ if } i \in \{3, 4\} \end{cases}$$

$$Q_i(\bar{x}) \equiv Z_i(\bar{x}) \geq \bar{0} \ \wedge \ \begin{cases} x_i > 0 \text{ if } i \in \{1, 2, 3, 4\} \\ x_1 > 0 \text{ if } i = 5 \end{cases}$$

$$C_i(\bar{x}) = \begin{cases} \frac{1}{x_i} & \text{if } i \in \{1, 2, 3, 4\} \ \wedge \ x_i \neq 0 \\ \frac{1}{x_1} & \text{if } i = 5 \ \wedge \ x_1 \neq 0 \end{cases}$$



$I = \{b_1, b_2\}, O = \{a_3, a_4\}$

$$\begin{aligned}
t_1 &= (s_1, s_1, b_1, a_4, Q_1, Z_1, C_1)\\
t_2 &= (s_1, s_1, b_2, a_3, Q_2, Z_2, C_2)
\end{aligned}$$

$$Z_i(\bar{x}) = \bar{x} + \begin{cases} (1, 1, -1, -1) & \text{if } i = 1 \\ (1, 0, 0, 0) & \text{if } i = 2 \end{cases}$$

$$Q_i(\bar{x}) \equiv Z_i(\bar{x}) \geq \bar{0} \ \wedge \ \begin{cases} x_1 > 0 \ \wedge \ x_2 > 0 & \text{if } i = 1 \\ x_3 > 0 & \text{if } i = 2 \end{cases}$$

$$C_i(\bar{x}) = \begin{cases} \frac{1}{x_i \cdot x_2} & \text{if } i = 1 \ \wedge \ x_1 \neq 0 \ \wedge \ x_2 \neq 0 \\ \frac{1}{x_3} & \text{if } i = 2 \ \wedge \ x_3 \neq 0 \end{cases}$$

We suppose that $\bar{x} \in \mathbb{R}^4_+$. We denote by x_i the *i-th* component of $\bar{x}$.

Fig. 1. Examples of TEFSM

Next, we formally define the sets of specifications and implementations: Spec and Imp. A specification is a timed extended finite state machine. Regarding implementations, we consider that they are also given by means of TEFSMs. In this case, we assume, as usual, that all the input actions are always enabled in any state of the implementation. Thus, we can assume that for any input i and any state of the implementation s there always exists a transition $(s, s, i, \texttt{null}, Q, Z, C)$ where null is a special (empty) output symbol, $Q(\bar{x}) \equiv \neg \bigvee\{Q'(\bar{x}) \,|\, \exists$ a transition $(s, s', i, o, Q', Z', C')\}$, $Z(\bar{x}) = \bar{x}$, and $C(\bar{x}) = 0$. Let us note that such a transition will be performed when (and only) no other transition is available (that is, either there are no transitions outgoing from s or none of the corresponding predicates hold). Other solutions consist in adding a transition leading to an *error* state or generating a transition to the initial state. In addition, we will initially consider that implementations may not present non-observable non-deterministic behavior (see Definition 1). The removal of this restriction gives raise to the definition of two additional conformance relations.

First, we recall the implementation relation $\texttt{conf}_{nt}$ where time is not considered.

Definition 2. Let S and I be two TEFSMs. We say that I *non-timely conforms* to S, denoted by $I \, \texttt{conf}_{nt} \, S$, if for each non-timed evolution $e \in \texttt{NTEvol}(S)$ with $e = (i_1/o_1, \ldots, i_{r-1}/o_{r-1}, i_r/o_r)$, $r \geq 1$, we have that

$$e' = (i_1/o_1, \ldots, i_{r-1}/o_{r-1}, i_r/o'_r) \in \texttt{NTEvol}(I) \ \text{ implies } \ e' \in \texttt{NTEvol}(S) \qquad \square$$

Next, we introduce our timed implementation relations. In the $\mathtt{conf}_a$ relation (conforms *always*) we consider that for any timed evolution of the implementation (e,t) we have that if e is a non-timed evolution of the specification then (e,t) is also a timed evolution of the specification. In the $\mathtt{conf}_w$ relation (conforms in the *worst* case) the implementation is forced, for each timed evolution fulfilling the previous conditions, to be faster than the slowest instance of the same evolution in the specification. The $\mathtt{conf}_b$ relation (conforms in the *best* case) is similar but considering the fastest instance.

Definition 3. Let S and I be two a TEFSMs. We define the following implementation relations:

– $I\,\mathtt{conf}_a\,S$ iff $I\,\mathtt{conf}_{nt}\,S$ and for all timed evolution $(e,t) \in \mathtt{TEvol}(I)$ we have $e \in \mathtt{NTEvol}(S) \Longrightarrow (e,t) \in \mathtt{TEvol}(S)$.
– $I\,\mathtt{conf}_w\,S$ iff $I\,\mathtt{conf}_{nt}\,S$ and for all timed evolution $(e,t) \in \mathtt{TEvol}(I)$ we have $e \in \mathtt{NTEvol}(S) \Longrightarrow (\exists\, t' : (e,t') \in \mathtt{TEvol}(S) \ \wedge\ t \leq t')$.
– $I\,\mathtt{conf}_b\,S$ iff $I\,\mathtt{conf}_{nt}\,S$ and for all timed evolution $(e,t) \in \mathtt{TEvol}(I)$ we have $e \in \mathtt{NTEvol}(S) \Longrightarrow (\forall\, t' : ((e,t') \in \mathtt{TEvol}(S) \Longrightarrow t \leq t'))$. □

Theorem 1. [NR02] The relations given in Definition 3 are related as follows:

$$I\mathtt{conf}_a S \Rightarrow I\mathtt{conf}_w S \Leftarrow I\mathtt{conf}_b S \qquad \square$$

It is interesting to note that if specifications are restricted to take always the same time for each given evolution (independently from the possible derivation taken for such evolution) then the relations $\mathtt{conf}_b$ and $\mathtt{conf}_w$ would coincide, but they would be still different from the $\mathtt{conf}_a$ relation.

Lemma 1. Let $M = (S, I, O, Tr, s_{in}, \bar{y})$ be a TEFSM. Let us suppose that there do not exist $((i_1/o_1, \ldots, i_r/o_r), t), ((i_1/o_1, \ldots, i_r/o_r), t') \in \mathtt{TEvol}(M)$ with $t \neq t'$. For any a TEFSM I we have $I\,\mathtt{conf}_b\,M$ iff $I\,\mathtt{conf}_w\,M$. □

4 Definition and Application of Test Cases

A test represents a sequence of inputs applied to an implementation under test. Once an output is received, we check whether it belongs to the set of expected ones or not. In the latter case, a fail signal is produced. In the former case, either a pass signal is emitted (indicating successful termination) or the testing process continues by applying another input. If we are testing an implementation with input and output sets I and O, respectively, tests are deterministic acyclic I/O labelled transition systems (i.e. trees) with a strict alternation between an input action and the set of output actions. After an output action we may find either a leaf or another input action. Leaves can be labelled either by *pass* or by *fail*. In the first case we add a *time stamp*. This time will be contrasted with the one that the implementation took to arrive to that point.

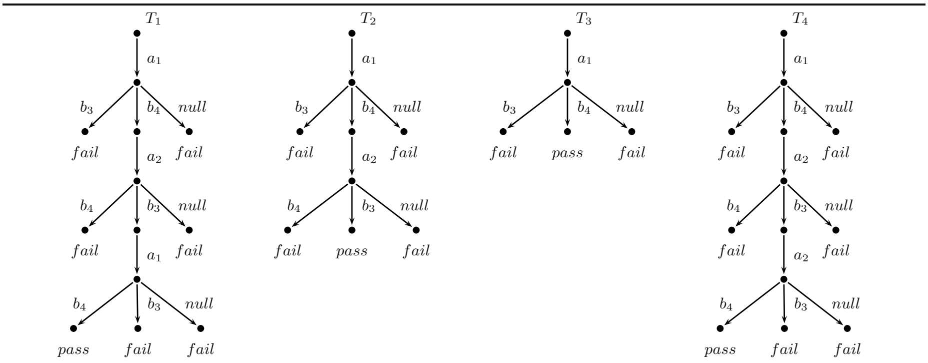


Fig. 2. Examples of Test Cases

Definition 4. A *test case* is a tuple $T = (S, I, O, Tr, s_0, S_I, S_O, S_F, S_P, C)$ where S is the set of states, I and O are disjoint sets of input and output actions, respectively, $Tr \subseteq S \times I \cup O \times S$ is the transition relation, $s_0 \in S$ is the initial state, and the sets $S_I, S_O, S_F, S_P \subseteq S$ are a partition of S. The transition relation and the sets of states fulfill the following conditions:

- S_I is the set of *input* states. We have that $s_0 \in S_I$. For all input state $s \in S_I$ there exists a unique outgoing transition $(s, a, s') \in Tr$. For this transition we have that $a \in I$ and $s' \in S_O$.
- S_O is the set of *output* states. For all output state $s \in S_O$ we have that for all $o \in O$ there exists a unique state s' such that $(s, o, s') \in Tr$. In this case, $s' \notin S_O$. Moreover, there do not exist $i \in I, s' \in S$ such that $(s, i, s') \in Tr$.
- S_F and S_P are the sets of *fail* and *pass* states, respectively. We say that these states are *terminal*. Besides, for all state $s \in S_F \cup S_P$ we have that there do not exist $a \in I \cup O$ and $s' \in S$ such that $(s, a, s') \in Tr$.

Finally, $C : S_P \longrightarrow \texttt{Time}$ is a function associating time stamps with passing states.

Let $\sigma = i_1/o_1, \ldots, i_r/o_r$. We write $T \overset{\sigma}{\Longrightarrow} s$, if $s \in S_F \cup S_P$ and there exist states $s_{12}, s_{21}, s_{22}, \ldots s_{r1}, s_{r2} \in S$ such that $\{(s_0, i_1, s_{12}), (s_{r2}, o_r, s)\} \subseteq Tr$, for all $2 \leq j \leq r$ we have $(s_{j1}, i_j, s_{j2}) \in Tr$, and for all $1 \leq j \leq r-1$ we have $(s_{j2}, o_j, s_{(j+1)1}) \in Tr$.

We say that a test case T is an *instance* of the test case T' if they only differ in the associated function C assigning times to passing states.

We say that a test case T is *valid* if the graph induced by T is a tree with root at the initial state s_0. □

In Figure 2 we present some examples of test cases (time stamps are omitted). Next we define the application of a tests suite (i.e. a set of tests) to an implementation. We say that the tests suite $\mathcal{T}$ is *passed* if for all test the terminal states reached by the composition of implementation and test are *pass* states. Besides, we give timing conditions according to the different implementation relations.

Definition 5. Let I be a TEFSM, T be a valid test, and s^T be a state of T. We write $I \parallel T \overset{\sigma}{\Longrightarrow}_t s^T$ if $T \overset{\sigma}{\Longrightarrow} s^T$ and $(\sigma, t) \in \mathtt{TEvol}(I)$.

We say that I *passes* the set of tests $\mathcal{T}$, denoted by $\mathtt{pass}(I, \mathcal{T})$, if for all test $T = (S, I, O, Tr, s, S_I, S_O, S_F, S_P, C) \in \mathcal{T}$ and $\sigma \in \mathtt{NTEvol}(I)$ there do not exist s^T and t such that $I \parallel T \overset{\sigma}{\Longrightarrow}_t s^T$ and $s^T \in S_F$.

We say that I *passes* the set of tests $\mathcal{T}$ *for any time* if $\mathtt{pass}(I, \mathcal{T})$ and for all $(\sigma, t) \in \mathtt{TEvol}(I)$ such that $T' \overset{\sigma}{\Longrightarrow} s^{T'}$, for some $T' \in \mathcal{T}$, there exists $T = (S, I, O, Tr, s, S_I, S_O, S_F, S_P, C) \in \mathcal{T}$ such that $I \parallel T \overset{\sigma}{\Longrightarrow}_t s^T$ with $s^T \in S_P$ and $t = C(s^T)$.

We say that I *passes* the set of tests $\mathcal{T}$ *in the worst time* if $\mathtt{pass}(I, \mathcal{T})$ and for all $(\sigma, t) \in \mathtt{TEvol}(I)$ such that $T' \overset{\sigma}{\Longrightarrow} s^{T'}$, for some $T' \in \mathcal{T}$, there exists $T = (S, I, O, Tr, s, S_I, S_O, S_F, S_P, C) \in \mathcal{T}$ such that $I \parallel T \overset{\sigma}{\Longrightarrow}_t s^T$ with $s^T \in S_P$ and $t \leq C(s^T)$.

We say that I *passes* the set of tests $\mathcal{T}$ *in the best time* if $\mathtt{pass}(I, \mathcal{T})$ and for all $(\sigma, t) \in \mathtt{TEvol}(I)$ such that $T' \overset{\sigma}{\Longrightarrow} s^{T'}$, for some $T' \in \mathcal{T}$, we have that for all $T = (S, I, O, Tr, s, S_I, S_O, S_F, S_P, C) \in \mathcal{T}$ such that $I \parallel T \overset{\sigma}{\Longrightarrow}_t s^T$ with $s^T \in S_P$ it holds that $t \leq C(s^T)$. □

5 Test Derivation

In this section we present an algorithm to derive test cases from specifications. As usual, the basic idea underlying our algorithm consists in traversing the specification in order to get all the possible traces in an appropriate way. First, we introduce some additional notation.

Definition 6. Let $M = (S, I, O, T, s_{in}, \bar{y})$ be a TEFSM. We consider the following sets:

$$\mathtt{out}(s, i, \bar{x}) = \{o \mid \exists\, s', Q, Z, C :\ (s, s', i, o, Q, Z, C) \in T \ \wedge\ Q(\bar{x})\}$$

$$\mathtt{after}(s, i, o, \bar{x}, t) = \left\{ (s', \bar{x}', t + t') \;\middle|\; \begin{array}{r} \exists\, Q, Z, C : (s, s', i, o, Q, Z, C) \in T \wedge \\ Q(\bar{x}) \ \wedge\ Z(\bar{x}) = \bar{x}' \ \wedge\ C(\bar{x}) = t' \end{array} \right\}$$

□

The function $\mathtt{out}(s, i, \bar{x})$ computes the set of output actions associated with those transitions that can be executed from s after receiving the input i, and assuming that the value of the variables is given by $\bar{x}$. The function $\mathtt{after}(s, i, o, \bar{x}, t)$ computes all the *situations* that can be reached from a state s after receiving the input i, producing the output o, for a value of the variables $\bar{x}$, and after passing t units of time. By *situation* we mean triples denoting the reached state, the new value of the variables, and the cumulated time since the system started its performance.

The previously defined functions can be extended in the natural way to deal with sets:

$$\mathtt{out}(S, i) = \textstyle\bigcup_{(s, \bar{x}) \in S} \mathtt{out}(s, i, \bar{x})$$

$$\mathtt{after}(D, i, o) = \textstyle\bigcup_{(s, \bar{x}, t) \in D} \mathtt{after}(s, i, o, \bar{x}, t)$$

```
Input: A specification $M = (S, I, O, Tran, s_{in}, \bar{y})$.
Output: A test case $T = (S', I, O \cup \{\texttt{null}\}, Tran', s_0, S_I, S_O, S_F, S_P, C)$.

Initialization:

  – $S' := \{s_0\}, Tran' := S_I := S_O := S_F := S_P := C := \emptyset$.
  – $S_{aux} := \{(\{(s_{in}, \bar{y}, 0)\}, s_0)\}$.

Inductive Cases: Choose one of the following two options until $S_{aux} = \emptyset$.

  1. if $(D, s^T) \in S_{aux}$ then perform the following steps:
     (a) $S_{aux} := S_{aux} - \{(D, s^T)\}$.
     (b) $S_P := S_P \cup \{s^T\};\ C(s^T) := \{t \mid (s, \overline{x}, t) \in D\}$.
  2. If $S_{aux} = \{(D, s^T)\}$ and $\exists\, i \in I : \texttt{out}(S_M, i) \neq \emptyset$, with $S_M = \{(s, \bar{x}) \mid (s, \bar{x}, t) \in D\}$,
     then perform the following steps:
     (a) $S_{aux} := \emptyset$.
     (b) Choose $i$ such that $\texttt{out}(S_M, i) \neq \emptyset$.
     (c) Consider a fresh state $s' \notin S'$ and let $S' := S' \cup \{s'\}$.
     (d) $S_I := S_I \cup \{s^T\};\ S_O := S_O \cup \{s'\};\ Tran' := Tran' \cup \{(s^T, i, s')\}$.
     (e) For all $o \notin \texttt{out}(S_M, i)$ do {null is in this case}
           – Consider a fresh state $s'' \notin S'$ and let $S' := S' \cup \{s''\}$.
           – $S_F := S_F \cup \{s''\};\ Tran' := Tran' \cup \{(s', o, s'')\}$.
     (f) For all $o \in \texttt{out}(S_M, i)$ do
           – Consider a fresh state $s'' \notin S'$ and let $S' := S' \cup \{s''\}$.
           – $Tran' := Tran' \cup \{(s', o, s'')\}$.
           – $D' := \texttt{after}(D, i, o)$.
           – $S_{aux} := S_{aux} \cup \{(D', s'')\}$.
```

Fig. 3. Derivation of test cases from a specification

The algorithm to derive tests from a specification is given in Figure 3. By considering the possible non-deterministic choices in the algorithm we may extract a full set of tests from the specification. For a given specification M, we denote this set of tests by $\texttt{tests}(M)$. Next we explain how our algorithm works. A set of *pending situations* D keeps those triples denoting the possible states, value of the variables, and time values that could appear in a state of the test whose definition, that is, its outgoing transitions, has not been yet completed. A pair $(D, s^T) \in S_{aux}$ indicates that we did not complete the state s^T of the test and that the possible situations for that state are given by the set D. Let us remark that D is a set of situations, instead of a single one, due to the non-determinism that can appear in the specification.

Example 2. Let $M = (S, I, O, Tran, s_{in}, \bar{y})$ be a specification. Suppose that we have two transitions $(s, s', i, o, Q_1, Z_1, C_1), (s, s'', i, o, Q_2, Z_2, C_2) \in Tran$. If we want to compute the evolutions of M after performing i/o we have to consider both s' and s''. Formally, for a configuration $(s, \bar{x})$ and taking into account that the time elapsed so far equals t, we have to consider the set $\texttt{after}(\{(s, \bar{x}, t)\}, i, o)$. The application of this function will return the different configurations, as well as the total elapsed time values, that could be obtained from $(s, \bar{x})$ and time t after receiving the input i and generating the output o. □

Following with the explanation of the algorithm, the set S_{aux} initially contains a tuple with the initial states (of both specification and test) and the initial situation of the process (that is, the initial state, the initial value of variables, and time 0). For each tuple belonging to S_{aux} we may choose one possibility. It is important to remark that the second step can be applied only when the set S_{aux} becomes singleton. So, our derived tests correspond to valid tests as given in Definition 4. The first possibility simply indicates that the state of the test becomes a passing state. The second possibility takes an input and generates a transition in the test labelled by this input. Then, the whole sets of outputs is considered. If the output is not expected by the implementation (step 2.(e) of the algorithm) then a transition leading to a failing state is created. This could be simulated by a single branch in the test, labelled by `else`, leading to a failing state (in the algorithm we suppose that *all* the possible outputs appear in the test). For the expected outputs (step 2.(f) of the algorithm) we create a transition with the corresponding output action and add the appropriate tuple to the set S_{aux}.

Finally, let us remark that finite test cases are constructed simply by considering a step where the second inductive case is not applied.

The next result relates, for a specification S and an implementation I, implementation relations and application of test suites. The non-timed aspects of our algorithm are based on the algorithm developed for the *ioco* relation. So, in spite of the differences, the non-timed part of the proof of our result is a simple adaptation of that in [Tre96]. Regarding temporal aspects, let us remark that the existence of different instances of the same timed evolution in the specification is the reason why only some tests (and for some time values) are forced to be passed by the implementation (e.g. sometimes we only need the *fastest/slowest* test). Specifically, we take those tests matching the requirements of the specific implementation relation. In this sense, the result holds because the temporal conditions required to conform to the specification and to pass the test suite are in fact the same.

Theorem 2. Let S, I be two `TEFSM`s. We have that:

- $I \, \texttt{conf}_a \, S$ iff I passes $\texttt{tests}(S)$ for any time.
- $I \, \texttt{conf}_w \, S$ iff I passes $\texttt{tests}(S)$ in the worst time.
- $I \, \texttt{conf}_b \, S$ iff I passes $\texttt{tests}(S)$ in the best time.

Proof. We will only prove the first of the results since the technique is similar for all of them.

First, let us show that I passes $\texttt{tests}(S)$ for any time implies $I \, \texttt{conf}_a \, S$. We will use the contrapositive, that is, we will suppose that $I \, \texttt{conf}_a \, S$ does not hold and we will prove that I does not pass $\texttt{tests}(S)$ for any time. If $I \, \texttt{conf}_a \, S$ does not hold then we have two possibilities:

- Either $I\ \texttt{conf}_{nt}\ S$ does not hold, or
- there exists a temporal evolution $(e,t) \in \texttt{TEvol}(I)$ such that $e \in \texttt{NTEvol}(S)$ and $(e,t) \notin \texttt{TEvol}(S)$.

Let us consider the first case, that is, we suppose that $I \texttt{conf}_{nt} S$ does not hold. Then, there exist two non-timed evolutions $e = (i_1/o_1, \ldots, i_{r-1}/o_{r-1}, i_r/o_r)$ and $e' = (i_1/o_1, \ldots, i_{r-1}/o_{r-1}, i_r/o'_r)$, with $r \geq 1$, such that $e \in \texttt{NTEvol}(S)$, $e' \in \texttt{NTEvol}(I)$, and $e' \notin \texttt{NTEvol}(S)$. We have to show that if $e \in \texttt{NTEvol}(S)$ then there exists a test $T = (S, I, O, Tr, s, S_I, S_O, S_F, S_P, C) \in \texttt{tests}(S)$ such that $T \overset{e}{\Longrightarrow} s^T$ and $s^T \in S_P$. We can construct this test by applying the algorithm given in Figure 3 and by resolving the non-deterministic choices in the following way:

```
for $1 \leq j \leq r$ do
        • apply the second inductive case for the input action $i_j$
        • apply the first inductive case for all the elements $(D, s^T) \in S_{aux}$
            that have been obtained by processing an output different from $o_j$
endfor
apply first inductive case for the last (i.e. remaining) element $(D, s^T) \in S_{aux}$
```

The previous algorithm generates a test T such that $T \overset{e'}{\Longrightarrow} u^T$, with $u^T \in S_F$. This is so because the last application of the second inductive case for the output o'_r must be necessarily associated to the step 2.(e) since $e' \notin \texttt{NTEvol}(S)$. Then, $I \parallel T \overset{e'}{\Longrightarrow}_t u^T$ for some time t. Given the fact that $T \in \texttt{tests}(S)$ we deduce that $\texttt{pass}(I, \texttt{tests}(S))$ does not hold. Thus, we conclude I does not pass $\texttt{tests}(S)$ for any time.

Let us suppose now that $I\ \texttt{conf}_a\ S$ does not hold because there exists a temporal evolution $(e,t) \in \texttt{TEvol}(I)$ such that $e \in \texttt{NTEvol}(S)$ and $(e,t) \notin \texttt{TEvol}(S)$. Let us consider the same test T that we defined before by taking into consideration the trace e. Since $e \in \texttt{NTEvol}(S)$ we have that $T \overset{e}{\Longrightarrow} u^T$, with $s^T \in S_P$. Besides, since $(e,t) \in \texttt{TEvol}(I)$, we also have $I \parallel T \overset{e}{\Longrightarrow}_t s^T$. The time stamps associated with the state s^T are generated by considering all the possible time values in which e could be performed in S. Thus, if $(e,t) \notin \texttt{TEvol}(S)$ then $t \notin C(s^T)$. We conclude I does not pass $\texttt{tests}(S)$ for any time.

Let us prove now that $I\ \texttt{conf}_a\ S$ implies I passes $\texttt{tests}(S)$ for any time. We will use again the contrapositive, that is, we will assume that I does not pass $\texttt{tests}(S)$ for any time and we will conclude that $I\ \texttt{conf}_a\ S$ does not hold. If I does not pass $\texttt{tests}(S)$ for any time then we have two possibilities:

- Either $\texttt{pass}(I, \texttt{tests}(S))$ does not hold, or
- there exists $(e,t) \in \texttt{TEvol}(I)$ and $T' \in \texttt{tests}(S)$ such that $T' \overset{e}{\Longrightarrow} s^{T'}$ but there does not exist $T = (S, I, O, Tr, s, S_I, S_O, S_F, S_P, C) \in \texttt{tests}(S)$ such that $I \parallel T \overset{e}{\Longrightarrow}_t s^T$, with $s^T \in S_P$ and $t \in C(s^T)$.

First, let us assume that I does not pass $\texttt{tests}(S)$ for any time because $\texttt{pass}(I, \texttt{tests}(S))$ does not hold. This means that there exists a test $T \in \texttt{tests}(S)$ such that there exist $e' = (i_1/o_1, \ldots, i_{r-1}/o_{r-1}, i_r/o'_r)$, $s^T \in S_F$, and t fulfilling

$I \parallel T \overset{e'}{\Longrightarrow}_t s^T$. Then, we have $e' \in \texttt{NTEvol}(I)$ and $T \overset{e'}{\Longrightarrow} s^T$. According to our derivation algorithm, a branch of a derived test leads to a fail state only if its associated output action is not expected in the specification. Thus, $e' \notin \texttt{NTEvol}(S)$. Let us note that our algorithm allows to create fail state only as the result of the application of the second inductive case. One of the premises of this inductive case is $\texttt{out}(S_M, i) \neq \emptyset$, that is, the specification is allowed to perform some output actions after the reception of the corresponding input. Thus, there exists an output action o_r and a trace $e = (i_1/o_1, \ldots, i_{r-1}/o_{r-1}, i_r/o_r)$ such that $e \in \texttt{NTEvol}(S)$. Given the fact that $e' \in \texttt{NTEvol}(I)$, $e' \notin \texttt{NTEvol}(S)$, and $e \in \texttt{NTEvol}(S)$, we have that $I \ \texttt{conf}_{nt} \ S$ does not hold. We conclude $I \ \texttt{conf}_a \ S$ does not hold.

Let us suppose now that I does not pass $\texttt{tests}(S)$ for any time because there exist $(e, t) \in \texttt{TEvol}(I)$ and $T' \in \texttt{tests}(S)$ such that $T' \overset{e}{\Longrightarrow} s^{T'}$ but there do not exist $T = (S, I, O, Tr, s, S_I, S_O, S_F, S_P, C) \in \texttt{tests}(S)$ such that $I \parallel T \overset{e}{\Longrightarrow}_t s^T$, with $s^T \in S_P$ and $t \in C(s^T)$. We consider three possibilities. First, if $s^{T'}$ is a fail state then the considerations given in the previous paragraph can be applied. Thus, $I \ \texttt{conf}_a \ S$ does not hold. Second, if the performance of the trace e in a test $T' \in \texttt{tests}(S)$ reaches a state $s^{T'}$ that it is neither an acceptance or a fail state, then we can always find another test $T \in \texttt{tests}(S)$ such that the performance of the sequence e reaches an acceptance state. Such a test T can be obtained by applying the first inductive case of the algorithm, instead of the second one, when dealing with the last input of the trace e. Finally, if $s^{T'}$ is an acceptance state then we simply consider $T' = T$. In the last two cases we obtain a test T such that $I \parallel T \overset{e}{\Longrightarrow}_t s^T$ and $s^T \in S_P$. Moreover, by taking into account our initial assumptions, we have $t \notin C(s^T)$. Since $s^T \in S_P$ we deduce $e \in \texttt{NTEvol}(S)$. Besides, by considering that $t \notin C(s^T)$, we deduce $(e, t) \notin \texttt{TEvol}(S)$. Finally, using that $(e, t) \in \texttt{TEvol}(I)$, we conclude $I \ \texttt{conf}_a \ S$ does not hold. □

As a straightforward corollary we have that the dependencies between conformance relations that we presented in Theorem 1 also hold for the corresponding testing relations. For instance, since $\texttt{conf}_a \Rightarrow \texttt{conf}_w$ we also deduce that "passes X at any time" implies "passes X in the worst time."

6 Removing Restrictions on Implementations

If we allow implementations to present non-observable non-deterministic behavior then we may naturally introduce two more relations. The $\texttt{conf}_{sw}$ relation requests that, for each of its evolutions, at least one instance of the implementation must be faster than the slowest instance, for the same evolution, of the specification. The $\texttt{conf}_{sb}$ requests that, for each of its evolutions, at least one instance of the implementation is faster than the fastest instance of the specification.

Definition 7. Let S and I be two TEFSMs. We write $I \ \texttt{conf}_{sw} \ S$ if $I \ \texttt{conf}_{nt} \ S$ and for all evolution $(i_1/o_1, \ldots, i_r/o_r) \in \texttt{NTEvol}(I) \cap \texttt{NTEvol}(S)$ we have

$$\exists\, t_1, t_2 : \begin{pmatrix} ((i_1/o_1, \ldots, i_r/o_r), t_1) \in \texttt{TEvol}(I) \land \\ ((i_1/o_1, \ldots, i_r/o_r), t_2) \in \texttt{TEvol}(S) \land \\ t_1 \leq t_2 \end{pmatrix}$$

We write $I\ \mathtt{conf}_{sb}\ S$ if $I\ \mathtt{conf}_{nt}\ S$ and for all evolution $(i_1/o_1, \ldots, i_r/o_r) \in \mathtt{NTEvol}(I) \cap \mathtt{NTEvol}(S)$ we have

$$\exists\, t_1 : \begin{pmatrix} ((i_1/o_1, \ldots, i_r/o_r), t_1) \in \mathtt{TEvol}(I) \wedge \\ \forall\, ((i_1/o_1, \ldots, i_r/o_r), t_2) \in \mathtt{TEvol}(S) : t_1 \leq t_2 \end{pmatrix}$$

□

Next we generalize Lemma 1 to deal with the case where either the implementation or the specification (or both) are deterministic.

Lemma 2. Let I, S be two TEFSMs. We have the following results:

(1) If for all non-temporal evolution $(i_1/o_1, \ldots, i_r/o_r) \in \mathtt{NTEvol}(S)$ there do not exist two different time values t, t' such that $((i_1/o_1, \ldots, i_r/o_r), t)$ and $((i_1/o_1, \ldots, i_r/o_r), t') \in \mathtt{TEvol}(S)$, then $I\,\mathtt{conf}_w S$ iff $I\,\mathtt{conf}_b S$ and $I\,\mathtt{conf}_{sw} S$ iff $I\ \mathtt{conf}_{sb}\ S$.

(2) If for all non-temporal evolution $(i_1/o_1, \ldots, i_r/o_r) \in \mathtt{NTEvol}(I)$ there do not exist two different time values t, t' such that $((i_1/o_1, \ldots, i_r/o_r), t)$ and $((i_1/o_1, \ldots, i_r/o_r), t') \in \mathtt{TEvol}(I)$, then $I\,\mathtt{conf}_w S$ iff $I\,\mathtt{conf}_{sw} S$ and $I\,\mathtt{conf}_b S$ iff $I\ \mathtt{conf}_{sb}\ S$.

(3) If the conditions of the results (1) and (2) hold then the relations $\mathtt{conf}_w$, $\mathtt{conf}_b$, $\mathtt{conf}_{sw}$, and $\mathtt{conf}_{sb}$ coincide.

Proof. If the condition in (1) holds then the best instance of each evolution in the specification is actually the worst instance of that evolution. Similarly, if the condition in (2) holds then the best instance of each evolution in the implementation is the worst one as well. The last result is obtained from results (1) and (2) by applying transitivity between relations. □

Let us note that the relation $\mathtt{conf}_a$ is different from the other relations even when the temporal behavior of both the specification and the implementation is deterministic. This is so because $\mathtt{conf}_a$ requires that time values in the implementation coincide with those in the specification, while other relations require that time values in the implementation are *less than or equal to* those of the specification.

Taking into account these new relations, we can extend Theorem 1 to include our five timed relations.

Theorem 3. The relations given in Definitions 3 and 7 are related as follows:

$$\begin{tikzcd}
 & I\ \mathtt{conf}_b\ S \arrow[r, Rightarrow] \arrow[d, Rightarrow] & I\ \mathtt{conf}_{sb}\ S \arrow[d, Rightarrow] \\
I\ \mathtt{conf}_a\ S \arrow[r, Rightarrow] & I\ \mathtt{conf}_w\ S \arrow[r, Rightarrow] & I\ \mathtt{conf}_{sw}\ S
\end{tikzcd}$$

Proof Sketch: The relation between $\mathtt{conf}_a$, $\mathtt{conf}_b S$, and $\mathtt{conf}_w$ was established in Theorem 1 and the proof can be found in [NR02]. Thus, we only need to consider $\mathtt{conf}_{sb}$ and $\mathtt{conf}_{sw}$. If $I\ \mathtt{conf}_w\ S$ then we know that each instance of a temporal evolution of I needs a time less than or equal to the one corresponding to the slowest instance, for the same evolution, of the specification S. In particular, there exists an instance fulfilling the condition imposed by $\texttt{conf}_{sw}$. So, we conclude $I \, \texttt{conf}_{sw} \, S$. The same reasoning can be also used to prove that $I \, \texttt{conf}_{b} \, S$ implies $I \, \texttt{conf}_{sb} \, S$.

Finally, we have to study the relation between $\texttt{conf}_{sb}$ and $\texttt{conf}_{sw}$. If $I \texttt{conf}_{sb} S$ then we have that for any evolution of I there exists an instance being faster than the fastest instance of the same evolution in S. In particular, this instance is also faster than the slowest instance of S, so that we conclude $I \, \texttt{conf}_{sw} \, S$. □

7 Conclusions and Future Work

We have presented a methodology for testing both functional and temporal aspects of systems where temporal behavior is critical. This requires us to endow tests with temporal requirements. Five implementation relations, differing in their temporal requirements, have been introduced and related. The nondeterminism of either the implementation or the specification induces some peculiarities in the testing methodology. In particular, when a test suite is applied to an implementation, the correctness of the temporal behavior is not assessed by checking the correctness of each test separately, but by checking temporal constraints over all the tests together. A sound and complete test derivation algorithm is constructed.

As future work we plan to improve the capability of our framework to express temporal constraints. In particular, we want to express conditions over the *minimal* time consumed by an action. Symbolic temporal constraints would allow to express both minimal and maximal bounds in a compact fashion. In addition, we plan to endow specifications with the capability to express temporal constraints over both actions and traces. In our current framework, temporal requirements are applied to traces. So, we assume that a low performance of an action may be compensated by other previous actions where performance was high. However, individual actions may have specific temporal requirements in some particular domains.

Acknowledgments. We would like to thank the anonymous referees for their helpful remarks.

References

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[BB04] L. Brandán Briones and E. Brinksma. A test generation framework for quiescent real-time systems. In *4th Int. Workshop on Formal Approaches to Testing of Software (FATES 2004), LNCS 3395*, pages 64-78. Springer, 2004.

[BSS86] E. Brinksma, G. Scollo, and C. Steenbergen. LOTOS specifications, their implementations and their tests. In *Protocol Specification, Testing and Verification VI*, pages 349–360. North Holland, 1986.

[CL97] D. Clarke and I. Lee. Automatic generation of tests for timing constraints from requirements. In *3rd Workshop on Object-Oriented Real-Time Dependable Systems*, 1997.

[ED03] A. En-Nouaary and R. Dssouli. A guided method for testing timed input output automata. In *TestCom 2003, LNCS 2644*, pages 211–225. Springer, 2003.

[EDK02] A. En-Nouaary, R. Dssouli, and F. Khendek. Timed Wp-method: Testing real time systems. *IEEE Transactions on Software Engineering*, 28(11):1024–1039, 2002.

[HNTC99] T. Higashino, A. Nakata, K. Taniguchi, and A. Cavalli. Generating test cases for a timed I/O automaton model. In *12th Workshop on Testing of Communicating Systems*, pages 197–214. Kluwer Academic Publishers, 1999.

[LMN04] K.G. Larsen, M. Mikucionis, and B. Nielsen. Online testing of real-time systems using uppaal. In *4th Int. Workshop on Formal Approaches to Testing of Software (FATES 2004), LNCS 3395*, pages 79–94. Springer, 2004.

[MMM95] D. Mandrioli, S. Morasca, and A. Morzenti. Generating test cases for real time systems from logic specifications. *ACM Transactions on Computer Systems*, 13(4):356–398, 1995.

[NR02] M. Núñez and I. Rodríguez. Encoding PAMR into (timed) EFSMs. In *FORTE 2002, LNCS 2529*, pages 1–16. Springer, 2002.

[SVD01] J. Springintveld, F. Vaandrager, and P.R. D'Argenio. Testing timed automata. *Theoretical Computer Science*, 254(1-2):225–257, 2001.

[Tre96] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software – Concepts and Tools*, 17(3):103–120, 1996.

[Tre99] J. Tretmans. Testing concurrent systems: A formal approach. In *CONCUR'99, LNCS 1664*, pages 46–65. Springer, 1999.

Conformance Tests as Checking Experiments for Partial Nondeterministic FSM

Alexandre Petrenko [1] and Nina Yevtushenko [2]

[1] CRIM, Centre de recherche informatique de Montréal, 550 Sherbrooke Street West,
Suite 100, Montreal, H3A 1B9, Canada
petrenko@crim.ca
[2] Tomsk State University, 36 Lenin Street, Tomsk, 634050, Russia
yevtushenko.RFF@elefot.tsu.ru

Abstract. The paper addresses the problem of conformance test generation from input/output FSMs that might be partially specified and nondeterministic. Two conformance relations are considered, quasi-reduction and quasi-equivalence. The former requires that in response to each input sequence defined in a specification FSM, a conforming implementation FSM produces only output sequences of the specification FSM, while the latter is stronger: a conforming implementation FSM must produce all of them and nothing else. For each relation, a test generation method is elaborated. The resulting tests are proven to be complete, i.e., sound and exhaustive, for a given bound on the number of states; they include as special cases checking experiments for deterministic FSMs.

1 Introduction

Testing software systems often requires taking into account potential nondeterministic behavior, when for given sequence of inputs different sequences of outputs can be produced. Specifying a system, the developer may use nondeterminism to describe, for example, implementation options and allowable interleaving of outputs. Testing implementations obtained from a nondeterministic specification, one has to choose a conformance relation that either requires the implementation under test to be as nondeterministic as its specification or allows it to be less nondeterministic. Test generation should take into account a chosen conformance relation, so the resulting tests will be different. In this paper, we consider the problem of test generation from a Finite State Machine (FSM) with inputs and outputs, which can be nondeterministic, moreover, not necessarily completely specified. The latter means that for some combinations of states and inputs further behavior of a machine is not defined, any behavior of an implementation is considered allowable. The FSM model considered in this paper differs from other state-oriented models, such as input/output automata or input/output transition systems that are also used to express nondeterminism, labels of its transitions are pairs of input and output actions and not single actions.

The problem of testing from nondeterministic FSMs has been studied since the end of eighties of the last century; see the list of references. We can categorize them into the following groups:

W. Grieskamp and C. Weise (Eds.): FATES 2005, LNCS 3997, pp. 118 – 133, 2006.

1. Traversal problems [9, 15, 22].
2. State distinguishability [1, 2, 3, 10, 21, 22].
3. Constructing sound or complete tests [4-7, 11-18, 22].
4. Test selection for execution [8, 9, 22].

This paper belongs to the third category, where the goal is to develop methods for determining a sound or complete test that offers a fault coverage guarantee: an implementation FSM passes a test if and only if it conforms to its specification FSM (under some additional assumption about the number of its states). Assuming that an implementation FSM is complete and nondeterministic; conformance relations to test are quasi-equivalence and quasi-reduction, which are similar to trace equivalence or trace inclusion (the difference occurs when the specification FSM is not completely specified). Such tests include as a special case checking experiments originally defined for deterministic machines.

In particular, we propose methods for generating complete tests (within a given bound on the number of states) from an FSM that might be partially specified and nondeterministic at the same time. No assumption on state distinguishability is made, in other words, the proposed methods equally apply to FSMs that may not be minimized.

This paper is organized as follows. Section 2 presents the notions used to study nondeterministic FSMs. In Section 3, we define an FSM test as a nondeterministic (tree) FSM that is an unfolding of a specification FSM and define what we mean by a complete test. Sections 4 and 5 present methods for generating complete tests w.r.t. quasi-equivalence and quasi-reduction relations, respectively. We compare our results with previous work in Section 6 and conclude in Section 7.

2 General Definitions

Definition 1. A *Finite State Machine* (FSM) A is a 5-tuple (S, s_0, I, O, h), where

- S is a finite set of states with the initial state s_0;
- I and O are finite non-empty sets of inputs and outputs, respectively, which satisfy the condition $I \cap O = \varnothing$;
- h is a behavior function h: $S \times I \to 2^{S \times O}$, where $2^{S \times O}$ is the powerset of $S \times O$.

Definition 2. FSM $A = (S, s_0, I, O, h)$ is

- *completely specified* (a complete FSM) if $h(s, a) \neq \varnothing$ for all $(s, a) \in S \times I$;
- *partially specified* (a partial FSM) if $h(s, a) = \varnothing$ for some $(s, a) \in S \times I$;
- *deterministic* if $|h(s, a)| \leq 1$ for all $(s, a) \in S \times I$;
- *nondeterministic* if $h(s, a) > 1$ for some $(s, a) \in S \times I$;
- *observable* if the automaton $A_\times = (S, s_0, I \times O, \delta)$, where $\delta(s, ao) \ni s'$ iff $(s', o) \in h(s, a)$, is deterministic.

In this paper, we consider only observable machines; one could use a standard procedure for automata determinization to convert a given FSM into observable one.

A word α of the automaton $A_\times$ in state s is a *trace* of A in state s; let $Tr(s)$ denote the set of all traces of A in state s, while $Tr(A)$ denote the set of traces of A in the initial state. Given sequence $\alpha \in (IO)^*$, the *input projection* of α, denoted $\alpha_{\downarrow I}$, is a sequence obtained from α by erasing symbols in O. Input sequence $\beta \in I^*$ is a *defined* input sequence in state s of A if there exists $\alpha \in Tr(s)$ such that $\beta = \alpha_{\downarrow I}$. We use $\Omega(s)$ to denote the set of all defined input sequences for state s and $\Omega(A)$ for the state s_0, i.e., for A. $\Omega(A) = Tr(A)_{\downarrow I} = I^*$ holds for any complete machine A, while for a partial FSM $\Omega(A) \subseteq I^*$. We define in terms of traces several relations which characterize how similar the behavior of different states might be.

Definition 3. Given FSM $A = (S, s_0, I, O, h)$ and $s, t \in S$,

- s and t are (*trace-*) *equivalent*, $s \cong t$, if $Tr(s) = Tr(t)$;
- t is *quasi-equivalent* to s, $t \sqsupseteq s$, if $\Omega(t) \supseteq \Omega(s)$ and $\{\beta \in Tr(s) \mid \beta_{\downarrow I} = \alpha\} = \{\beta \in Tr(t) \mid \beta_{\downarrow I} = \alpha\}$ for all $\alpha \in \Omega(s)$;
- t is *trace-included* into (is a *reduction* of) s, $t \leq s$, if $Tr(t) \subseteq Tr(s)$;
- t is *a quasi-reduction* of s, $t \lesssim s$, if $\Omega(t) \supseteq \Omega(s)$ and $\{\beta \in Tr(t) \mid \beta_{\downarrow I} = \alpha\} \subseteq \{\beta \in Tr(s) \mid \beta_{\downarrow I} = \alpha\}$ for all $\alpha \in \Omega(s)$;
- s and t are *inseparable*, $s \sim t$, if $\{\beta \in Tr(s) \mid \beta_{\downarrow I} = \alpha\} \cap \{\beta \in Tr(t) \mid \beta_{\downarrow I} = \alpha\} \neq \varnothing$ for all $\alpha \in \Omega(s) \cap \Omega(t)$.

We state some relationships between the relations defined above.

Proposition 1. If FSM is

- partial and nondeterministic, then $\cong \subseteq \leq$ and $\cong \subseteq \sqsupseteq \subseteq \lesssim$;
- complete, then $\cong = \sqsupseteq$ and $\lesssim = \leq$;
- deterministic, then $\cong = \sim$ and $\sqsupseteq = \lesssim$;
- complete and deterministic, then $\cong = \lesssim = \sqsupseteq = \leq = \sim$.

For testing we also need various relations which characterize how dissimilar the behavior of different states might be.

Definition 4. Given FSM $A = (S, s_0, I, O, h)$ and $s, t \in S$,

- t is *distinguishable* from s w.r.t. the reduction (and quasi-reduction) relation, written $t \not\lesssim s$, if $\{\beta \in Tr(t) \mid \beta_{\downarrow I} = \alpha\} \not\subseteq \{\beta \in Tr(s) \mid \beta_{\downarrow I} = \alpha\}$ for some $\alpha \in \Omega(t) \cap \Omega(s)$; we use the notation $t \not\lesssim_\alpha s$ when we need to refer to the input sequence α that detects that t is not a reduction of s;
- s and t are *distinguishable*, $s \ncong t$, if there exists $\alpha \in \Omega(s) \cap \Omega(t)$ such that $\{\beta \in Tr(s) \mid \beta_{\downarrow I} = \alpha\} \neq \{\beta \in Tr(t) \mid \beta_{\downarrow I} = \alpha\}$, called an input sequence *distinguishing* s and t; then the set of traces $\{\beta \in Tr(s) \mid \beta_{\downarrow I} = \alpha\}$ is the *distinguishing* set of s w.r.t. t, denoted $d(s, t)$, and $\{\beta \in Tr(t) \mid \beta_{\downarrow I} = \alpha\}$ is the *distinguishing* set of t w.r.t. s, denoted $d(t, s)$; we also use the notation $s \ncong_\alpha t$ when we need to refer to the input sequence distinguishing s and t;
- s and t are *separable*, $s \nsim t$, if there exists $\alpha \in \Omega(s) \cap \Omega(t)$ such that $\{\beta \in Tr(s) \mid \beta_{\downarrow I} = \alpha\} \cap \{\beta \in Tr(t) \mid \beta_{\downarrow I} = \alpha\} = \varnothing$, called a *separating* sequence, this fact is also denoted $s \nsim_\alpha t$.

We briefly present the intuition captured by these notions. If for two states, s and t, of a complete machine, t can produce an output sequence in response to some input sequence that s cannot, then t cannot be a (quasi-) reduction of s. However, state s may still be a quasi-reduction of t. Similarly, distinguishable states are not equivalent, but one may be a quasi-reduction of another. Finally, separable states are not equivalent, one state cannot be a quasi-reduction of another, moreover, no other state can be a (quasi-) reduction of both states. However, the converse is not necessarily true. States that are not separable may still have no common reduction. This is reflected in the notion of r-distinguishability [1, 18, 21, 23]. To define it, we first introduce several notations. Given $(s, a) \in S \times I$, let $out(s, a)$ denote the set of outputs produced by A in state s for input a, that is $\{o \mid \exists s' \in S \text{ s.t. } (s', o) \in h(s, a)\}$. For a trace $\alpha \in Tr(s)$, s-after-α denotes the state reached by A when it executes the trace α from state s. If s is the initial state s_0 then instead of s_0-after-α we write A-after-α.

Definition 5. Given FSM $A = (S, s_0, I, O, h)$, states $s, t \in S$,

- s and t are $r(1)$*-distinguishable* if there exists input a such that $s \nsim_a t$;
- given $k > 1$, s and t are $r(k)$*-distinguishable*, if the states are $r(k - 1)$-distinguishable or there exists an input $a \in I$ such that for each trace $\gamma \in a(out(s, a) \cap out(t, a))$ states s-after-γ and t-after-γ are $r(k - 1)$-distinguishable;
- s and t are *r-distinguishable*, denoted $s \not\simeq t$, if there exists $k > 0$ such that states s and t are $r(k)$-distinguishable.

If $s \nsim_a t$ for some input a then the set of traces $\{ab \mid b \in out(s, a)\}$ is the $r(1)$-*distinguishing* set of traces of s *w.r.t.* t, denoted $\rho(s, t)$; while $\{ac \mid c \in out(t, a)\}$ is the $r(1)$-*distinguishing* set of traces of t w.r.t. s, denoted $\rho(t, s)$. If the states s and t are $r(k)$-distinguishable, but not $r(k - 1)$-distinguishable and there exists an input $a \in I$ such that for each trace $\gamma \in \{ab \mid b \in out(s, a) \cap out(t, a)\}$ states s-after-γ and t-after-γ are $r(k - 1)$-distinguishable, while $\rho(s\text{-after-}\gamma, t\text{-after-}\gamma)$ and $\rho(t\text{-after-}\gamma, s\text{-after-}\gamma)$ are their $r(k - 1)$-distinguishing sets, respectively, then the set of traces $\rho(s, t) = \{\gamma\kappa \mid \gamma \in \{ab \mid b \in (out(s, a) \cap out(t, a)\} \wedge \kappa \in \rho(s\text{-after-}\gamma, t\text{-after-}\gamma)\}$ is the *r-distinguishing* set of s *w.r.t.* t and $\rho(t, s) = \{\gamma\kappa \mid \gamma \in \{ab \mid b \in out(s, a) \cap out(t, a)\} \wedge \kappa \in \rho(t\text{-after-}\gamma, s\text{-after-}\gamma)\}$ is that of t w.r.t. s.

The above relations could also be applied to states from different machines. Considering the disjoint union of the machines, A and B, we have $A\ r\ B$, iff $s_0\ r\ t_0$, where $r \in \{\sqsupseteq, \cong, \lesssim, \leq, \nleq, \ncong, \nsim, \not\simeq\}$.

We state some relationships between the relations defined above.

Proposition 2. If FSM is

- deterministic, then $\ncong\ =\ \nsim\ =\ \not\simeq\ =\ \nleq$;
- nondeterministic, then $\nsim\ \subseteq\ \not\simeq\ \subseteq\ \nleq\ \subseteq\ \ncong$.

The following proposition adds more on their relationships, in particular, it states necessary and sufficient conditions when given two states s_1 and s_2 of an observable FSM A, there exists an FSM B and state t of B such that t is quasi-equivalent to (a quasi-reduction of) both states s_1 and s_2.

Proposition 3. Let s_1 and s_2 be two states of an observable FSM A.

1. There exists an FSM B and state t of B such that t is quasi-equivalent to both states s_1 and s_2 if and only if states s_1 and s_2 are not distinguishable.
2. If states $s_1 \not\cong_\alpha s_2$ then for every complete FSM B and for every state t of B it holds that $\{\beta \in Tr(t) \mid \beta_{\downarrow I} = \alpha\} \neq \{\beta \in Tr(s_1) \mid \beta_{\downarrow I} = \alpha\}$ or $\{\beta \in Tr(t) \mid \beta_{\downarrow I} = \alpha\} \neq \{\beta \in Tr(s_2) \mid \beta_{\downarrow I} = \alpha\}$, i.e., $t \not\cong_\alpha s_1$ or $t \not\cong_\alpha s_2$.
3. There exists an FSM B and state t of B such that t is a quasi-reduction of both states s_1 and s_2 if and only if states s_1 and s_2 are not r-distinguishable.
4. If states s_1 and s_2 are r-distinguishable and $\rho(s_1, s_2)$ and $\rho(s_2, s_1)$ are corresponding r-distinguishing sets then for every FSM B and for every state t of B it holds that $\{\beta \in Tr(t) \mid \beta_{\downarrow I} \in \rho(s_1, s_2)_{\downarrow I}\} \not\subseteq \rho(s_1, s_2)$ or $\{\beta \in Tr(t) \mid \beta_{\downarrow I} \in \rho(s_2, s_1)_{\downarrow I}\} \not\subseteq \rho(s_2, s_1)$; i.e., state t can be distinguished from state s_1 with the set $\rho(s_1, s_2)$ or from state s_2 with the set $\rho(s_2, s_1)$.

There exists a simple means to characterize the maximal common behavior of two machines.

Definition 6. Let $A = (S, s_0, I, O, h)$ and $B = (T, t_0, I, O, g)$ be FSMs. The *intersection* of A and B is an FSM $A \cap B = (Q\ q_0, I, O, \varphi)$ with the state set $Q \subseteq S \times T$, the initial state $q_0 = (s_0t_0)$, and the behavior function φ: $Q \times I \to 2^{Q \times O}$, such that Q is the smallest state set obtained by using the following rule $(s't', o) \in \varphi(st, a)$ iff $(s', o) \in h(s, a)$ and $(t', o) \in g(t, a)$.

The FSM intersection preserves only common traces of the component machines, in other words, we have $Tr(A \cap B) = Tr(A) \cap Tr(B)$.

We use the FSM intersection to formulate methods for test generation.

Proposition 4. Given observable FSMs A and B, let $A \cap B$ be the intersection of FSMs A and B. There exist $\beta \in Tr(A \cap B)$ and $a \in I$, such that $out(B\text{-after-}\beta, a) \neq out(A\ \text{-after-}\beta, a)$ if and only if $B \not\cong A$.

Given state st of the intersection $A \cap B$, we say that there exists a *distinguishing transition* (w.r.t. the quasi-equivalence relation) from state st, if there exists $a \in I$, such that $out(s, a) \neq out(t, a)$.

Proposition 5. Given observable FSMs A and B, there exist $\beta \in Tr(A \cap B)$ and $a \in I$, such that $out((A \cap B)\text{-after-}\beta, a) \neq out(B\text{-after-}\beta, a)$ if and only if $B \not\leq A$.

Given state st of the intersection $A \cap B$, we say that there exists a *distinguishing transition* (w.r.t. the quasi-reduction relation) from state st if there exists $a \in I$, such that $out(st, a) \neq out(t, a)$.

3 FSM Tests

In this paper, we assume that a specification FSM from which we generate tests is an observable machine but not necessarily complete and deterministic, while any implementation FSM is complete and observable.

Definition 7. Given FSMs $A = (S, s_0, I, O, h)$ and $U = (T, t_0, I, O, \lambda)$, U is a *test* for the FSM A (in the state s_0) if the following conditions are satisfied:

- $U \leq A$,
- $Tr(U)$ is finite (i.e., U has no cycles),
- U-after-$\alpha = U$-after-β implies $\alpha = \beta$ for all $\alpha, \beta \in Tr(U)$ (i.e., U is a tree).

If U is a test for FSM A such that $A \sqsupseteq U$ then we sometimes call a test U an *unfolding* of A. Any test of A can be obtained by transforming the graph of A into a tree, while skipping some inputs and outputs of A. The tree structure of a test is fully determined by the set of its traces, so we can always use $Tr(U)$ to refer to the test U. Note that inputs (outputs) of the specification machine are also inputs (outputs) of a test, so a tester, executing such a test, applies inputs of the test to an implementation under test and observe its outputs.

A test may have transitions with different outputs from a same state and under the same input. A test U is *input-homogeneous* if for each two traces $\alpha, \beta \in Tr(U)$, such that $\alpha_{\downarrow I} = \beta_{\downarrow I}$, the test U has a trace $\alpha\gamma$ if and only if it has a trace $\beta\kappa$; $\kappa_{\downarrow I} = \gamma_{\downarrow I}$. For input-homogeneous tests, an output produced by an implementation under test does not decide which next input to apply. However, in a test that is not input-homogeneous, an observed output may decide which input to choose for execution. Such adaptiveness may become necessary when a specification FSM is partial to avoid using in tests its unspecified inputs. These inputs can even be forbidden to apply, so they are not used in the definitions of conformance relations considered in this paper.

A test may have transitions with different inputs from a same state. We assume, therefore, that a reliable reset operation is available in any implementation, so a tester executing such a test selects one among alternative inputs during a particular test run. The tester produces the verdict *fail* whenever the implementation FSM executes an input/output sequence that is not a trace of the test. To produce the verdict *pass*, the tester has to repeatedly execute every input in each state of the test until there is a sufficient confidence that the implementation FSM exhibited all nondeterministic options, according to some fairness, called also complete testing assumption [12]. The verdict *pass* expresses a chosen notion of conformance of an implementation FSM to a specification FSM.

To characterize conformance in this paper, we use quasi-equivalence and quasi-reduction relations. The quasi-reduction conformance relation requires that in response to each input sequence defined in the specification FSM a conforming implementation FSM produces only output sequences of the specification FSM. The quasi-equivalence conformance relation is stronger than the quasi-reduction: a conforming implementation FSM must produce all the output sequences of the specification FSM for each defined input sequence and only them.

Let $\Im(A)$ be a set of complete observable (implementation) machines over the input alphabet of A, called a *fault domain*. FSM $B \in \Im(A)$ is a *conforming* implementation machine of A w.r.t. the quasi-equivalence (quasi-reduction) relation if $B \sqsupseteq A$ ($B \lesssim A$).

Definition 8. Given a test U for the specification FSM A and an implementation FSM $B \in \Im(A)$,

- *B passes* the test U for quasi-equivalence (quasi-reduction) relation, if $B \sqsupseteq U$ ($B \lesssim U$). The test U is *sound* for FSM A in the fault domain $\Im(A)$ w.r.t. quasi-equivalence (quasi-reduction) relation, if any $B \in \Im(A)$, which is a conforming implementation machine of A w.r.t. quasi-equivalence (quasi-reduction) relation, passes the test U.
- *B fails* U for quasi-equivalence (quasi-reduction) relation if $B \not\sqsupseteq U$ ($B \not\lesssim U$). The test U is *exhaustive* for FSM A in the fault domain $\Im(A)$ w.r.t. quasi-equivalence (quasi-reduction) relation, if any $B \in \Im(A)$, which is not a conforming implementation FSM of A, fails the test U.
- The test U is *complete* for FSM A in $\Im(A)$ w.r.t. quasi-equivalence (quasi-reduction) relation, if it is sound and exhaustive in the fault domain $\Im(A)$ w.r.t. quasi-equivalence (quasi-reduction) relation.

The set $\Im(A)$ that contains all complete observable FSM with at most m states is denoted $\Im_m(A)$. A test is *m-complete* if it is complete in the fault domain $\Im_m(A)$. Clearly, an m-complete test is also k-complete for any $k < m$.

There exist partial FSMs for which a complete test is easy to determine. Those are FSMs that have no cycling behavior, as stated in the following.

Proposition 6. Given a specification FSM A such that the set of traces $Tr(A)$ is finite and an arbitrary fault domain $\Im(A)$, the unfolding U of A such that $Tr(U) = Tr(A)$ is a test complete in $\Im(A)$ w.r.t. quasi-equivalence and quasi-reduction relations.

Thus, as opposed to complete FSMs (whose behavior is cyclic), some partial FSMs have a complete test regardless of a bound m on the number of states in implementation machines and conformance relation. Systematic methods are required to generate m-complete tests for a general type of nondeterministic FSMs and quasi-equivalence as well as quasi-reduction relations. Tests complete for one relation are not necessary complete for the other, so methods should be specialized for each relation.

In the following sections, we elaborate two methods for determining m-complete tests from partially specified nondeterministic FSMs w.r.t. both conformance relations, quasi-equivalence and quasi-reduction.

4 Testing for Quasi-equivalence

4.1 Preliminaries

The following property of quasi-equivalence relation is essential for constructing tests m-complete w.r.t. this relation.

Lemma 1. If $s' \sqsupseteq s$ in A and $\beta \in Tr(s)$ then $\beta \in Tr(s')$ and s'-after-$\beta \sqsupseteq s$-after-β.

Given a specification FSM $A = (S, s_0, I, O, h)$ and an implementation FSM $B = (T, t_0, I, O, g)$, the FSM intersection $A \cap B$ contains the common behavior of the two machines. As stated in Proposition 4, for a non-conforming implementation machine $B \in \Im(A)$, $B \not\sqsupseteq A$, the FSM B fails a test U if the test U has a trace β followed by input a such that $\beta \in Tr(A \cap B)$ and $out(B\text{-after-}\beta, a) \neq out(A\text{-after-}\beta, a)$. To generate a test complete in the fault domain $\mathfrak{I}_m(A)$ it is sufficient to find a test for A that has a trace with the above property for every $B \in \mathfrak{I}_m(A)$, $B \not\cong A$.

Proposition 7. Given a specification FSM $A = (S, s_0, I, O, h)$ with n states and an implementation FSM $B = (T, t_0, I, O, g)$ with m states, let $A \cap B = (Q, q_0, I, O, \psi)$ be the intersection of A and B. Then

1. $|Q| \leq nm$;
2. let (s', t) and (s, t) be states of $A \cap B$ reached via traces β and γ such that $s \not\cong s'$, then FSM B fails a test U if $Tr(U) \supseteq \beta d(s, s') \cup \gamma d(s', s)$.
3. let (s', t) and (s, t) be states of $A \cap B$ reached via traces β and γ, respectively, such that $s' \sqsupseteq s$, if FSM B fails a test U, $Tr(U) = \gamma V$, then FSM B fails a test W such that $Tr(W) \supseteq \beta V$.

Proposition 7.1 implies the tight upper bound, nm, on tests for partial nondeterministic FSMs earlier established for deterministic machines [19, 20]. Proposition 7.2 is a corollary to Proposition 3.2, while Proposition 7.3 is implied by Lemma 1.

4.2 Test Generation Method for Quasi-equivalence

We formulate the sufficient conditions for the completeness of a test w.r.t. quasi-equivalence relation.

Let $Pref(\beta)$ denote the set of all non-empty prefixes of trace $\beta \in (IO)^*$. Given a specification FSM A, states $s, p \in S$ and trace $\beta \in Tr(s)$, let $Pref_{s,p}(\beta)$ be the set $\{\omega \mid \omega \in Pref(\beta) \wedge s\text{-after-}\omega \sqsupseteq p\}$. We define a relation $\sqsupseteq_{s,p}$ on the set $Pref_{s,p}(\beta)$, such that $\omega \sqsupseteq_{s,p} \omega'$ for $\omega, \omega' \in Pref_{s,p}(\beta)$, if $|\omega| \leq |\omega'|$ and $s\text{-after-}\omega \sqsupseteq s\text{-after-}\omega'$.

Proposition 8. Given an FSM A, states $s, p \in S$ and trace $\beta \in Tr(s)$, the relation $\sqsupseteq_{s,p}$ is a partial order on the set $Pref_{s,p}(\beta)$.

We denote $l(Pref_{s,p}(\beta), \sqsupseteq_{s,p})$ the length of the poset $(Pref_{s,p}(\beta), \sqsupseteq_{s,p})$, that is the cardinality of a longest chain of the poset. We select one of the longest chains of the poset $(Pref_{s,p}(\beta), \sqsupseteq_{s,p})$ and denote it $C_{s,p}(\beta)$. By definition, we assume that $l(Pref_{s,p}(\beta), \sqsupseteq_{s,p}) = 0$ if the set $Pref_{s,p}(\beta)$ is empty.

Proposition 7.3 indicates that to obtain an m-complete test it is sufficient to unfold FSM A only from certain states of A, in particular, given $s' \sqsupseteq s$, unfold it only from state s' and not from s. This leads us to the concept of a core of FSM. Given FSM A, we determine a minimal set of states of A, called a *core* of FSM A, that contains the initial state and, for each state $s \in S$, a state quasi-equivalent to s (a set is minimal w.r.t. the inclusion ordering). If the machine A has no quasi-equivalent states, the core coincides with the state set S. A minimal set K of traces of A is a *core cover* of A if for each state s in the core of A, K has a trace that takes A from the initial state to state s. Given a subset R of states of A, we denote K_R a minimal subset of K such that $\forall s \in R$ $\exists \alpha \in K_R$ $(A\text{-after-}\alpha \sqsupseteq s)$.

Let $R \subseteq S$ be such that $\forall s_1, s_2 \in R$ ($s_1 \neq s_2$ implies $s_1 \not\cong s_2$), such a set is called a *set of pairwise distinguishable states* of A, we use $\mathcal{R}_A$ to denote the set of all such sets of states of A. Assume first that there exists $R \in \mathcal{R}_A$ such that $m < |R|$. We consider a core cover K of FSM A and determine a minimal subset K_R of K, where for each $s \in R$ there exists $\sigma \in K_R$ such that A-after-$\sigma \sqsupseteq s$. Proposition 7.2 implies that any test that includes the set of traces $\cup_{\alpha \in K_R}\{\alpha d(A\text{-after-}\alpha, s) \mid s \in R\}$ is an m-complete test for FSM A w.r.t. the quasi-equivalence relation. Thus, we further assume that for each $R \in \mathcal{R}_A$ it holds that $m \geq |R|$.

Let K be a core cover of FSM A and G be an unfolding of A, such that $Tr(G) = \cup_{\alpha_i \in K}\alpha_i N_i$, where $N_i = \{\beta \in Tr(A\text{-after-}\alpha_i) \mid \Sigma_{p \in R} l(\textit{Pref}_{A\text{-after-}\alpha_i,p}(\nu), \sqsupseteq_{A\text{-after-}\alpha_i,p}) + |R| \leq m$ for each proper prefix ν of β and for $\forall R \in \mathcal{R}_A$ and $(Tr(A\text{-after-}\alpha_i\beta) = \varnothing$ or $\exists R \in \mathcal{R}_A$ s.t. $\Sigma_{p \in R} l(\textit{Pref}_{A\text{-after-}\alpha_i,p}(\beta), \sqsupseteq_{A\text{-after-}\alpha_i,p}) + |R| = m + 1)\}$. In words, the unfolding N_i of A contains each shortest trace β of state A-after-α_i such that a state with no defined inputs is reached or the total lengths of posets induced by pairwise distinguishable states reaches the value of m - $|R|$ + 1.

Proposition 9. Given $B = (T, t_0, I, O, g)$, $B \in \mathfrak{I}_m(A)$, $B \not\cong A$, that passes the test G, there exist $\alpha_i \in K$, $\beta \in N_i$, such that for any set $R \in \mathcal{R}_A$ if $\Sigma_{p \in R} l(\textit{Pref}_{A\text{-after-}\alpha_i,p}(\beta), \sqsupseteq_{A\text{-after-}\alpha_i,p}) + |R| = m + 1$ then there exist (s, t), (s', t), $s \not\cong s'$, in the set $\{(A \cap B)\text{-after-}\gamma \mid \gamma \in (K_i(\beta) \cup \cup_{p \in R}\alpha_i C_{A\text{-after-}\alpha_i,p}(\beta))\}$, where $K_i(\beta) \subseteq K$ and for each $s \in (R \cup \{A\text{-after-}\omega \mid \omega \in \cup_{p \in R}\alpha_i C_{A\text{-after-}\alpha_i,p}(\beta)\})$ there exists $\sigma \in K_i(\beta)$ such that A-after-$\sigma \sqsupseteq s$.

Combining now the results of Proposition 9 and Proposition 7.2, we have the following method.

The SC-method for deriving an m-complete test w.r.t. the quasi-equivalence relation.
Input. FSM $A = (S, s_0, I, O, h)$ and an integer m.
Output. An m-complete test U for FSM A w.r.t. the quasi-equivalence relation.

Step 1. Determine a core cover K of A and the set of traces $U = \{\beta \in Tr(A) \mid \beta{\downarrow_I} = \alpha_i{\downarrow_I} \wedge \alpha_i \in K\}$.

Step 2. For each $\alpha_i \in K$, determine the set of traces N_i that comprises each shortest trace $\beta \in Tr(A\text{-after-}\alpha_i)$ such that $\Sigma_{p \in R} l(\textit{Pref}_{A\text{-after-}\alpha_i,p}(\beta), \sqsupseteq_{A\text{-after-}\alpha_i,p}) + |R| = m + 1$ for some set $R \in \mathcal{R}_A$ or $(\Sigma_{p \in R} l(\textit{Pref}_{A\text{-after-}\alpha_i,p}(\beta), \sqsupseteq_{A\text{-after-}\alpha_i,p}) + |R| \bullet m$ for all $R \in \mathcal{R}_A$ and $Tr(A\text{-after-}\alpha_i\beta) = \varnothing)$.

Step 3. For each set N_i and each trace $\beta \in N_i$, select a set $R(\beta) \in \mathcal{R}_A$ such that $\Sigma_{p \in R(\beta)} l(\textit{Pref}_{A\text{-after-}\alpha_i,p}(\beta), \sqsupseteq_{A\text{-after-}\alpha_i,p}) + |R(\beta)| = m + 1$, if it exists, and for each pair (p, β), $p \in R(\beta)$, determine a longest chain $C_{A\text{-after-}\alpha_i,p}(\beta)$ in the poset $(\textit{Pref}_{A\text{-after-}\alpha_i,p}(\beta), \sqsupseteq_{A\text{-after-}\alpha_i,p})$.

Step 4. For each $R(\beta)$ determine a minimal subset $K_i(\beta) \subseteq K$ such that for each $s \in (R(\beta) \cup \{A\text{-after-}\alpha_i\omega \mid \omega \in \cup_{p \in R(\beta)} C_{A\text{-after-}\alpha_i,p}(\beta)\})$ there exists $\alpha \in K_i(\beta)$ such that A-after-$\alpha \sqsupseteq s$.

Step 5. For each pair of distinguishable states s and s' determine their distinguishing traces $d(s, s')$, $d(s', s)$.

Step 6. For each $K_i(\beta)$ determine the sets of traces $\{\mu d(A\text{-after-}\mu, A\text{-after-}\omega) \mid \mu, \omega \in K_i(\beta)\}$ and $\{\mu d(A\text{-after-}\mu, A\text{-after-}\alpha_i\omega) \mid \mu \in K_i(\beta), \omega \in \cup_{p \in R(\beta)} C_{\delta(s_0, \alpha_i), p}(\beta)\}$, and include all the obtained traces into the set U.

Step 7. For each $\alpha_i \in K$, and each trace $\beta \in N_i$, determine the sets of traces $\{\alpha_i\omega d(A\text{-after-}\alpha_i\omega, A\text{-after-}\mu) \mid \omega \in \cup_{p \in R(\beta)} C_{A\text{-after-}\alpha_i, p}(\beta), \mu \in K_i(\beta)\}$ and $\{\alpha_i\mu d(A\text{-after-}\alpha_i\mu, A\text{-after-}\alpha_i\omega) \mid \mu, \omega \in \cup_{p \in R(\beta)} C_{A\text{-after-}\alpha_i, p}(\beta)\}$, and include all the obtained traces into the set U.

Theorem 1. A test with the set U of traces obtained by the above method is m-complete w.r.t. the quasi-equivalence relation.

5 Testing for Quasi-reduction

5.1 Preliminaries

The following property of the quasi-reduction relation is essential for constructing tests complete w.r.t. this relation.

Lemma 2. If $s' \lesssim s$ in A and $\beta \in Tr(s)$ then $\beta\downarrow_I \in \Omega(s')$, while $\beta \in Tr(s')$ such that $\beta\downarrow_I \in \Omega(s)$ implies that s'-after-$\beta \lesssim s$-after-β.

In case of the quasi-reduction relation, if $B = (T, t_0, I, O, g)$ is a non-conforming implementation machine of A, $B \not\lesssim A$, then it fails a test U if and only if the test U has a trace β followed by input a such that $\beta \in Tr(A \cap B)$, $out(B\text{-after-}\beta, a) \neq out((A \cap B)\text{-after-}\beta, a)$. To generate a test complete in the fault domain $\mathfrak{I}_m(A)$ w.r.t. the quasi-reduction relation it is sufficient to find a test for A that has a trace with the above property for every $B \in \mathfrak{I}_m(A)$, $B \not\lesssim A$. Similar to the quasi-equivalence relation, m-complete tests w.r.t. the quasi-reduction relation have a tight upper nm.

When deriving an m-complete test w.r.t. the quasi-equivalence relation in order to shorten the test we use distinguishable states of the specification FSM. The reason is no state of any implementation FSM is quasi-equivalent to two distinguishable states (Proposition 3.1). However, two distinguishable states can have a common quasi-reduction, in other words, they could be represented, i.e., “implemented” in a conforming implementation machine as a single state that is a quasi-reduction of each of these two states of the specification FSM. Due to Proposition 3.3, another, namely, r-distinguishability, relation between states of the specification FSM should be used when deriving an m-complete test w.r.t. the quasi-reduction relation.

Proposition 10. Given a specification FSM $A = (S, s_0, I, O, h)$ and an implementation FSM $B = (T, t_0, I, O, g)$, let $A \cap B = (Q, q_0, I, O, \psi)$ be the intersection of A and B. Then

1. let (s', t) and (s, t) be states of $A \cap B$ reached via traces β and γ such that $s \not\simeq s'$, then FSM B fails a test U if $Tr(U) \supseteq \beta\rho(s, s') \cup \gamma\rho(s', s)$.

2. let (s', t) and (s, t) be states of $A \cap B$ reached via traces β and γ, respectively, such that $s \lesssim s'$, if FSM B fails a test U, $Tr(U) = \gamma V$, then FSM B fails a test W such that $Tr(W) \supseteq \beta V$.

Proposition 10.1 is a corollary to Proposition 3.4, while Proposition 10.2 is implied by Lemma 2.

Differently from the quasi-equivalence relation, given a conforming implementation B of the specification FSM A, $B \lesssim A$, certain states of the specification FSM may not be present in the intersection $A \cap B$. The reason is FSM B is allowed to produce fewer output sequences than A. As a corollary, all states of the specification FSM need not be implemented to obtain a conforming implementation. However, these are states of the specification FSM that have to be implemented in every conforming implementation. Those are deterministically reachable states.

State s of the specification FSM A is *deterministically reachable* (*d-reachable* state) if there exists an input sequence α such that all traces with the input projection α take the FSM A from the initial state to the state s. The input sequence α is called a *d-transfer* sequence for deterministically reachable state s. Each deterministically reachable state has to be implemented in every conforming implementation. The specification FSM has at least one deterministically reachable state, namely the initial state that is reachable via the empty sequence.

Proposition 11. Given a specification FSM A and an implementation FSM B that is a quasi-reduction of A, let s be a state of the specification FSM. If s is a deterministically reachable state then there exists a state t of the implementation FSM B such that the intersection $A \cap B$ has state st. Moreover, if state s is not deterministically reachable then there exists a quasi-reduction B of A such that the intersection $A \cap B$ has no state st for any state t of the FSM B.

5.2 Test Generation Method for Quasi-reduction

Given a specification FSM A, states $s, p \in S$, and trace $\beta \in Tr(s)$, let $Pref_{s,p}(\beta)$ be the set $\{\omega \mid \omega \in Pref(\beta) \wedge s\text{-after-}\omega \lesssim p\}$. We define a relation $\lesssim_{s,p}$ on the set $Pref_{s,p}(\beta)$, such that $\omega \lesssim_{s,p} \omega'$ for $\omega, \omega' \in Pref_{s,p}(\beta)$, if $|\omega| \leq |\omega'|$ and s-after-$\omega \lesssim s$-after-ω'.

Proposition 12. Given a specification FSM A, states $s, p \in S$ and trace $\beta \in Tr(s)$, the relation $\lesssim_{s,p}$ is a partial order on the set $Pref_{s,p}(\beta)$.

Let $\lesssim_{s,p}$ be a partial order relation on the set $Pref_{s,p}(\beta)$. We select one of the longest chains of the poset $(Pref_{s,p}(\beta), \lesssim_{s,p})$ and denote it $C_{s,p}(\beta)$.

Proposition 11 indicates that, deriving a test, it might be unnecessary to unfold FSM A from states that are not deterministically reachable, since these states may be not implemented in an implementation at hand. Moreover, Proposition 10.2 indicates that it is sufficient to unfold FSM A only from certain states of A, in particular, given $s \lesssim s'$, unfold it only from state s and not from s'. This leads us to the concept of a d-core of FSM. Given FSM A, we determine a minimal set of states of A, called a *d-core* of FSM A, that contains the initial state and, for each deterministically reachable state $s \in S$, a state that is deterministically reachable and is a quasi-reduction of s. A minimal set K of traces of A is a *d-core cover* of A if for each state s in the d-core of A, K has the empty sequence and a trace that takes A from the initial state to the state s. Given a subset R of states of A, we denote R_d a subset of states of R that are deterministically reachable in the FSM A. Given a subset R_d of d-reachable states of A, we denote K_R a minimal subset of K such that $\forall s \in R_d\ \exists \alpha \in K_R$ (A-after-$\alpha \lesssim s$).

Let $R \subseteq S$ be such that $\forall s_1, s_2 \in R$ ($s_1 \neq s_2$ implies $s_1 \not\simeq s_2$), such a set is called a *set of pairwise r-distinguishable states* of A, we use $\mathcal{R}_A$ to denote the set of all such sets of states of A.

Assume that there exists a set of d-reachable states $R_d \in \mathcal{R}_A$ such that $m < |R_d|$. We consider a d-core cover K of FSM A and determine a minimal subset K_{R_d} of K where for each $p \in R_d$ there exists $\sigma \in K_R$ such that A-after-$\sigma \lesssim p$. According to Proposition 10.1, any test U that includes the set of traces $\cup_{\alpha \in K_{R_d}}\{\alpha\rho(A\text{-after-}\alpha, s) \mid s \in R_d\}$ is an m-complete test for FSM A w.r.t. the quasi-reduction relation. Thus, we further assume that for each $R \in \mathcal{R}_A$ it holds that $m \geq |R_d|$.

Let K be a d-core cover of FSM A and G be an unfolding of A, such that $Tr(G) = \cup_{\alpha_i \in K}\alpha_i N_i$, where $N_i = \{\beta \in Tr(A\text{-after-}\alpha_i) \mid \Sigma_{p \in R} l(Pref_{A\text{-after-}\alpha_i,p}(\nu), \lesssim_{A\text{-after-}\alpha_i,p}) + |R_d| \leq m$ for each proper prefix ν of β and for $\forall R \in \mathcal{R}_A$ and $(Tr(A\text{-after-}\alpha_i\beta) = \varnothing$ or $\exists R \in \mathcal{R}_A$ s.t. $\Sigma_{p \in R} l(Pref_{A\text{-after-}\alpha_i,p}(\beta), \lesssim_{A\text{-after-}\alpha_i,p}) + |R_d| = m + 1)\}$.

Proposition 13. Given $B = (T, t_0, I, O, g)$, $B \in \mathfrak{I}_m(A)$, $B \not\lesssim A$, that passes the test G, there exist $\alpha_i \in K$, $\beta \in N_i$, such that for any set $R \in \mathcal{R}_A$ if $\Sigma_{p \in R} l(Pref_{A\text{-after-}\alpha_i,p}(\beta), \lesssim_{A\text{-after-}\alpha_i,p}) + |R_d| = m + 1$ then there exist (s, t) and (s', t), $s \not\simeq s'$, in the set $\{(A \cap B)\text{-after-}\gamma \mid \gamma \in (K_i(\beta) \cup \cup_{p \in R}\alpha_i C_{A\text{-after-}\alpha_i,p}(\beta))\}$, where $K_i(\beta) \subseteq K$ and for each $s \in (R \cup \{A\text{-after-}\omega \mid \omega \in \cup_{p \in R}\alpha_i C_{A\text{-after-}\alpha_i,p}(\beta)\})$ that has d-reachable quasi-reduction there exists $\sigma \in K_i(\beta)$ such that A-after-$\sigma \lesssim s$.

Combining now the results of Proposition 13 and Proposition 10.1, we have the following method.

The SCR-method for deriving an m-complete test w.r.t. the quasi-reduction relation.
Input. FSM $A = (S, s_0, X, Y, h)$ and an integer m.
Output. An m-complete test U for FSM A w.r.t. the quasi-reduction relation.

Step 1. Determine a d-core cover K of A and the set of traces $U = \{\beta \in Tr(A) \mid \beta\downarrow_I = \alpha_i\downarrow_I \wedge \alpha_i \in K\}$.

Step 2. For each $\alpha_i \in K$, determine the set of traces N_i that comprises each shortest trace $\beta \in Tr(A\text{-after-}\alpha_i)$ such that $\Sigma_{p \in R} l(Pref_{A\text{-after-}\alpha_i,p}(\beta), \lesssim_{A\text{-after-}\alpha_i,p}) + |R_d| = m + 1$ for some set $R \in \mathcal{R}_A$ or $(\Sigma_{p \in R} l(Pref_{A\text{-after-}\alpha_i,p}(\beta), \lesssim_{A\text{-after-}\alpha_i,p}) + |R_d| \bullet m$ for all $R \in \mathcal{R}_A$ and $Tr(A\text{-after-}\alpha_i\beta) = \varnothing)$.

Step 3. For each set N_i and each trace $\beta \in N_i$, select a set $R(\beta) \in \mathcal{R}_A$ such that $\Sigma_{p\in R(\beta)} l(Pref_{A\text{-after-}\alpha_i,p}(\beta), \lesssim_{A\text{-after-}\alpha_i,p}) + |R(\beta)_d| = m + 1$, if it exists, and for each pair (p, β), $p \in R(\beta)$, determine a longest chain $C_{A\text{-after-}\alpha_i,p}(\beta)$ in the poset $(Pref_{A\text{-after-}\alpha_i,p}(\beta), \lesssim_{A\text{-after-}\alpha_i,p})$.

Step 4. For each $R(\beta)$ determine a minimal subset $K_i(\beta) \subseteq K$ such that for each $s \in (R(\beta) \cup \{A\text{-after-}\alpha_i\omega \mid \omega \in \cup_{p\in R(\beta)} C_{A\text{-after-}\alpha_i,p}(\beta)\})$ that has a d-reachable quasi-reduction, there exists $\alpha \in K_i(\beta)$ such that $A\text{-after-}\alpha \lesssim s$.

Step 5. For each pair of r-distinguishable states s and s' determine their r-distinguishing traces $\rho(s, s')$, $\rho(s', s)$.

Step 6. For each $K_i(\beta)$ determine the sets of traces $\{\mu\rho(A\text{-after-}\mu, A\text{-after-}\omega) \mid \mu, \omega \in K_i(\beta)\}$ and $\{\mu\rho(A\text{-after-}\mu, A\text{-after-}\alpha_i\omega) \mid \mu \in K_i(\beta), \omega \in \cup_{p\in R(\beta)} C_{\delta(s_0,\alpha_i),p}(\beta)\}$, and include all the obtained traces into the set U.

Step 7. For each $\alpha_i \in K$, and each trace $\beta \in N_i$, determine the sets of traces $\{\alpha_i\omega\rho(A\text{-after-}\alpha_i\omega, A\text{-after-}\mu) \mid \omega \in \cup_{p\in R(\beta)} C_{A\text{-after-}\alpha_i,p}(\beta), \mu \in K_i(\beta)\}$ and $\{\alpha_i\mu\rho(A\text{-after-}\alpha_i\mu, A\text{-after-}\alpha_i\omega) \mid \mu, \omega \in \cup_{p\in R(\beta)} C_{A\text{-after-}\alpha_i,p}(\beta)\}$, and include all the obtained traces into the set U.

Theorem 2. A test with the set U of traces obtained by the above method is m-complete w.r.t. the quasi-reduction relation.

6 Related Work

We elaborated a common approach for generating complete tests for quasi-equivalence and quasi-reduction relations and presented two methods that implement this approach. Several non-trivial distinctions between these methods are due to the fact that a conforming implementation w.r.t. quasi-reduction is not required to execute each trace of a complete test, as is the case for quasi-equivalence. As a result, not every state of a specification FSM may be mapped onto a state of a conforming implementation FSM w.r.t. quasi-reduction. Recall that classical checking experiments for minimal deterministic complete FSMs test the machine isomorphism or homomorphism.

Applied to (complete or partial) deterministic FSMs, both methods reduce to the version of the SC-method presented in [20]. In case of complete nondeterministic FSMs, our methods improve earlier works [11, 12, 17, 18, 23]. Compared to those results, the proposed methods deliver shorter tests, as they derive traversal sets from fewer states, use the rule for terminating traversal sequences (traces) that refines those previously used, and more sparingly use separating (distinguishing) sequences. In case of minimal complete nondeterministic FSMs and trace equivalence conformance relation, complete tests can be determined using adapted versions of methods developed for complete deterministic FSMs, as shown, e.g., in [12]. Though, our approach does not even require FSMs to be minimal. When a specification FSM is partial, trace equivalence is refined to quasi-equivalence, while trace inclusion (reduction) to quasi-reduction. Complete tests for quasi-equivalence are considered in

[13], where the termination rule for traversal traces is much coarser than the one used in the proposed method. Methods for finding complete tests for quasi-reduction are reported in [18, 23]. Again, our method never produces longer tests due to features mentioned above, namely, compared to all the previous methods, the proposed methods build traversal traces from fewer states, use the rules that prune traversal traces earlier, and more sparingly use separating (distinguishing) sequences, in particular, they are not used after each prefix of traversal traces, as in the previous methods.

All implementation FSMs conforming to the specification FSM w.r.t. quasi-equivalence share the set of traces of the specification FSM. This implies that a test that decides the conformance is the same no matter which particular FSM (for a given fault domain) is submitted for testing, as our tests avoid undefined inputs. However, this is no longer the case for quasi-reduction. Conforming implementation FSMs w.r.t. quasi-reduction can contain different traces for the same set of defined input sequences of the specification FSM, i.e., be pairwise distinguishable. This indicates that while each of the traces of a complete test is eventually executed when all implementation FSMs with at most *m* states are tested, an *m*-complete test may contain traces that are never executed when only a single implementation is tested. In this scenario, we may avoid using *m*-complete tests, by finding a test for an unknown FSM with at most *m* states. Soundness and exhaustiveness in this case mean that if the implementation FSM is a quasi-reduction of its specification FSM then the test should produce the pass verdict, otherwise the fail verdict. Clearly, such test may be not sound and/or exhaustive for other FSMs in the fault domain. The testing scenario with a single implementation FSM requires choosing inputs depending on outputs produced by the implementation FSM in response to previous input, so test generation and execution have to be merged into a testing on-the-fly procedure. This scenario, addressed in [4-6], is different from the one considered in this paper, as we assume that same complete test is used no matter which implementation FSM is tested. The procedures for test generation from a complete specification FSM, called adaptive in [4-6], adopt earlier versions of the SC-method for the reduction relation. Thus improvements of this method extend to adaptive methods as well.

Yet another testing scenario occurs when any implementation FSM derived from a given nondeterministic FSM is assumed to be deterministic. In this case, some traces of the specification cannot be executed by any deterministic FSM with a known number of states, so a complete test constructed for a fault domain with nondeterministic FSMs may be redundant. Some results for this scenario are reported in [5, 6, 11, 14, 18]. Our SCR-method also applies to this scenario, but does not assume that implementation FSMs are deterministic.

7 Conclusion

We addressed the problem of conformance testing by generating complete tests from a most general type of input/output FSMs, namely partially specified nondeterministic FSMs. Completeness is imbedded in the very notion of checking experiments, originally defined for deterministic FSMs and recently extended to the nondeterministic case. We proposed an approach for deriving tests complete w.r.t. quasi-equivalence and quasi-reduction conformance relations from a general type of FSM. The proposed methods surpass the existing methods either in the efficiency (shorter tests) or in the applicability (fewer assumptions about specification FSMs). We discussed the complexity of tests produced by the proposed methods. The tight upper bound on the length of each trace in tests is the product of the numbers of states in specification and implementation machines, for nondeterministic and deterministic cases, quasi-equivalence and quasi-reduction relations. In the worst case, the number of tests grows exponentially with the number of states, similar to tests for deterministic machines. This implies that the proposed methods scale as the existing methods for constructing checking experiments for deterministic FSMs. To us, the completeness of tests comes always with the price that includes the scalability of a test generation method. Nondeterminism and partiality of a specification cannot improve the scalability. All things being equal, the scalability of complete test generation methods improves with a smaller fault domain. In this paper, we assumed a coarse fault model defined just by a maximal number of states in implementation machines. As a result, the fault domain is the universe of all FSMs with given parameters.

Further research should address test generation with finer fault models (thus, with smaller fault domains), generalizing the corresponding fault model-based test generation methods for deterministic machines. Our current work includes extending the proposed approach to other testing scenarios.

References

1. H. AboElFotoh, O. Abou-Rabia, and H. Ural. A Test Generation Algorithm for Protocols Modeled as Non-Deterministic FSMs. The Software Eng. Journal, 1993, 8(4), pp.184-188.
2. R. Alur, C. Courcoubetis, and M. Yannakakis. Distinguishing Tests for Nondeterministic and Probabilistic Machines. 27th ACM Symp. on Theory of Comp., 1995, pp. 363-372.
3. S. Yu. Boroday. Distinguishing Tests for Non-Deterministic Finite State Machines. Testing of Communicating Systems, IFIP TC6 11th International Workshop on Testing of Communicating Systems, 1998, Kluwer, pp. 101-107.
4. M. Dorofeeva, A. Petrenko, M. Vetrova, and N. Yevtushenko. Adaptive Test Generation from a nondeterministic FSM, Radioelektronika i informatika. 2004, No. 3, pp. 91-95.
5. R. M. Hierons. Adaptive Testing of a Deterministic Implementation against a Nondeterministic Finite State Machine. The Computer Journal, 1998, 41(5), pp. 349-355.
6. R. M. Hierons, Testing from a Non-Deterministic Finite State Machine Using Adaptive State Counting, IEEE Transactions on Computers, 2004, 53, 10, pp. 1330-1342.
7. R. M. Hierons. Using Candidates to Test a Deterministic Implementation Against a Non-deterministic Finite State Machine, The Computer Journal, 2003, 46, 3, pp. 307-318.
8. R. M. Hierons and H. Ural. Concerning the Ordering of Adaptive Test Sequences, 23rd IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2003), 2003, LNCS volume 2767, pp. 289-302.
9. I. Hwang, T. Kim, S. Hong, J. Lee, Test Selection for a Nondeterministic FSM, Computer Communications, 2001, Vol. 24/12, 7, pp.1213-1223.
10. H. Kloosterman, Test Derivation from Non-Deterministic Finite State Machines. Protocol Test Systems, V, Proceedings of the IFIP TC6/WG6.1 Fifth International Workshop on Protocol Test Systems, Canada, 1992. North-Holland 1993, pp. 297-308.

11. I. Kufareva, N. Yevtushenko, and A. Petrenko, Design of Tests for Nondeterministic Machines with Respect to Reduction, Automatic Control and Computer Sciences, Allerton Press Inc., USA, No. 3, 1998.
12. G. L. Luo, G. v. Bochmann, and A. Petrenko. Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized Wp-method. IEEE Transactions on Software Engineering, 1994, 20(2), pp. 149–161.
13. G. Luo, A. Petrenko, G. v. Bochmann. Selecting Test Sequences for Partially Specified Nondeterministic Finite State Machines, the IFIP Seventh International Workshop on Protocol Test Systems, Japan, 1994, pp. 95-118.
14. R. Miller, D. Chen, D. Lee, R. Hao. Coping with Nondeterminism in Network Protocol Testing. TestCom 2005.
15. L. Nachmanson, M. Veanes, W. Schulte, N Tillmann, W. Grieskamp, Optimal Strategies for Testing Nondeterministic Systems, ISSTA 2004, Software Engineering Notes ACM, 29, pp. 55–64.
16. A. Petrenko. Checking Experiments with Protocol Machines, Proceedings of the IFIP Fourth International Workshop on Protocol Test Systems, the Netherlands, 1991, pp. 83-94.
17. A. Petrenko, N. Yevtushenko, A. Lebedev, and A. Das. Nondeterministic State Machines in Protocol Conformance Testing, Proceedings of the IFIP Sixth International Workshop on Protocol Test Systems, France, 1993, pp. 363-378.
18. A. Petrenko, N. Yevtushenko, and G. v. Bochmann. Testing Deterministic Implementations from their Nondeterministic Specifications, Proceedings of the IFIP Ninth International Workshop on Testing of Communicating Systems, 1996, pp. 125-140.
19. A. Petrenko and N. Yevtushenko, On Test Derivation from Partial Specifications, Proceedings of the IFIP Joint International Conference, FORTE/PSTV'2000, on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification, Italy, 2000, pp. 85-102.
20. A. Petrenko and N. Yevtushenko. Testing from Partial Deterministic FSM Specifications, IEEE Transactions on Computers, 2005, Vol. 54, No. 9, pp.1154-1165.
21. P. Tripathy and K. Naik, Generation of Adaptive Test Cases from Nondeterministic Finite State Models. Protocol Test Systems, V, Proceedings of the IFIP TC6/WG6.1 Fifth International Workshop on Protocol Test Systems, North-Holland 1993, pp. 309-320.
22. F. Zhang and T. Cheung, Optimal Transfer Trees and Distinguishing Trees for Testing Observable Nondeterministic Finite-State Machines, IEEE Transactions on Software Engineering, 2003, Vol. 29, No. 1, pp. 1-14.
23. N. Yevtushenko, A. Lebedev, and A. Petrenko. On Checking Experiments with Nondeterministic Automata. Automatic Control and Computer Sciences, 1991, 6, pp. 81–85.

Calculating Probabilities of Real-Time Test Cases*

Marcin Jurdziński, Doron Peled, and Hongyang Qu

Department of Computer Science,
University of Warwick, Coventry CV4 7AL, UK

Abstract. When testing a system, it is often necessary to execute a suspicious trace in a realistic environment. Due to nondeterministic choices existing in concurrent systems, such a particular trace may not be scheduled for execution. Thus it is useful to compute the probability of executing the trace. Our probabilistic model of real-time systems requires that for each transition, the period from the time when its enabling condition becomes satisfied to the time when it is fired is bounded and the length of the period obeys a probabilistic distribution. This model is not Markovian if the distribution is not exponential. Therefore it cannot be analyzed by Markov processes. We propose to use integration to calculate the probability for a path. Then we discuss the possibility to optimize the calculation.

1 Introduction

A common result in testing systems is a scenario that is suspicious of being faulty. Such a scenario, often called an 'error trace', is of great value for the system developers. One can follow the description of the scenario and try to figure out the location or even the cause for the problem. While the list of events that constitute the scenario can be easily traced on paper, demonstrating that the problem really occurred during the execution of the system may be difficult due to race conditions. Adding code that controls the execution to capture the particular scenario, as proposed in [16], can be used in discrete systems. But it typically changes the time constraints in real-time systems and therefore the checked systems. A naive solution can be to try executing the system several times given the same initial conditions. However, it is not a priori given how many times one needs to repeat such an experiment, nor even if it is realistic to assume that enough repetitions can help. Thus one needs to compute the probability of executing the suspicious trace in order to decide the times we are expected to run the system until the scenario occurs.

Another important issue is the frequency of the occurrence of problems found in the code. In some cases, it is more practical and economical to develop recoverable code than foolproof one. Given a discovered failure, the decision of whether

* This research was partially supported by Subcontract UTA03-031 to The University of Warwick under University of Texas at Austin's prime National Science Foundation Grant #CCR-0205483.

W. Grieskamp and C. Weise (Eds.): FATES 2005, LNCS 3997, pp. 134–151, 2006.

to correct it or let a recovery algorithm try to catch it depends on the probability of the failure to occur. In such cases, an analysis that estimates the probability of occurrence of a given scenario under given initial conditions is necessary. Consider a typical situation that a bug is found in a communication protocol. This bug causes a sent packet to be damaged. If the chance of the bug being triggered is very small, e.g., one packet out of 100,000 can be damaged, one would allow retransmitting the damaged packet. On the contrary, if its probability is one out of 100, one would redesign the protocol. Overall, the probability calculation of a particular path is of significance in theory and practice. We intend to provide the methods and tools for such calculations.

In our execution model, we assume that we can control the initial state of the system to start the execution of the scenario. However, due to concurrency and real-time constraints, other probabilistic choices may occur. We assume that each transition that the system can make must be delayed for a random period, which is bounded by a lower and upper time limit and obeys a continuous probability distribution, such as uniform distribution or normal distribution. Such an assumption has gained more and more attention, e.g., [2, 4, 11]. For simplicity, we consider uniform distribution in this paper, but the result can be applied to other distributions, as shown in the conclusion. Due to probabilistic choices in the system description, the probability of executing the system from a given initial state and making particular choices, depends not only on the given scenario; it also depends on other probabilistic choices available (but not taken) by the analysed system. Thus, the probabilistic analysis of a path involves considering larger parts of the code than only the transitions participating in the given scenario. Moreover, the particular ordered occurrence of concurrent transitions in a given path is probabilistic itself. Instead of using a path as a linear order between occurrences of transitions, it is sometimes more natural to look at a partial order that can be extracted from a path, and represents dependencies between transitions, according to the processes in which they participate.

In this paper, such a system is modeled by a *transition system*. In the literature, many probabilistic systems have been proposed. Probabilistic timed automata (PTA) have been studied by [12, 19]. PTA in both of these papers have discrete probabilistic choices. Thus probability can be calculated using Markov chains. However, in our model, the execution of a transition in a path might depend on part of, or even the entire execution history before this transition. Therefore, the probabilistic model we define is not Markovian if not all of distributions are exponential. The work in [6] considered continuous-time Markov chains (CTMC), which can only be generated if all distributions in the model are exponential distributions.

In order to allow general continuous distributions, a semi-Markov chain model, which is an extension of CTMC, was studied in [14]. The models defined in [2, 5, 11, 20, 21] are similar to ours. These works proposed to model systems by generalized semi-Markov processes (GSMP),an compositional extension of semi-Markov chains. However, the processes have uncountably many states. In [2], the processes are projected to a finite state space, which, unfortunately, is not

Markovian. Although it is possible to approximate the probability of a particular path in this finite state space, the calculation would suffer from high complexity. In [21], the GSMP is approximated with a continuous-time Markov process using phase-type distributions and thereafter turned to a discrete-time Markov process by *uniformization*. However, this method gives only approximate results. The algorithm in [11] adopted a similar technique to calculate the probability. Since the time delay for all transitions on the path from the beginning of enabledness to fire are independently determined, it is intuitive to perceive that the probability to execute the path can be calculated through integration on multidimensional random variable. A method using integration to model checking stochastic automata was informally discussed [5] through an example. No concurrency was shown in the example since it contained only one automaton. An integration formula is presented in [20] for the probability of executing a transition. However, no discussion on computing that formula was given in the paper. Similarly, the semantics of the GSMP model has been studied in the context of process algebra, e.g. [10, 17], but no means of computing the probability of a given path has been reported.

The rest of this paper is organised as follows. The necessary background on probability is given in Section 2. Section 3 describes transition systems with probabilistic distribution. In Section 4, we introduce how to calculate the probability of a path by integration and analyse the complexity of the calculation. Furthermore, we show the probability can be computed without integration when our method is applied to exponential distribution. Section 5 discusses optimizing the probability calculation for a partial order. Section 6 concludes the paper.

2 Probability Background

The content of this section is based on [3]. Let $Y_1, Y_2, \ldots, Y_n$ be independent continuous random variables on a common probability space. Let $y_1, y_2, \ldots, y_n$ be their values and $f_1(y_1), f_2(y_2), \ldots, f_n(y_n)$ be their density functions. The n-dimensional vector $(Y_1, Y_2, \ldots, Y_n)$ is called a *multidimensional random variable*, whose domain is a set of ordered vectors $(y_1, y_2, \ldots, y_n)$. Its density function is $f(y_1, \ldots, y_n)$. The distribution of the vector, also called the *joint distribution* of these variables, is the probability distribution over the region defined by

$$P(B) = P((Y_1, Y_2, \ldots, Y_n) \in B),$$

where B is a region in the n-dimensional space. The density function of the vector is

$$f(y_1, \ldots, y_n) = f_1(y_1) \cdots f_n(y_n).$$

Therefore, $P(B)$ is given by the following integral:

$$P(B) = \int \cdots \int_{(Y_1, Y_2, \ldots, Y_n) \in B} f_1(y_1) \cdots f_n(y_n) \; \mathrm{d}y_n \cdots \mathrm{d}y_1.$$

The probability density function for the continuous uniform distribution on the interval $[l, u]$ $(l < u)$ is

$$f(x) = \begin{cases} \frac{1}{u-l} & l \leq x \leq u; \\ 0 & \text{elsewhere.} \end{cases}$$

For the sake of simplicity, we say that $f(x)$ is $\frac{1}{u-l}$ in this paper.

Suppose $Y_1, Y_2, \ldots, Y_n$ are continuous random variables with joint density $f(y_1, y_2, \ldots, y_n)$ and $X_1 = g_1(Y_1, Y_2, \ldots, Y_n), \cdots, X_n = g_n(Y_1, Y_2, \ldots, Y_n)$. The joint density of $(X_1, \ldots, X_n)$ is then given by $f(y_1, \ldots, y_n)|J|$ where $|J|$ is the determinant of the Jacobian of the variable transformation, given by

$$J = \begin{pmatrix} \partial y_1/\partial x_1 & \cdots & \partial y_1/\partial x_n \\ \vdots & \vdots & \vdots \\ \partial y_n/\partial x_1 & \cdots & \partial y_n/\partial x_n \end{pmatrix}$$

3 The Model

3.1 Transition Systems

Definition 1. *A* transition system $\mathcal{T} = \langle V, \Sigma \rangle$ *includes*

- *A finite set V of program variables.*
- *A finite set Σ of transitions. Each transition $\alpha \in \Sigma$ includes the following components.*
  - *A first order predicate $en(\alpha)$ over the variables V. This is called the* enabling condition *of α.*
  - *A transformation $F_\alpha : V \rightarrow V$ on the program variables V. This transformation is applied to the program variables if a transition is taken.*
  - *l_α the lower bound on the period during which $en(\alpha)$ holds before α can be executed.*
  - *u_α the upper bound on the period during which $en(\alpha)$ holds before α must be executed. We require that $l_\alpha < u_\alpha$.*
  - *$cl(\alpha)$ is a local clock that measures the amount of time since α became enabled. It is a count-down clock [2].*

For simplicity, both l_α and u_α are limited to non-negative integers, though we use dense time clocks. In addition to the clocks per each transition, we have a global clock *gt*. The global clock is used to measure the time elapsed since the system began to run so that its reading keeps increasing after being started.

The probabilistic characteristic of this system lies in the following behavior: when a transition α becomes enabled, its clock $cl(\alpha)$ is set to an initial value which is chosen randomly from the interval $[l_\alpha, u_\alpha]$ according to uniform distribution. Then the clock begins to count down. When it reaches 0, α is triggered. If α is disabled before $cl(\alpha)$ reaches 0, $cl(\alpha)$ is set to 0 as well, but α is not triggered. Any clock except the global clock stops running when its reading is 0.

Definition 2. *A* state *of a transition system* $\mathcal{T} = \langle V, \Sigma \rangle$ *contains an assignment to the program variables* V *and a non-negative real-time value for each transition clock* $cl(\alpha)$ *(* $\alpha \in \Sigma$*) and for the global clock* gt*.*

We denote the value of a clock $cl(\alpha)$ in a state s by $cl(\alpha)(s)$ and the value of gt in s by $cl(gt)(s)$. Similarly, the value of a variable v in s is $v(s)$. We also generalize this and write $V(s)$ for the valuation of all the variables V at the state s. A transition α is *enabled* at state s if $en(\alpha)$ holds in s; then we say that $s \models en(\alpha)$ holds.

Definition 3. *An* initial state *of a transition system* $\mathcal{T} = \langle V, \Sigma \rangle$ *assigns a non-negative value to the clock of every transition* $\alpha \in \Sigma$ *that is enabled in the initial state. Each value is chosen randomly between* l_α *and* u_α *according to the uniform distribution. The initial state assigns the value* 0 *to every clock* $cl(\alpha')$ *for all disabled* $\alpha' \in \Sigma$ *in the initial state and to the global clock.*

Definition 4. *A* system $\mathcal{S}$ *is a pair* $\langle \mathcal{T}, s \rangle$ *with* s *an initial state of* $\mathcal{T}$*. Thus, we assume each system has a given initial state.*

Probabilistic behavior is reflected in repeated executions, not a single one. We define a *probabilistic execution* of the system induced from multiple executions.

Definition 5. *A probabilistic execution of a system* $\mathcal{S}$ *is a finite sequence of the form* $s_0 g_1 \alpha_1 s_1 g_2 \alpha_2 \ldots$ *where* s_i*,* g_i *are states, and* α_i *are transitions. An execution has to satisfy the following constraints:*

- s_0 *is the initial state of* $\mathcal{S}$*.*
- *For any adjacent pair of states* s_i*,* g_{i+1} *(representing time passing):*
  - *For the clock of any enabled transition* α *we have that*
    $$cl(\alpha)(s_i) - cl(\alpha)(g_{i+1}) = cl(gt)(g_{i+1}) - cl(gt)(s_i),$$
    which means that all clocks move at the same speed. The clock of any disabled transition β *remains 0.*
  - *For every variable* $v \in V$*,* $v(s_i) = v(g_{i+1})$*.*
- *For any sequence* g_i*,* α_i*,* s_i *on the path (the execution of transition* α_i*), the following hold:*
  - $g_i \models en(\alpha_i)$ *and* $V(s_i) = F_{\alpha_i}(V(g_i))$*.*
  - $cl(\alpha_i)(g_i) = 0$ *and* $cl(gt)(g_i) = cl(gt)(s_i)$
  - *For each* $\beta \in \Sigma$ *we have that if* $g_i \models \neg en(\beta)$ *and* $s_i \models en(\beta)$ *(*β *became enabled by the execution of* α_i*), or* $\beta = \alpha_i$ *and* $s_i \models en(\beta)$ *(although* $\alpha_i = \beta$ *was executed, it is enabled immediately again), then* $cl(\beta)(s_i)$ *is a random non-negative value which is chosen between* l_β *and* u_β *according to the uniform distribution. Or if* $g_i \models en(\beta)$ *and* $s_i \models \neg en(\beta)$*, then* $cl(\beta)(s_i) = 0$*. That is, when a transition becomes disabled, its clock is set to zero. Otherwise,* $cl(\beta)(g_i) = cl(\beta)(s_i)$*.*
  - *For* $\beta \neq \alpha_i$ *such that* $g_i \models en(\beta)$ *we have that* $cl(\beta)(g_i) > 0$*. The reason is that the probability of two events being triggered at the same time*[1] *is 0 from probability point of view. Detailed discussion can be seen in [11].*

[1] Here a pair of synchronized transition is considered as one transition.

Definition 6. *A* path *is a finite sequence of transitions $\sigma = \alpha_1\alpha_2\alpha_3\ldots$. A path σ is* consistent *with an execution ρ if σ is obtained by removing all the state components of ρ.*

3.2 Calculating Participating Transitions

Since transitions may occur several times on a path, we refer to the *occurrences* of transitions. So we either rename or number different repeated occurrences of transitions to be different. In fact, we need to look not only on the transitions that occur on the path, but also on those that become enabled (and disabled).

As a preliminary step for the calculation performed in the next section, we need to compute the transitions that are enabled given a path σ (but not necessarily executed). Assume that the transitions are executed according to the order in σ. Then consider the states of the form g_i, where the transition α is fired. We can ignore now the timing considerations, thus also the states of the form s_j (that are the same as the g_i states). The states $g_1, \ldots, g_n$ are easily calculated, as $V(g_i) = F_{\alpha_i}(V(g_{i-1}))$. Moreover, it is easy to calculate whether $g_i \models en(\beta)$ holds for $0 \leq i \leq n$.

Definition 7. *The* enabledness period *of an occurrence of a transition α_j, is a maximal interval, where it is enabled. That is, each such interval is bounded by states g_i and g_j such that for each $i \leq k \leq j$, $g_k \models en(\alpha_j)$. Furthermore, g_i and g_j are maximal in the sense that these conditions do not hold for the interval g_{i-1} to g_j (if $i > 0$) nor for the interval g_i to g_{j+1} (if $i < n$).*

Thus, the transitions participating in a path σ are those that have a nonempty enabledness period.

3.3 A Partial Order Between Path Events

It is commonly argued that specifying a given sequence of events is less natural than specifying a partial order between them. In particular, some events are local to their separate processes. In these cases, it is often the case that they can occur in either order. Given time constraints, it is possible that such an order exists, or that the occurrence of two local independent transitions cannot appear in the order specified by a given sequence.

Definition 8. *We add to the definition of a transition system a set of* processes *P. Furthermore, we have a function $p : \Sigma \rightarrow 2^P$ that describes to each transition the set of processes in which they are* involved*. Thus, we redefine a transition system as $\mathcal{T} = \langle V, \Sigma, P, p \rangle$.*

Definition 9. *For a transition system with a set of transitions Σ, we have that $\alpha, \beta \in \Sigma$ are* independent *iff $p(\alpha) \cap p(\beta) = \emptyset$. That is, if α and β are associated with disjoint processes. We denote the dependency relation over Σ by $D \subseteq \Sigma \times \Sigma$.*

Definition 10. *Let σ be a path, i.e., a sequence of occurrences of transitions. Then $<_\sigma$, the* essential partial order *between occurrences from σ is defined for α, β in σ as follows: $\alpha <_\sigma \beta$ iff α appears in σ before β and $\alpha D \beta$.*

We can now define the *partial order* $<^*_\sigma$ between occurrences on a path σ as the reflexive and transitive closure of $<_\sigma$.

3.4 An Example System

In Figure 1, a system is composed of three processes, each of which is represented by a subsystem. Process 1 has four transitions a, b, c, d, process 2 has one transition g and process has one transitions h. The transitions of the different subsystems are being put together into the set of transitions $\mathcal{T}$. The variables $V = \{s, v, w\}$ representing the "program counters" of the different processes are used to control the enabledness of these transitions. The initial state of the system has the assignment $s = 1$, $v = 1$ and $w = 1$. The bounds for every transition are noted in the figure. The density functions of transitions in this system are $f_a = f_g = f_h = \frac{1}{4}$, $f_b = f_c = \frac{1}{3}$ and $f_d = \frac{1}{2}$.

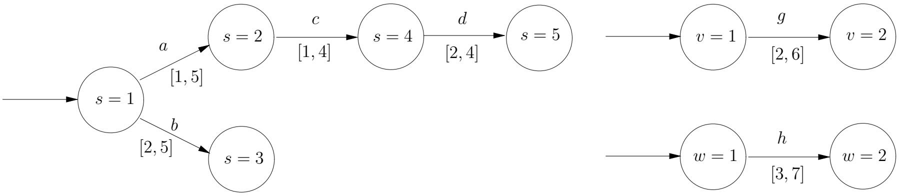


Fig. 1. The example system

It is quite easy to model various kinds of concurrency constructs using transition systems [15], for example, synchronous communication can be represented as a transition that involves two processes (hence affecting program counters of both processes) assigning values from the sending variables to the receiving one.

4 The Probability of a Path

4.1 Timing Relations Along a Path

As per Definition 6, a path ρ is a sequence of transitions $\alpha_1\alpha_2 \ldots \alpha_n$. The corresponding execution is $s_0 g_1 \alpha_1 s_1 g_2 \alpha_2 s_2 g_3 \ldots s_{n-1} g_n \alpha_n s_n$. The transition α_i ($1 \leq i \leq n$) has the lower bound l_i and the upper bound u_i. The path is executed from state s_0 at global time 0, which is represented as x_0. Let x_i be $cl(gt)(g_i)$ or, equivalently, $cl(gt)(s_i)$, i.e., the value of the global clock, when the transition α_i is fired. So we have a sequence of global time points $x_0 x_1 \ldots x_n$. We obtain the relation among these time points:

$$x_0 < x_1 < \cdots < x_n. \tag{1}$$

For a transition α_j that becomes enabled at x_i and is triggered at x_j, the duration of its enabledness, which is decided by the initial value of $cl(\alpha_j)(s_i)$ at x_i, satisfies the formula:

$$l_j \leq x_j - x_i \leq u_j. \tag{2}$$

Now let us consider an occurrence of a transition α' that does not appear in the given path but is first enabled at s_i, while disabled at s_j $(0 \leq i < j \leq n)$ or remains enabled after s_n. In the latter case, α' is said to be disabled by the end of path at x_n. Thus we need not distinguish these two cases. Let x_i and x_j be the global time points with respect to s_i and s_j respectively. $cl(\alpha')(s_i)$ is the initial value of the clock when α' becomes enabled. Let $x_{\alpha'} = x_i + cl(\alpha')(s_i)$. Then $x_{\alpha'}$ satisfies the following formulae:

$$l_{\alpha'} \leq x_{\alpha'} - x_i \leq u_{\alpha'}, \tag{3}$$

$$x_j < x_{\alpha'}. \tag{4}$$

Obviously, $x_{\alpha'} - x_i$, the initial clock value of α', is bounded by the lower bound and the upper bound of α'. The formula (4) must hold because otherwise α' would have been triggered before α_j is triggered and would have appeared in the path. All of the occurrences of transitions that are enabled at some states and disabled at some later states form the set $\{\alpha'_1, \alpha'_2, \ldots, \alpha'_m\}$. Every x_i or $x_{\alpha'_k}$ is the value of a random variable X_i or X'_k.

Consider for example a path $\rho = ag$ of the system in Figure 1. At time x_0, there are four enabled transitions a, b, g and h. According to ρ, a is fired earlier than others at time x_1. At this point, b is disabled, while g and h are continuously enabled. Also, c becomes enabled at x_1. To make it clear, we use x_a to replace x_1 and x_g to replace x_2. Therefore, we obtain constraints

$1 \leq x_a \leq 5$	(formula (2))
$2 \leq x_b \leq 5$	(formula (3))
$x_a < x_b.$	(formula (4))

At time x_2 after g is fired, both h and c are disabled by the end of the path. We obtain

$2 \leq x_g \leq 6$	(formula (2))
$3 \leq x_h \leq 7$	(formula (3))
$x_a + 1 \leq x_c \leq x_a + 4$	(formula (3))
$x_a < x_g$	(formula (1))
$x_g < x_h$	(formula (4))
$x_g < x_c.$	(formula (4))

The final constraint for the path is as follows:

$$(1 \leq x_a \leq 5) \wedge (2 \leq x_g \leq 6) \wedge (3 \leq x_h \leq 7) \wedge (2 \leq x_b \leq 5) \wedge$$
$$(x_a + 1 \leq x_c \leq x_a + 4) \wedge (x_a < x_b) \wedge (x_a < x_g) \wedge (x_g < x_h) \wedge (x_g < x_c).$$

4.2 Computing the Probability of a Path

For a path in which n transitions are taken and m transitions are not taken but have been enabled for a period of time, calculating the probability that the path is executed will involve $n+m$ independent random variables $Y_1, \ldots, Y_n, Y'_1, \ldots, Y'_m$. Each transition has a corresponding random variable whose value is the initial value of the clock of the transition. The fact that a transition is enabled after another does not make their corresponding random variable dependent on each other because on every execution of a path, the initial value of the clock of a transition is chosen independently according to its probability distribution. Any transitions that are never enabled along the path do not compete with the transitions in the path for execution. Thus they do not contribute to the probability and need not be considered in the calculation of the probability.

The probability distribution of the path is the joint distribution of $Y_1, \ldots, Y_n$, $Y'_1, \ldots, Y'_m$. Let f_i $(1 \leq i \leq n)$ and f'_k $(1 \leq k \leq m)$ be the density function of Y_i and Y'_k respectively. Let l_i and u_i be the lower bound and the upper bound of Y_i and l'_k and u'_k be the lower bound and the upper bound of Y'_k. Thus, $f_i = \frac{1}{u_i - l_i}$ and $f'_k = \frac{1}{u'_k - l'_k}$. To calculate the probability, we use variable transformation from $\{Y_1, \ldots, Y_n, Y'_1, \ldots, Y'_m\}$ to $\{X_i | 1 \leq i \leq n\} \cup \{X'_k | 1 \leq k \leq m\}$, because the time constraint is defined on the latter set of variables. Let y_j and y'_k be the value of the variable Y_j and Y'_k. We have $Y_j = X_j - X_i$ according to the formula (2), $Y'_k = X'_k - X_i$ according to the formula (3) and $X_0 = 0$. The density function $f(x_1, \ldots, x_n, x'_1, \ldots, x'_m) = f(y_1, \ldots, y_n, y'_1, \ldots, y'_m)|J|$.

$$J = \begin{pmatrix} \partial y_1/\partial x_1 & \cdots & \partial y_1/\partial x_n & \partial y_1/\partial x'_1 & \cdots & \partial y_1/\partial x'_m \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \partial y_n/\partial x_1 & \cdots & \partial y_n/\partial x_n & \partial y_n/\partial x'_1 & \cdots & \partial y_n/\partial x'_m \\ \partial y'_1/\partial x_1 & \cdots & \partial y'_1/\partial x_n & \partial y'_1/\partial x'_1 & \cdots & \partial y'_1/\partial x'_m \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \partial y'_m/\partial x_1 & \cdots & \partial y'_m/\partial x_n & \partial y'_m/\partial x'_1 & \cdots & \partial y'_m/\partial x'_m \end{pmatrix}$$

Since $\partial y_j/\partial x_j = 1$, $\partial y_j/\partial x_i = 0$ for every $i > j$, $\partial y_j/\partial x'_k = 0$ for every $1 \leq k \leq m$, $\partial y'_k/\partial x'_k = 1$, and $\partial y'_k/\partial x'_i = 0$ for every $i > k$, J is a lower triangular square matrix and every diagonal element is 1. Thus the determinant of Jacobian matrix is 1.

$$f(x_1, \ldots, x_n, x'_1, \ldots, x'_m) = f(y_1, \ldots, y_n, y'_1, \ldots, y'_m) = \left(\prod_{i=1}^{n} f_i\right) \cdot \left(\prod_{k=1}^{m} f'_k\right).$$

Now we calculate the probability over $X_1, \ldots, X_n, X'_1, \ldots, X'_m$.

The formulae (1)-(4) characterize the constraint for a path. The constraint defines a region B in the $(n+m)$-dimensional space. Let $P[\rho] = P[\alpha_1\alpha_2 \ldots \alpha_n]$ be the probability of the path ρ. $P[\rho]$ is calculated by

$$P[\rho] = \int \cdots \int_{(X_1, \ldots, X_n, X'_1, \ldots, X'_m) \in B} f_1 \cdots f_n f'_1 \cdots f'_m \, \mathrm{d}x'_m \cdots \mathrm{d}x'_1 \mathrm{d}x_n \cdots \mathrm{d}x_1. \qquad (5)$$

The formula is calculated from innermost integral to outermost integral. Let $Z = \{X_1, \ldots, X_n, X'_1, \ldots, X'_m\}$. The integral range of variable $z_j \in Z$ might depend on the range of $z_i \in Z$ for $i < j$. However, a technical difficulty exists in the above formula: the dependency relation is different in the different parts of range of z_i. In our example,

$$P[ag] = \int_1^5 \left(\int_{max_1}^6 \left(\int_{max_2}^7 \left(\int_{max_3}^5 \left(\int_{max_4}^{x_a+4} \frac{1}{576}\, \mathrm{d}x_c\right)\mathrm{d}x_b\right)\mathrm{d}x_h\right)\mathrm{d}x_g\right)\mathrm{d}x_a,$$

where $max_1 = \max\{2, x_a\}$, $max_2 = \max\{3, x_g\}$, $max_3 = \max\{2, x_a\}$, $max_4 = \max\{x_a + 1, x_g\}$ and $\frac{1}{576} = \frac{1}{4} \times \frac{1}{3} \times \frac{1}{4} \times \frac{1}{4} \times \frac{1}{3}$. These max functions are deduced from the constraints of the path ag. At different parts of region, the functions obtain different values.

Therefore, the region needs to be split into a set of disjoint blocks, in each of which the dependence relation is fixed. Then the integration is performed over every block and the results are summed up. The region-splitting can be done using the Fourier-Motzkin elimination method [18]. The constraint of path ag is split into eight disjoint regions:

$$\begin{array}{l}
1 \le x_a < 2 \wedge 2 \le x_g < 1+x_a \wedge 3 \le x_h \le 7 \wedge 2 \le x_b \le 5 \wedge 1+x_a \le x_c \le 4+x_a \\
1 \le x_a < 2 \wedge 1+x_a \le x_g < 3 \wedge 3 \le x_h \le 7 \wedge 2 \le x_b \le 5 \wedge x_g < x_c \le 4+x_a \\
1 \le x_a < 2 \wedge 3 \le x_g < 4+x_a \wedge x_g < x_h \le 7 \wedge 2 \le x_b \le 5 \wedge x_g < x_c \le 4+x_a \\
2 < x_a < 3 \wedge x_a < x_g < 3 \wedge 3 \le x_h \le 7 \wedge x_a < x_b \le 5 \wedge 1+x_a \le x_c \le 4+x_a \\
2 < x_a < 3 \wedge 3 \le x_g < 1+x_a \wedge x_g < x_h \le 7 \wedge x_a < x_b \le 5 \wedge 1+x_a \le x_c \le 4+x_a \\
2 < x_a < 3 \wedge 1+x_a \le x_g \le 6 \wedge x_g < x_h \le 7 \wedge x_a < x_b \le 5 \wedge x_g < x_c \le 4+x_a \\
3 \le x_a < 5 \wedge x_a < x_g < 1+x_a \wedge x_g < x_h \le 7 \wedge x_a < x_b \le 5 \wedge 1+x_a \le x_c \le 4+x_a \\
3 \le x_a < 5 \wedge 1+x_a \le x_g \le 6 \wedge x_g < x_h \le 7 \wedge x_a < x_b \le 5 \wedge x_g < x_c \le 4+x_a.
\end{array}$$

Each region describes the integral range for every variable. The following are these eight integrals:

$$\int_1^2 \left(\int_2^{x_a+1} \left(\int_3^7 \left(\int_2^5 \left(\int_{x_a+1}^{x_a+4} \frac{1}{576}\mathrm{d}x_c\right)\mathrm{d}x_b\right)\mathrm{d}x_h\right)\mathrm{d}x_g\right)\mathrm{d}x_a$$

$$\int_1^2 \left(\int_{x_a+1}^3 \left(\int_3^7 \left(\int_2^5 \left(\int_{x_g}^{x_a+4} \frac{1}{576}\mathrm{d}x_c\right)\mathrm{d}x_b\right)\mathrm{d}x_h\right)\mathrm{d}x_g\right)\mathrm{d}x_a$$

$$\int_1^2 \left(\int_3^{x_a+4} \left(\int_{x_g}^7 \left(\int_2^5 \left(\int_{x_g}^{x_a+4} \frac{1}{576}\mathrm{d}x_c\right)\mathrm{d}x_b\right)\mathrm{d}x_h\right)\mathrm{d}x_g\right)\mathrm{d}x_a$$

$$\int_2^3 \left(\int_{x_a}^3 \left(\int_3^7 \left(\int_{x_a}^5 \left(\int_{x_a+1}^{x_a+4} \frac{1}{576}\mathrm{d}x_c\right)\mathrm{d}x_b\right)\mathrm{d}x_h\right)\mathrm{d}x_g\right)\mathrm{d}x_a$$

$$\int_2^3 \left(\int_3^{x_a+1} \left(\int_{x_g}^7 \left(\int_{x_a}^5 \left(\int_{x_a+1}^{x_a+4} \frac{1}{576}\mathrm{d}x_c\right)\mathrm{d}x_b\right)\mathrm{d}x_h\right)\mathrm{d}x_g\right)\mathrm{d}x_a$$

$$\int_2^3 \left(\int_{x_a+1}^6 \left(\int_{x_g}^7 \left(\int_{x_a}^5 (\int_{x_g}^{x_a+4} \frac{1}{576} \mathrm{d}x_c) \mathrm{d}x_b \right) \mathrm{d}x_h \right) \mathrm{d}x_g \right) \mathrm{d}x_a$$

$$\int_3^5 \left(\int_{x_a}^{x_a+1} \left(\int_{x_g}^7 \left(\int_{x_a}^5 (\int_{x_a+1}^{x_a+4} \frac{1}{576} \mathrm{d}x_c) \mathrm{d}x_b \right) \mathrm{d}x_h \right) \mathrm{d}x_g \right) \mathrm{d}x_a$$

$$\int_3^5 \left(\int_{x_a+1}^6 \left(\int_{x_g}^7 \left(\int_{x_a}^5 (\int_{x_g}^{x_a+4} \frac{1}{576} \mathrm{d}x_c) \mathrm{d}x_b \right) \mathrm{d}x_h \right) \mathrm{d}x_g \right) \mathrm{d}x_a .$$

$P[ag]$ is $\frac{1489}{5760}$, the sum of these eight integrals[2].

4.3 The Complexity of the Computation

The computation of the integration over the range defined by a block has linear complexity, because there are no cyclic references, the density functions do not contain integration variables and all of integration bounds are linear. The general complexity of the Fourier-Motzkin (FM) elimination method is $O((\frac{\overline{m}}{2})^{2^{\overline{n}}})$, where $\overline{m}$ is the number of inequalities and $\overline{n}$ is the number of variables. In the constraint of a path, any linear inequality has at most two variables and every coefficient belongs to the set $\{-1, 0, 1\}$. Now we give a brief estimation of the complexity of the FM method for this special case. The FM method has $\overline{n}$ recursive steps, each of which deals with one variable. Each step generate some new inequalities. The number of new inequalities is maximal when $\frac{\overline{m}}{2}$ inequalities have a positive coefficient of the variable and other $\frac{\overline{m}}{2}$ inequalities have a negative coefficient. The maximal number is $\frac{\overline{m}^2}{4}$. After $\overline{n}$ steps, we gain the general complexity. Since we have $(n+m)$ transitions and at most three inequalities for each transition (which can be easily deduced from the formulae (1)-(4)), and there are at most two variables per inequality, there are at most $6(n+m)$ coefficients in all the inequalities. To produce the maximal number of new inequalities, each variable can appear in 6 inequalities. Three of them have a positive coefficient and other three have a negative coefficient. Therefore, we obtain $(3 \times 3)^{n+m}$ new inequalities in the end. Then the complexity is $O(9^{n+m})$ in our case. Each new inequality means a region is split, which means we may get exponentially increased number of integral regions. However, many of them are redundant. In practice, we expect only a small number of split regions. For example, for path ag, we obtain 8 regions, far less than 9^5.

On the other hand, the calculation of the formula (5) can be seen as the problem to calculate the volume of the region defined by the constraint because the joint density function is a constant. Since the constraints contain only conjunction and linear inequalities, the region is convex linear. The computational complexity of computing the volume of a convex body defined by linear inequalities is $\#P$-hard [8]. But approximation algorithms for computing volume can be polynomial. The fastest algorithm is $O^*(n^4)$ [13], where n is the dimension of

[2] These integrals were calculated using Mathematica.

the body. We also expect to find (but have not found yet) some helpful results for our special case to accelerate computing volumes.

4.4 Simplification for Exponential Distribution

We have already mentioned in the first section that the calculation could be simple if the delay of each transition obeys exponential distribution, since the system is Markovian. Now we give a short description how exponential distribution simplifies calculation. An exponential distribution is defined on interval $[0, \infty]$ with rate λ and density function $\lambda \cdot e^{-\lambda \cdot x}$. At first, the formula (2) is changed to $0 \leq x_j - x_i < \infty$, which is simplified to $x_i \leq x_j < \infty$. Because of $x_{j-1} < x_j$, the formula (2) is simplified to $x_{j-1} < x_j < \infty$ further, which makes the formula (1) is redundant and removed. Similarly, the formula (3) is simplified to $x_i \leq x_{\alpha'} < \infty$. Due to $x_i < x_j$, this formula and the formula (4) are merged to $x_j < x_{\alpha'} < \infty$. For example, consider all transitions in Figure 1 are following exponential distribution. The density functions for transitions a, b, c, d, g and h are $\lambda_a \cdot e^{-\lambda_a \cdot x_a}$, $\lambda_b \cdot e^{-\lambda_b \cdot x_b}$, $\lambda_c \cdot e^{-\lambda_c \cdot x_c}$, $\lambda_d \cdot e^{-\lambda_d \cdot x_d}$, $\lambda_g \cdot e^{-\lambda_g \cdot x_g}$ and $\lambda_h \cdot e^{-\lambda_h \cdot x_h}$. The constraint for path ag is ($x_a = x_1$ and $x_g = x_2$)

$$(0 \leq x_a < \infty) \wedge (x_a < x_g < \infty) \wedge (x_g < x_h < \infty) \wedge (x_a < x_b < \infty) \wedge (x_g < x_c < \infty).$$

After simplifying the formulae (1)-(4), each variable x_β with respect to transition β in time constraint is bounded by an inequality of the form $x_\gamma < x_\beta < \infty$, where x_γ is the time point at which the $(j-1)$th transition in the path is triggered if β is the jth transition ($x_\gamma = x_0$ if β is the first transition), or at which β is disabled if β does not appear in the path. Moreover, this constraint need not be split into smaller integral blocks.

The integration over this constraint can be done inductively as follows. For all transitions that do not appear in the path, their lower integral bounds are the time points at which they become disabled. Let β be such a transition with constraint $x_\gamma < x_\beta < \infty$. The integration over x_β is

$$\int_{x_\gamma}^{\infty} \lambda_\beta \cdot e^{-\lambda_\beta \cdot x_\beta} \mathrm{d}x_\beta = e^{-\lambda_\gamma \cdot x_\gamma}. \tag{6}$$

For a path with n transitions, we divide the transitions not appearing in the path into n groups such that the transitions in group G_i is disabled by the ith transition. Integration over variables corresponding to transitions in a non-empty group G_i where each transition $\beta_k \in G_i$ has rate $\lambda_{i,k}$ produces $e^{-(\sum \lambda_{i,k}) \cdot x_i}$ by applying the formula (6) for each β_k and multiply their results.

After eliminating all variables corresponding to the transitions not triggered from integral, we need to calculate the integral over variables corresponding to the transitions in the path. Now the integral for such a variable x_j is of the form

$$\int_{x_{j-1}}^{\infty} \lambda_j \cdot e^{-(\lambda_j + \sum \lambda_{j,k}) \cdot x_j} \mathrm{d}x_j.$$

We calculate the integration from variable x_n to x_1 recursively. It is easy to prove by induction that after we calculate the integration over x_{j+1}, the integral over x_j is as follows:

$$\int_{x_{j-1}}^{\infty} \lambda_j \cdot \prod_{i=j+1}^{n} \frac{\lambda_i}{\lambda_i + \sum \lambda_{i,k}} \cdot e^{-(\sum_{i=j}^{n}(\lambda_i + \sum \lambda_{i,k}))\cdot x_j} \mathrm{d}x_j.$$

Finally, the result of the integration after variable x_1 is below:

$$\prod_{j=1}^{n} \frac{\lambda_j}{\sum_{i=j}^{n}(\lambda_i + \sum \lambda_{i,k})}. \tag{7}$$

For example, the probability of path ag is calculated as follows:

$$\int_0^{\infty} \lambda_a \cdot e^{-\lambda_a \cdot x_a} \Bigg(\int_{x_a}^{\infty} \lambda_g \cdot e^{-\lambda_g \cdot x_g} \bigg(\int_{x_g}^{\infty} \lambda_h \cdot e^{-\lambda_h \cdot x_h}$$

$$\Big(\int_{x_a}^{\infty} \lambda_b \cdot e^{-\lambda_b \cdot x_b} \big(\int_{x_g}^{\infty} \lambda_c \cdot e^{-\lambda_c \cdot x_c} \mathrm{d}x_c \big) \mathrm{d}x_b \Big) \mathrm{d}x_h \bigg) \mathrm{d}x_g \Bigg) \mathrm{d}x_a =$$

$$\int_0^{\infty} \lambda_a \cdot e^{-(\lambda_a + \lambda_b)\cdot x_a} \big(\int_{x_a}^{\infty} \lambda_g \cdot e^{-(\lambda_g + \lambda_h + \lambda_c)\cdot x_a} \mathrm{d}x_g \big) \mathrm{d}x_a =$$

$$\frac{\lambda_g}{\lambda_g + \lambda_h + \lambda_c} \cdot \frac{\lambda_a}{\lambda_a + \lambda_b + \lambda_g + \lambda_h + \lambda_c}.$$

The formula (7) shows that for a system where all transition delays obey exponential distribution, we do not need to calculate the integration for the probability of a path, instead to calculate the probability directly over all rates.

5 Optimizing the Calculation for a Partial Order

A partial order represents a group of equivalent paths that has the same essential partial order. Given a path and an independence relation between its transitions (representing pairs of transitions that can concurrently overlap), one can generate the partial order relation $<_{\sigma}^{*}$, see Section 3.3. A simple way to calculate the probability of executing a partial order is to sum up the probabilities of all equivalent paths. But this calculation can be optimized. For example, consider the system in Section 3.4. The partial order $\langle a, g \rangle$ containing a and g and no order relation between them represents two paths: ag and ga. The constraint of ga is

$$(2 \leq x_g \leq 6) \wedge (1 \leq x_a \leq 5) \wedge (x_g < x_a) \wedge$$
$$(2 \leq x_b \leq 5) \wedge (3 \leq x_h \leq 7) \wedge (x_a < x_b) \wedge (x_a \leq x_h).$$

It is split into two integral regions. So in total, there are ten integrals to calculate the probability of the partial order[3]. On this partial order, a could be triggered earlier or later than g and thus we do not have relation $x_a < x_g$ or $x_a > x_g$. The key constraint for the partial order is

$$(x_a < x_b) \wedge (x_a < x_h) \wedge (x_g < x_h) \wedge (x_g < x_c).$$

[3] The FM method cannot handle disjunction of two sets of inequalities.

The conjunction of this constraint and the basic constraint

$$(1 \leq x_a \leq 5) \wedge (2 \leq x_g \leq 6) \wedge (3 \leq x_h \leq 7) \wedge (2 \leq x_b \leq 5) \wedge (x_a + 1 \leq x_c \leq x_a + 4)$$

is split into nine blocks. Thus, it is possible to give heuristics for optimizing the calculation of the probability of a partial order. (It is possible to merge two adjacent blocks into one when they are generated from different paths. Indeed, there are such two blocks when we split the constraints for paths ag and ga separately. But it wastes time to generate them first and merge them later. It is better to generate the merged block directly from a simple constraint.)

The key idea for optimizing probability calculation for a partial order is that we remove from the time constraint of the partial ordering relations between any pair of transitions that can be fired concurrently. Hence the time constraint of the partial order only describes the necessary relations that guarantee the partial order. For instance, the relations $x_a < x_g$ and $x_g < x_a$ between transitions a and g in the above example are not necessary for the partial order $\langle a, g \rangle$ since either a or g can be triggered first. The time constraint for a partial order cannot be constructed simply as the disjunction of time constraints for all linearizations of the partial order with removing the unnecessary relations among concurrent transitions. The reason is that other relations in a time constraint of a linearization may depend on the ordering relations among concurrent transitions on this linearization.

The time relation defined by formulae (1)-(4) gives time constraint for a linear sequence. To allow maximum concurrency, we need to release the relation so that it can describe a partial order. Indeed, if two transitions do not depend on each other, either one can be fired earlier than the other. Thus we remove inequality between their execution time points. We need to modify formulae (1) and (2) to reflect this change. For two transitions α and β such that the firing of α enables β, we obtain $l_\beta < x_\beta - x_\alpha < u_\beta$. In a general case that β becomes enabled only after all transitions $\alpha_1, \ldots, \alpha_n$ are fired, i.e., β depends on $\alpha_1, \ldots, \alpha_n$,

$$l_\beta < x_\beta - \max\{x_{\alpha_1}, \ldots, x_{\alpha_n}\} < u_\beta. \tag{8}$$

Particularly, if $n = 0$, we have $l_\beta < x_\beta < u_\beta$. The formula (8) for every transition can be gained statically from the partial order.

Let ρ_i and ρ_j be projected path of the partial order on Process i and j. Let x_{n_i} (x_{n_j}) be the end time point on ρ_i (ρ_j) when the last transition on ρ_i (ρ_j) is triggered. For any transition α' belonging to Process i and remaining enabled after x_{n_i} (including the newly enabled transitions at x_{n_i}), we have the following relation in order to ensure α' cannot be triggered earlier than all transitions appearing on ρ_j:

$$x_{\alpha'} > x_{n_j}, \tag{9}$$

where $x_{\alpha'}$ is the same time point as the one in formula (4).

However, the optimization is complicated when handling a joint transition belonging to two processes. Figure 2 depicts a system consisting of two processes. For brevity, only labels of transitions are shown. Transitions c is a joint transition. Consider the partial order $a \rightarrow b, d \rightarrow g$. The possible paths represented by the

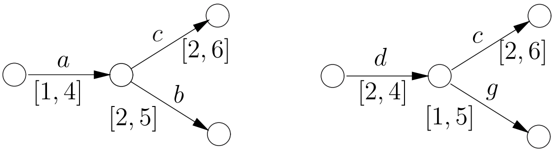


Fig. 2. A system example

partial order are *adbg*, *adgb*, *dabg*, *dagb*, *abdg* and *dgab*. Let x_a, x_b, x_c, x_d, x_g be the variables corresponding to a, b, c, d, g respectively. On the paths *abdg* and *dgab*, c is not enabled because during the execution of these paths, the system never reaches a state containing the two source location of c in both processes. The key time constraint to describe this situation is

$$x_a > x_g \vee x_d > x_b. \tag{10}$$

On paths *adbg*, *adgb*, *dabg* and *dagb*, c is first enabled at time point $\max\{x_a, x_d\}$ and disabled at $\min\{x_b, x_g\}$. The key time constraint is

$$(x_a < x_g \wedge x_d < x_b) \wedge (x_c > \min\{x_b, x_g\} \wedge l_c < x_c - \max\{x_a, x_d\} < u_c), \tag{11}$$

where l_c, u_c are the lower and the upper bound of c. However, since

$$(x_a > x_g \vee x_d > x_b) \wedge (x_c > \min\{x_b, x_g\} \wedge l_c < x_c - \max\{x_a, x_d\} < u_c)$$

is simplified to

$$(x_a > x_g \vee x_d > x_b) \wedge (l_c < x_c - \max\{x_a, x_d\} < u_c), \tag{12}$$

the integration on x_c is 1 when we integrate on the above formula if c is never enabled, which means the integration result on formula (12) is the same as that on formula (10). Therefore, from the integration point of view, we can use the following formula to describe the constraint for c, no matter c is enabled or not:

$$x_c > \min\{x_b, x_g\} \wedge l_c < x_c - \max\{x_a, x_d\} < u_c, \tag{13}$$

Note that x_a and x_d could be x_0, and x_b and x_g could be end time points on each processes.

Now the time constraint for a partial order is defined by the formulae (8), (9), (3–4) (for local transitions) and (13) (for joint transitions). We need to find all occurrences of joint transitions that could be enabled without being executed. One way is to generate all possible occurrences statically. But some occurrences might be prohibited by time constraints or enabling conditions. Another way is to execute all paths represented by the partial order to collect occurrences. Our experience shows that the optimization effect is significant when the partial order allows much concurrency. For example, in the partial order $a \to c, d \to c$, there are two equivalent paths *adc* and *dac*. The number of regions of the path *adc* after split is 2 and the number for the path *dac* is 1. After optimization being applied, the total number of regions is 3, which equals to the sum of regions of the two paths. However, for the partial order $a \to b, d \to g$, the sum of regions of the six paths is 38, while the number after optimization is 19, only a half of the sum.

6 Discussion and Conclusion

When performing unit testing of code, we often attempt to force the execution of a suspicious behavior. We can start by calculating some initial values for this behavior, e.g., by calculating the path condition [9]. However, due to concurrency, repeating the exact same behavior is not always guaranteed. A tester may instrument the code in such a way that the selected behavior is forced. However, such instrumentation alter the code, in particular, it may change the timing and consequently the interaction with external devices, which may be part of tested behavior. The alternative that we suggested in this paper, is to calculate the probability of the behavior to occur in order to obtain a tough estimate of expected number of repetitions we ought to perform the testing. Meaning that if the probability we calculated is p, then we should repeat our experiment $O(\frac{1}{p})$ times. If the desired test case hasn't happened, it shows that some information which we use to compute the probability is not accurate, such as probability distribution, or the time intervals of transitions. For example, although the time interval to execute some transition is $[2, 3]$, it actually, under the given testing parameters, be $[1, 4]$. This is also helpful for a tester to understand the system more accurately.

Our approach is also useful for optimizing test suites. One aspect is that we always want to run those test cases that have high probability first in order to save on time and resources. Another aspect is that given a choice of test cases, we may use the probability calculation in order to select the more likely cases.

Although we use uniform distribution in the example to demonstrate our methodology, the general formula for computing the probabilities holds for arbitrary distributions. This was shown for exponential distribution, because of the fact that the time constraint defined by the formulae (1)-(4) and the Jacobian for the random variable transformation are independent of probability distributions. A potential problem for any other distribution than uniform distribution and exponential distribution is that its density function contains the integration variable so that the calculation of an $(n+m)$-fold multiple integral on this kind of distributions could be much harder than the calculation on uniform distribution.

We also showed that our methodology can be optimized for a partial order. However, due to the complexity introduced by joint transitions, the effect of optimization can be counteracted by extra computation for handling this complexity, in particular, when there are many occurrences of joint transitions involved in the probability calculation.

References

1. R. Alur, Timed Automata, *NATO-ASI 1998 Summer School on Verification of Digital and Hybrid Systems*
2. R. Alur, C. Courcoubetis, D.L. Dill, Model-checking for probabilistic real-time systems, *Automata, Languages and Programming: Proceedings of the 18th ICALP*, LNCS 510, 115–136, 1991
3. A. A. Borovkov, Probability theory, Chapter 3, Gordon and Breach, 1998
4. M. Bravetti, P.R. D'Argenio, Tutte le algebre insieme: Concepts, Discussions and Relations of Stochastic Process Algebras with General Distributions, *GI/Dagstuhl Research Seminar Validation of Stochastic Systems*, LNCS 2925, 44–88, 2004
5. J. Bryans, H. Bowman, J. Derrick, Model checking stochastic automata, *ACM Transactions on Computational Logic*, 4(4), 452–492, 2003.
6. C. Baier, B. R. Haverkort, H. Hermanns, J.-P. Katoen, Model-checking algorithms for continuous-time markov chains, *IEEE Transactions on Software Engineering*, 29(6), 524–541, 2003.
7. D. L. Dill, Timing assumptions and verification of finite-state concurrent systems, *Automatic Verification Methods for Finite State Systems*, LNCS 407, 197–212, 1989
8. M. E. Dyer, A. M. Frieze, On the complexity of computing the volume of a polyhedron, *SIAM Journal on Computing*, 17(5), 967–974, 1988
9. E. Gunter, D. Peled, Path Exploration Tool, *5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, LNCS 1579, 405–419, 1999
10. J.-P. Katoen, P. R. D'Argenio, General distributions in process algebra, *Lectures on formal methods and performance analysis: first EEF/Euro summer school on trends in computer science*, LNCS 2090, 375–430, 2001.
11. M. Kwiatkowska, G. Norman, R. Segala, J. Sproston, Verifying Quantitative Properties of Continuous Probabilistic Timed Automata, *11th International Conference on Concurrency Theory*, LNCS 1877, 123–137, 2000
12. M. Kwiatkowska, G. Norman, R. Segala, J. Sproston, Automatic Verification of Real-time Systems with Discrete Probability Distributions, *Theoretical Computer Science*, 282, 101–150, 2002
13. L. Lovász, S. Vempala, Simulated Annealing in Convex Bodies and an $O^*(n^4)$ Volume Algorithm, *44th Annual IEEE Symposium on Foundations of Computer Science*, 650–659, 2003
14. G. G. Infante-López, H. Hermanns, J.-P. Katoen, Beyond memoryless distributions: Model checking semi-markov chains, *Process Algebra and Probabilistic Methods, Performance Modeling and Verification*, LNCS 2165, 57–70, 2001.
15. Z. Manna, A. Pnueli, How to cook a temporal proof system for your pet language, *Proceedings of the ACM Symposium on Principles on Programming Languages*, 141–151, 1983
16. D. Peled, H. Qu, Enforcing concurrent temporal behaviors, *Electronic Notes in Theoretical Computer Science*, 113, 65–83, 2005
17. T. C. Ruys, R. Langerak, J.-P. Katoen, D. Latella, M. Massink, First passage time analysis of stochastic process algebra using partial orders. *the* 7^{th} *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, LNCS 2031, 220–235, 2001.
18. M. Schechter, Integration Over a Polyhedron: an Application of the Fourier-Motzkin Elimination Method, *The American Mathematical Monthly*, 105(3), 246–251, 1998

19. R. Segala, Modelling and Verification of Randomized Distributed Real-Time Systems, Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1995
20. J. Sproston, Model checking for probabilistic timed systems, *Validation of Stochastic Systems*, LNCS 2925, 189–229, 2004.
21. H. L. S. Younes, R. G. Simmons, Solving Generalized Semi-Markov Decision Processes using Continuous Phase-Type Distributions, *Proc. Nineteenth National Conference on Artificial Intelligence*, AAAI Press, 742–748, 2004

Time Unbalanced Partial Order[*]

Doron Peled and Hongyang Qu

Department of Computer Science,
University of Warwick,
Coventry CV4 7AL, UK

Abstract. Calculating the precondition of a particular partial-ordered set of events is often necessary in software testing, such as for generating test cases. Things become even more complicated when the execution time is added to the picture. If the execution time of two processes along a partial order does not match each other, the precondition of the partial order is false and then the partial order is identified as time unbalanced partial order. We present its formal definition and an algorithm to distinguish it. Then we suggest a method to fill the gap of the execution time of participating processes. This method can also be adopted to simplify the calculation of the minimal and maximal bounds of a time parameter.

1 Introduction

Software testing has been applied broadly as a common technique for enhancing software quality. A general way to test software is to monitor system outputs by providing some initial inputs or path conditions (the necessary relation between program variables for executing the related path). The initial inputs or conditions can be designed according to a set of test criteria, such as branch coverage, either manually or automatically. In many cases the initial condition can be calculated along a specified path backwards. The latter scenario often occurs when testers want to focus on a suspicious path.

Specifying a path and calculating its precondition in an untimed system has been implemented in Path Exploration Tool (PET) [5]. The methodology to compute the precondition of a specified path in a timed system has been proposed in [3] and implemented as an extension to PET. Time constraints in timed systems make the calculation of precondition in such a system much more complicated than in an untimed system.

Moreover, a paradoxical problem is often generated by time constraints when testers specify a path in a timed system. In [3], a timed system is modeled by a transition system, which is translated into extended timed automata. Generally speaking, a path in a timed system represented by timed automata is composed of a sequence of actions and time progress transitions. However, an untimed system only has actions, no time progress transitions, which means a process in

[*] This research was partially supported by Subcontract UTA03-031 to The University of Warwick under University of Texas at Austin's prime National Science Foundation Grant #CCR-0205483.

W. Grieskamp and C. Weise (Eds.): FATES 2005, LNCS 3997, pp. 152–169, 2006.

an untimed system can stay at any location for any long time no matter how long other processes has been running. This characteristics benefits testers such that they can specify any length of a path without worrying about obtaining an unexpected precondition. However, they must be careful when specifying a path in a timed system. A problem occurs with respect to time progress transitions. If the execution time of all actions belonging to one process in the path is shorter than that of actions belonging to another process, then the path condition could be *false*. The reason is that the first process is required by time constraints to execute additional actions on top of those appearing in the path after it has finished executing its actions in the path but before the second process has finished execution. The *false* path precondition could confuse testers in the sense that they would believe that some sequence of actions cannot be executed and draw a wrong conclusion about the path. In addition, this problem can be extended to partial order naturally.

In the rest of this paper, we briefly introduce transition systems, extended timed automata and calculating partial order precondition in Section 2 to provide the necessary background knowledge. Then in Section 3 we use an example to explain the problem and afterwards, the formal description is given by the definition of time unbalanced partial order. In Section 4, a method is proposed to remedy this problem and a kind of its application is identified, which is helpful for testers in some circumstances. Finally Section 5 concludes this paper.

2 Modelling Timed Systems

A timed system could be a physical system, such as the train-gate controller (e.g., [1]), or a software system, such as a communication protocol. In this paper, we only consider the latter.

2.1 Transition System

A transition system (TS) contains a set of concurrent processes. A process is represented by a directed graph $G = \langle V, E \rangle$, where V is the set of nodes and E is the set of directed edges. A node is a control location and an edge is a transition starting at its source location and pointing to its target location. An edge, which is shown in Figure 1, is composed of an enabling condition en and a transformation t. Each of them is associated with a pair of time bounds. Let l and u be the lower bound and the upper bound of the enabling condition and L and U be the lower bound and the upper bound of the transformation.

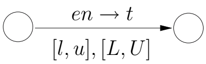


Fig. 1. The edge

When the process resides at the source location of a transition and its enabling condition is evaluated to *true* continuously for more than l time, the transformation has a chance to be triggered. The transformation must be started before the condition holds continuously for u time unless another transition is executed and falsifies the enabling condition before the duration for which the enabling condition holds continuously reaches u time. Once the transformation is triggered, it must be finished after L time and before U time.

2.2 Extended Timed Automata

An extended timed automaton (ETA) is a timed automaton [1] enhanced with program variables. Its formal definition is a tuple $\langle V, \mathcal{C}, \Psi, \Phi, S, S^0, \Sigma, \mathcal{T}, E\rangle$ where

- V is a finite set of program variables.
- $\mathcal{C}$ is a finite set of dense time clocks.
- Ψ is a finite set of assertions defined over V.
- Φ is a finite set of assertions over clocks. Every element $\phi \in \Phi$ is conjunction of constraints of the form $x\ rl\ c$, where x is a clock, rl is a relation from $\{<, >, \geq, \leq, =\}$ and c is a constant.
- S is a finite set of states. Each state $s \in S$ has a state invariant $I(s) = I(s)_\Psi \wedge I(s)_\Phi$, where $I(s)_\Psi \in \Psi$ and $I(s)_\Phi \in \Phi$.
- $S^0 \subseteq S$ is a set of initial states.
- Σ is a finite set of labels.
- $\mathcal{T}$ is a finite set of transformations over V. A transformation in $\mathcal{T}$ represents a set of multiple assignments. Each assignment assigns a new value to a program variable.
- E is a finite set of edges (switches). An edge $\langle s, \sigma, \psi \wedge \phi, t, 2^c, s'\rangle$ starts at *source* state s and points to *target* state s' and has multiple labels $\sigma \subseteq \Sigma$, an enabling condition $\psi \wedge \phi$ ($\psi \in \Psi, \phi \in \Phi$), a transformation $t \in \mathcal{T}$ and a set of clocks $2^c \subseteq \mathcal{C}$ reset by the edge.

Multiple labels in an edge allow the edge to be synchronized with multiple edges. The definition of an execution of ETAs is similar to that of timed automata. Thus it is omitted here and we only consider nonzeno executions.

2.3 Translation from TS to ETAs

Every transition system will be translated into extended timed automata in the following way [3]. Any location in a process is said to be the *neighborhood* of the transitions that must start at that location. The enabledness of each transition depends on the location counter, as well as an assertion over the program variables. For a neighborhood with n transitions, we construct 2^n *enabledness* states, one for each combination of enabledness conditions truth value. An *internal* edge from one such state to another corresponds to such a combination (through progress in other processes). Each such transition α_j has its own local clock x_j. Different transitions may have the same local clocks, if they do not participate in the same process or the same neighborhood. An edge that corresponds to such an assertion becoming true also resets x_j ($x_j := 0$) for measuring the amount of time that the transition is enabled. If an edge corresponds also to some assertion becoming false, we do not reset the local clock for that transition. On a given state where a transition α_j is enabled, we have the conjunct $x_j < u_j$ as part of the state invariant, disallowing a transition to stay in that state more than its upper limit u_j.

We also have an additional *intermediate* state per each transition in the neighborhood, from which the transformation associated with the selected transition is performed. From any enabledness state, as described in the previous paragraph, in which the enabledness c_j of α_j holds, we add a *decision* edge with the condition $x_j \geq l_j$, allowing the execution of α_j only after at least l_j (the lower bound for α_j) to be continuously enabled since it became enabled. On that edge, we also reset the clock x_j to measure now the time it takes to execute the transformation. On the target state for that edge, i.e., the intermediate state, we put the condition $x_j < U_j$, not allowing the transformation to be delayed more than U_j time, and add a *transformation* edge labeled with $x_j \geq L_j$ to any of the enabledness states representing the following location. Again, this is done according to the above construction, there can be multiple such states, for the successor neighborhood, and we need to reset the appropriate clocks.

2.4 Generating DAGs

After we model a timed system by a transition system and translate it into extended timed automata, the product of these automata is generated in a standard way as in [1]. We assume any transition can reference at most one shared variables. A shared variable cannot be accessed synchronously by the transformations of two or more transitions. When multiple transitions compete each other to access a shared variable, only one access is granted and other transitions are blocked. These blocked transitions are activated after the granted access is finished and their enabling conditions are evaluated from beginning. The internal edges are used to synchronize transitions on accessing shared variables and therefore are merged to the decision edges and the transformation edges in the product. When a tester specifies a sequence of transitions that a system needs to execute orderly, he specifies a single path; however, due to independence in the order of concurrent events, a partial order [7, 8], which describes the *essential* execution order between transitions, is more appropriate, as transitions belonging to the different processes and are not competing with each other for accessing a mutual variable are not necessarily ordered as in the given sequence. This partial order is represented as a formula over a finite set of *actions* $Act = A_c \cup A_f$, where the actions A_c represent the decision edges, and the actions A_f represent the transformation edges. Thus, a transition α is split into two components, $\alpha_c \in A_c$ and $\alpha_f \in A_f$. The essential order imposes sequencing all the actions in the same process, and pairs of actions that use or set a shared variable. In the latter case, the decision edge β_c of the latter transition succeed the transformation edge α_f of the earlier transition. However, other transitions can interleave in various ways (e.g., $\alpha_c \prec \gamma_c \prec \gamma_f \prec \alpha_f$). This order relation $\prec$ corresponds to a partial (irreflexive, asymmetric, transitive) order over *Act*. Let $\mathcal{A}_{\prec}$ be an automaton that recognizes the untimed language of words that are linearizations of $\prec$.

We label the edges in the product according to *Act* and then synchronize the partial order automaton with the product automaton to form a directed acyclic graph (DAG). The synchronization is done as commonly labeled transitions of different processes are executed together. A path from the root node to a leaf node in the DAG is one of equivalent paths represented by the partial order. In an untimed systems, the precondition of executing any path represented by the partial order is the same. When adding time, some paths are ruled out by the time constraints.

2.5 Calculating Path Precondition

Now calculating the partial order precondition is performed on the DAG. Each leaf node is associated with a first order predicate $\varphi^0 = \mathit{true}$ and a DBM [4] D^0, which is used to represent a time zone. A DBM is a $(m+1) \times (m+1)$ matrix[1] where m is the number of local clocks of all processes. Each element $D_{i,j}$ of a DBM D is an upper bound of the difference of two clocks x_i and x_j, i.e.,

$$x_i - x_j \leq D_{i,j}. \tag{1}$$

We use $x_1, \cdots, x_m$ to represent local clocks. x_0 is a special clock whose value is always 0. Therefore, $D_{i,0}$ $(i > 0)$, the upper bound of $x_i - x_0$, is the upper bound of clock x_i; $D_{0,j}$ $(j > 0)$, the upper bound of $x_0 - x_j$, is the negative form of the lower bound of clock x_j. D^0 represents the time zone after execution of the last transition of any path. The entry $D^0_{i,j}$ is set to $(0, \leq)$ for $i = 0 \vee i = j$, or $(\infty, <)$ otherwise, since we do not know the exact value of each clock before paths are executed. The exact clock value ranges can be computed during backward calculation. Note that we need to distinguish non-strict inequalities from strict inequalities in the formula (1).

We now progress backwards and "relativise" out path condition over the edge we traverse as follows: since each edge has a transformation $v := e$ and an assertion *en* as its enabling condition over program variables, starting at leaf nodes, φ (which is φ^0) is updated to $\varphi[v/e] \wedge en$ ($\varphi[v/e]$ means replacing each occurrence of v in φ by e) each time when we follow an edge backwards from the target node s to the source node s'. At the same time, D (which is D^0) is updated to D' for the backward reachability analysis, according to the following formula [9]

$$D' = ((([\lambda := 0]D) \wedge I(s')^c \wedge \psi^c) \Downarrow) \wedge I(s')^c,$$

where $I(s')^c$ is the assertion over clocks in the state invariant of the source node, ψ^c is the assertion over clocks attached to the edge, λ is the set of clocks reset by the edge, "$\wedge$" is the intersection of two DBMs, "$[\lambda := 0]D$" is the reset operation on DBMs and "$\Downarrow$" is the time predecessor operation on DBMs. When two edges have the same source node, the pairs of the relativised condition and the DBM of

[1] Here we omit the global clock in [3].

the form $\langle \varphi, D \rangle$ from these edges are disjointed together at the source node. For a non-leaf node, it may have more than one pair of $\langle \varphi, D \rangle$, each of which is the precondition of a different path from itself to a leaf node. Thus the precondition of the given path is the disjunction of all pairs of $\langle \varphi, D \rangle$ associated with the root nodes. In the rest of this paper, we only give the disjunction of predicates as the path precondition, since the main purpose to use DBMs is to calculate the reachability.

3 The Paradoxical Problem

3.1 An Example

Let us consider the following example in Figure 2. A timed system is composed of two processes represented by program 1 and 2. $x, cont, y, f1$ and $f2$ are local variables and s and z are shared variables.

```
Program 1                               Program 2
      begin                                   begin
p1:       $x := 1$;                     q1:       while (true) do
p2:       while (true) do                         begin
          begin                         q2:           $wait(s > 0, -1, f2)$;
p3:           $s := x$;                 q3:           $y := s$;
p4:           $wait(z > 0, l, f1)$;     q4:           $s := 0$;
p5:           if ($f1 = 0$) then        q5:           $z := 1$
              begin                               end
p6:               $z := 0$;                   end.
p7:               $x := x + 1$
              end
              else
p8:               $cont := 1$
          end
      end.
```

Fig. 2. A simple concurrent real-time system

The semantics of the *wait* statement is described as follows: It has three parameters. The first one is the condition it waits for to become true. The second is the time limit and the third is a variable. A timer is started when the statement is executed. If the time limit is reached before the condition becomes true, a timeout is triggered and the variable is set to 1. If the condition becomes true before timeout, the variable is set to 0 and the timer is stopped. It is not appropriate to detect whether the *wait* statement timeouts or not by testing the condition because the condition may not be accessed after *wait* statement. That the time limit is -1 means the process can wait for the condition forever without timeout. In this example, l is a parameter which is the time limit of a timer. (Note that l can be substituted to a constant as well.) If the condition $z > 0$ is not detected before the time limit is reached, a timeout would be triggered. The value range of l is computed automatically during precondition calculation and given by a predicate in the precondition. The ranges for program variables are given in the precondition as well.

The flow chart for Program 1 is shown on the left part of Figure 3, and the one for Program 2 on the right. Each flow chart node corresponds to an assignment statement, or a condition of a conditional or loop statement. A flow chart node and its corresponding statement or condition has the same label.

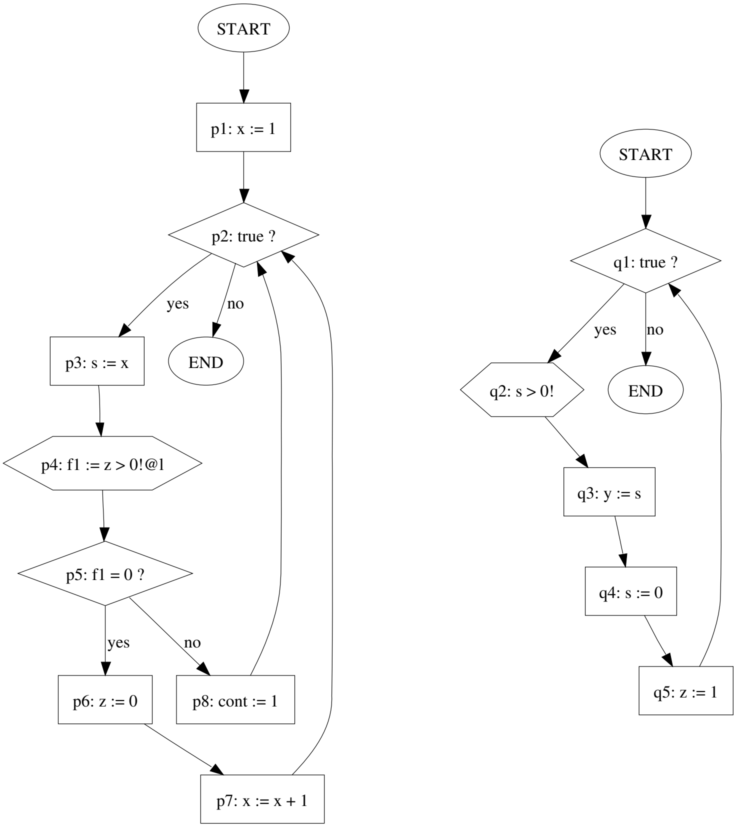


Fig. 3. The flow charts

The corresponding transition system of Program 1 and 2 is shown in Figure 4. Each node in the figure is a location and each edge represents a transition. The bounds are shown to the right of enabling conditions and transformations. An assignment is translated into a transition with the enabling condition *true*. A branch node is translated into two transitions with null transformation (denoted as No_op). (Note that there might be another way to translate a branch node. But every method has its pros and cons.) The time bounds in Figure 4 are chosen

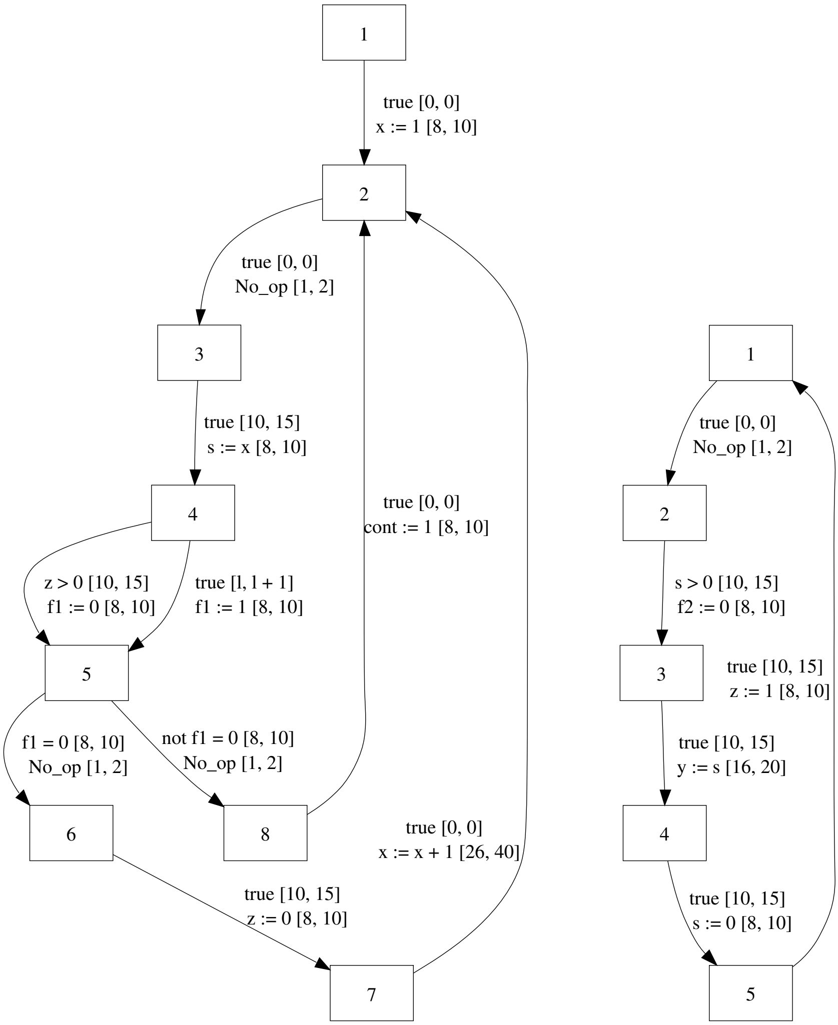


Fig. 4. Program 1 (left) and 2 (right)

as follows: the bound for condition *true* in timeout transition is $[l, l+1]$ and the bound for condition *true* in other transitions is $[0, 0]$; the bound for evaluating the nontautological enabling condition of a transition which does not access any shared variable is $[8, 10]$; the bound is $[10, 15]$ if the transition accesses a shared variable; assigning an instant value to a variable is bounded by $[8, 10]$; addition operation has the bounds $[10, 20]$ and No_op has the bounds $[1, 2]$.

3.2 The Problem

Figure 5 shows two partial order executions. They are depicted by flow chart nodes. An edge in a partial order means the source transition of the edge must

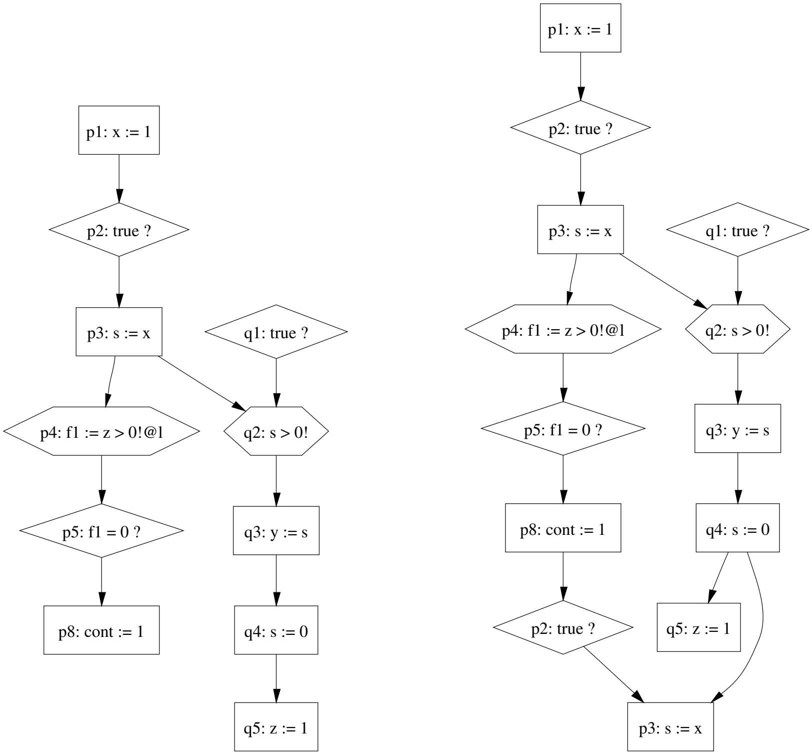


Fig. 5. Partial order 1 (left) and 2 (right)

be executed before the target transition being executed. For example, the edge from $p3$ to $q2$ means the statement $p3$ must be executed before the statement $q2$ is executed.

The precondition[2] for the partial order 1 is

$$z \leq 0 \wedge s \leq 0 \wedge 48 \leq l \leq 84$$

and that for partial order 2 is

$$(z \leq 0 \wedge s \leq 0 \wedge 13 \leq l \leq 65) \vee (z \geq 1 \wedge s \leq 0 \wedge 13 \leq l \leq 14).$$

We denote that a partial order is *executable* if its precondition is not *false*; otherwise, the partial order is *unexecutable.* Obviously, if we replace l by 40 in Program 1, partial order 1 is unexecutable, while partial order 2 is executable. The difference between these two partial orders is that Process 1 executes a few more statements on partial order 2 than on partial order 1, while Process 2 executes the same statements on both partial orders. Both partial orders are

[2] The Omega library, which we used to simplify Presburger formula, operates on integers such that it simplifies $l < n$ to $l \leq (n-1)$.

executable in untimed systems because the extra statements on partial order 2 do not change the precondition on program variables. This case reveals that time constraints could distinguish two partial orders that cannot be distinguished in untimed systems.

Now we explain how time constraints affect preconditions in this case. In both partial orders, statement $q2$ is first enabled at the same time or after statement $p4$ becomes enabled because $q2$ can only be enabled after both $q1$ and $p3$ are executed, while $p4$ could be enabled after $p3$ is executed but before $q1$ is executed. The sequences $\langle p4, p5, p8\rangle$ and $\langle q2, q3, q4, q5\rangle$ on the partial order 1 are local to Process 1 and 2 respectively since there is no dependence between them. (Note that $p4$ on $\langle p4, p5, p8\rangle$ behaves as the transition $true \rightarrow f1 := 1$, which can be seen in Figure 4.) Therefore, the execution of $\langle p4, p5, p8\rangle$ is independent of the execution of $\langle q2, q3, q4, q5\rangle$. To be executed, each sequence is translated into a sequence of automaton states and edges. Let a be the last automaton state of the sequence $\langle p4, p5, p8\rangle$ after translation, b that of the sequence $\langle q2, q3, q4, q5\rangle$. Since the DAG for partial order 1 is very complicated, Figure 6 only shows a part of it and the initial node. The dot lines in the figure denote the part of the DAG being omitted. Each DAG node is a compound state which contains a state in Process 1 and a state in Process 2. Only a and b are labeled in the figure. A node labeled as only a or b means that this node contains a or b but not both. In the DAG, there are a group of interleaving paths, each of which represents an execution schedule of these two sequences and ends with a compound state containing both a and b. By adding all lower bounds and the upper bounds along these two sequences, it is easy to see that the maximum execution time of the sequence $\langle p4, p5, p8\rangle$ is 73, while the minimum execution time of the sequence $\langle q2, q3, q4, q5\rangle$ is 80, under the condition $l = 40$. This means that during the execution of any interleaving path, the system would definitely reach some states Q (which are DAG nodes labeled "a" in Figure 6) that contain a, but not b. (Computation on DBMs shows that a state containing only b is unreachable.) For each state after $\mathcal{Q}$, Process 1 will stay at a until Process 2 reaches b. However, at state a, the statement $p2$ is enabled and thus there is a time constraint in the state invariant of a that requires Process 1 to execute $p2$ and then leave a before Process 2 reaches b. In other words, the given partial order requires that the system reaches a state which contains a and b, but the times constraints make this state unreachable. Starting from an unreachable state, the backward calculation of path precondition gives $false$ as the result.

On the other hand, let a' be the last automaton state of the sequence $\langle p4, p5, p8, p2, p3\rangle$ on the partial order 2. b is still the last state of $\langle q2, q3, q4, q5\rangle$. By applying the algorithm in the next section, we know that the maximum execution time of $\langle p4, p5, p8, p2, p3\rangle$ is 92. For the partial order 2, therefore, the system can reach the final state which contains both a' and b, i.e., the system could enter a state after which either Process 1 stays at a' until Process 2 reaches b or Process 2 stays at b until Process 1 reaches a'. Thus the precondition of partial order 2 is not $false$.

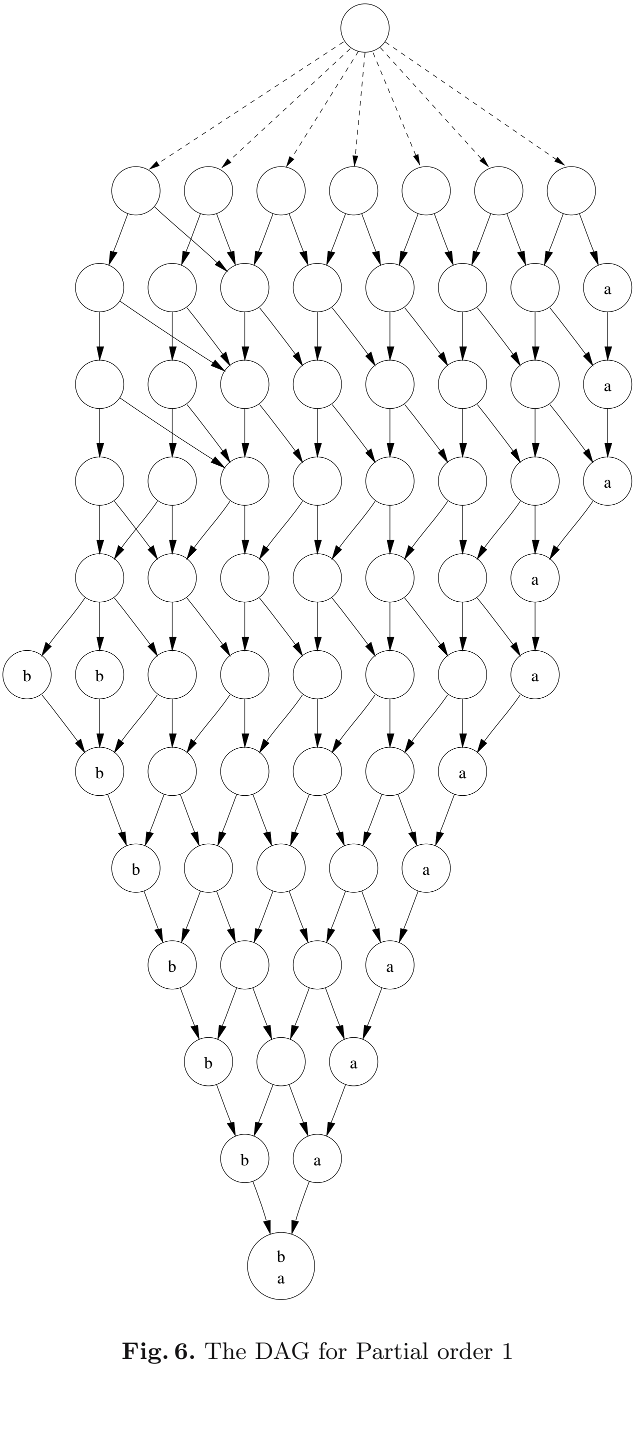


Fig. 6. The DAG for Partial order 1

3.3 The Definition

The problem above can be formally interpreted by *time unbalanced partial order*, which is defined as follows. Let ρ be a partial order in a system composed of n ($n > 1$) processes $P_1, \ldots, P_n$. Let $\rho^i = \alpha_0^i \alpha_1^i \ldots \alpha_{m_i}^i$ be the *projected path* which is the projection of ρ onto P_i. Each α_j^i ($0 \leq j \leq m_i$) is a statement of P_i. For example, in the partial order 1 in Figure 5, there are two projected paths: $\rho^1 = \langle p1, p2, p3, p4, p5, p8 \rangle$ and $\rho^2 = \langle q1, q2, q3, q4, q5 \rangle$. Another example is the partial order in Figure 7. This partial order is defined over Program 1 and 2 in Figure 2 as well. It is similar as the partial order 2 in Figure 5 except that the partial order 2 contains the statement $q5$ in Program 2, while this partial order does not. The edge from $q4$ to the second appearance of $p3$ in Figure 7 means that $q4$ must be fired earlier than the second appearance of $p3$. This partial order has two projected paths: $\rho^3 = \langle p1, p2, p3, p4, p5, p8, p2, p3 \rangle$ and $\rho^4 = \langle q1, q2, q3, q4 \rangle$.

Let $T(\rho^i)$ be the execution time of ρ^i. For any two sequences ρ^i and ρ^j ($1 \leq i, j \leq n$ and $i \neq j$), $T(\rho^i) < T(\rho^j)$ if ρ^i and ρ^j satisfy one of the following conditions:

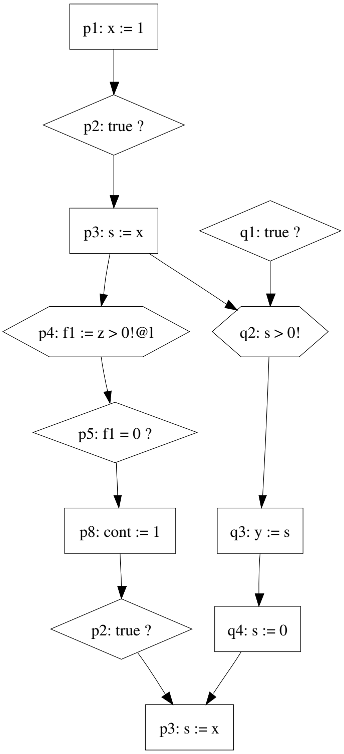


Fig. 7. An example partial order

1. The last statement of ρ^j, which is $\alpha^j_{m_j}$, *depends* on $\alpha^i_{m_i}$, which is the last statement of ρ^i. That is, $\alpha^i_{m_i}$ is required by the partial order to be fired before $\alpha^j_{m_j}$ is fired. For example, the second appearance of $p3$ in ρ^3, which is the last statement of ρ^3, depends on $q4$, the last statement of ρ^4, because of the edge from $q4$ to the second appearance of $p3$ in Figure 7.
2. If $\alpha^i_{m_i}$ and $\alpha^j_{m_j}$ do not depend on each other, the maximum execution time of ρ^i is smaller than the minimum execution time of ρ^j. For example, the maximal execution time of ρ^1 in the partial order 1 is 110 and the minimal execution time of ρ^2 is 117.

The projected paths ρ^i and ρ^j form a *time unbalanced projected path pair*, where ρ^i is the *short* projected path and ρ^j is the *long* one. For two projected paths ρ^k and ρ^l, $T(\rho^k) \approx T(\rho^l)$ if $T(\rho^k) \nless T(\rho^l)$ and $T(\rho^l) \nless T(\rho^k)$. A partial order ρ is a time unbalanced partial order if there exist some time unbalanced projected path pairs in ρ. For example, the partial order 1 is a time unbalanced partial order because ρ^1 and ρ^2 satisfy the second condition and then construct a time unbalanced projected path pair. The partial order in Figure 7 is time unbalanced as well since ρ^3 and ρ^4 satisfy the first condition.

Time unbalanced partial order is very common on timed automata. For example, a linear partial order, where there is only one root statement, one leaf statement and one path from the root to the leaf, is a time unbalanced partial order. We denote a partial order ρ_1 is a *prefix* of another partial order ρ_2 if any equivalent path represented by ρ_1 is a prefix of an equivalent path represented by ρ_2. ρ_2 is *longer* than ρ_1. A time unbalanced partial order ρ whose precondition is *false* is *extendable* if it is a prefix of a longer path ρ' whose precondition is not *false* and the projected paths which have the longest execution time in both ρ and ρ' are the same. The second requirement ensures that only the projected paths which have short execution time are extended.

When calculating the precondition of a time unbalanced partial order, there is a danger that the partial order precondition could be *false*, but the precondition of the corresponding untimed partial order is not *false*, which is the case occurred on the partial order 1. However, not all time unbalanced partial orders have *false* as their preconditions. Although an unbalanced pair of projected paths could lead the system entering a state from which the final state is unreachable, whether the system enters such a state invariantly or not depends on not only time unbalanced projected path pairs, but also their successive statements. Let a be the last automaton state of the short projected path of a pair. Let τ be any transition starting at a and u be the upper bound for its enabling condition. If τ is continuously enabled longer than u before the long projected path ends, the partial order precondition is *false* because u forces τ to be fired (but τ does not appear on the partial order). On the other hand, if a partial order is not extendable, i.e., it is unexecutable due to some time constraints other than the gap between execution times of two processes in an unbalanced projected path pair, its precondition should be *false*, though the corresponding untimed partial order could have non-false precondition.

Computing each projected path's running time is as follows. Let n_i be a node in the partial order. Let $MAX(n_i)$ and $MIN(n_i)$ be the maximum execution time and the minimum execution time of a projected path which contains n_i after n_i is executed, and $max(n_i)$ and $min(n_i)$ be the maximum execution time and the minimum execution time of n_i. $max(n_i)$ and $min(n_i)$ are obtained from time bounds. If n_i is a root node, $MAX(n_i) = max(n_i)$ and $MIN(n_i) = min(n_i)$. Otherwise, assume n_i has k predecessors $n_j, \ldots, n_{j+k-1}$:

$$MAX(n_i) = \max\{MAX(n_j), \ldots, MAX(n_{j+k-1})\} + max(n_i)$$

$$MIN(n_i) = \max\{MIN(n_j), \ldots, MIN(n_{j+k-1})\} + min(n_i).$$

A projected path's maximum and minimum execution time is the MAX and the MIN of its last node respectively.

The algorithm above only gives estimated execution time, not accurate one, because the real execution time may also depends on the transitions that do not appear in the partial order. Therefore, this algorithm cannot be used to substitute the one in [3] to calculate the precondition of a partial order. But it is enough to tell us whether a particular partial order is time unbalanced or not.

The time unbalanced partial order also reveals why the lower bound for l on the partial order 1 is 48. It is natural to think that Process 1 would timeout even when $l < 48$. But 48 ensures that the partial order 1 is not time unbalanced. For the same reason, the lower bound of l is 13 on the partial order 2.

4 A Remedy to the Problem

Though it is helpful to indicate that a partial order is time unbalanced in addition to tell users the partial order precondition, it is better to let the users learn more about the partial order than that the partial order precondition is *false*. The basic reason that a partial order is time unbalanced is that there is a gap between the executions of two processes in an unbalanced pair. If we allow the process which has the short projected path to continue running, i.e., leave its last state on the projected path, the gap could be filled and the unbalanced pair is changed to a balanced one. But the successive transitions should not alter the interprocess partial order relations, i.e., the successive transitions could not change the value of shared variables used by other processes.

4.1 The Remedy Method

Since the successive statements of a projected path are different on different partial orders and in different systems, and the execution time gap between a pair of unbalanced projected paths is different among partial orders and systems, it could be difficult to explore all possibilities to allow a process executing successive statements, in particular, in the case of nondeterminism. A simple method to allow processes to execute extra transitions and not change the partial order is for each projected path, to remove the time constraints in state invariant of the last node during constructing the DAG. This method allows a process to execute any successive statements without care of which statements are chosen and how many statements need be executed to fill the time gap. After being applied this method, the precondition of the partial order 1 under $l = 40$ is $z \leq 0 \wedge s \leq 0$, which is what we expect. The new precondition is noted as the *underlying precondition.*

However, this method only remedies the gap between execution times among processes for a time unbalanced partial order so that it cannot be used carelessly. It cannot be applied to partial orders that are not extendable, since for a time unbalanced and unextendable partial order, it is not the gap between execution times of two processes that makes the precondition of the partial order become *false*. Thus, it would calculate a wrong precondition if we apply this methods to an unextendable partial order. For example, consider the following programs. We choose the time bounds according the criteria in Section 3.1, i.e., the upper bounds for the enabling condition and the transformation of transition Q1 are 15 and 10, respectively. The time limit of the timer in statement P1 is 50 according to the semantics of the *wait* statement. The partial order in the left path of Figure 9 is time unbalanced and cannot be extended to an executable balanced partial order. It is also unexecutable since it can only be extended to the partial order in the right part of the figure, which is unexecutable because the condition $x > 0$ is satisfied before time progresses up to 50 units and the variable $f1$ is set to 0.

```
         Program 3                      Program 4
         begin                          begin
P1:        wait(x > 0, 50, f1);    Q1:    x := 1
P2:        if (f1 = 0) then             end.
P3:          y := 0
           else
p4:          y := 1;
         end.
```

Fig. 8. A negative example

Until now, the methodology assumes all processes on the partial order start at the same time. If not all of them can start at the same time, the precondition could be *false*. It is a reasonable assumption in many cases, especially when the partial order contains the first transition of each participant process. In fact, a partial order does not necessarily start at the first transition of every process. The methodology in [3] also does not have such requirement. This gives testers great convenience because sometimes it is very difficult to know the whole sequence of transitions from the beginning that causes an error, but it is easy to know a part of sequence which causes the error. Testers are able to concentrate on this part of sequence, without worrying about the whole sequence. However, this convenience does not cost nothing. The precondition of

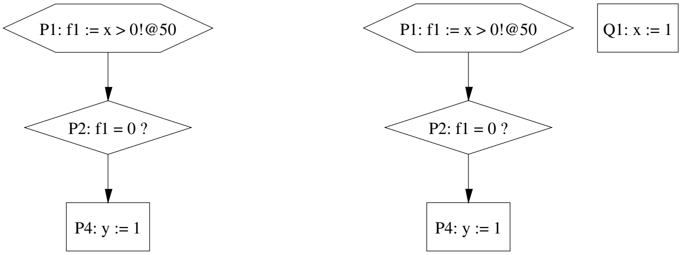


Fig. 9. Two unextendable partial orders

a partial order could be $false$ because one process starts later than another. Let $\overline{a}_i$ be the first state of Process i on the partial order. That is, we assume the system begins to run from a compound state containing all $\overline{a}_i$ states, while this state is unreachable from the initial state. But it is often difficult to know which starting state is a reachable state from the initial state. In such cases, the concepts introduced previously can be extended easily to handle this situation. Removing the time constraint in the state invariant of every $\overline{a}_i$ allows that each process starts at a different time and then we can calculate the underlying precondition.

4.2 An Application of the Method

This method has another usage that it also releases the users from specifying a set of partial orders to obtain a complete precondition. One partial order could give them a satisfying answer. For example, we may be interested in the maximal upper bound and minimal lower bound of l between that timeout may occur. When $z < 0$, the conditions of l on the partial order 1 and the partial order 2 are $48 \leq l \leq 84$ and $13 \leq l \leq 65$. It is obvious that 48 is not the minimal lower bound since it is grater than 13 and 65 is not the maximal upper bound since $65 < 84$. Furthermore, we cannot draw the conclusion that 13 is the minimal lower bound and 84 the maximal upper bound because of an obvious fact that timeout would occur when $l = 0$. We have to check more partial orders, such as the partial order 3 and the partial order 4 in Figure 10, in order to obtain the proper bounds. These partial orders are similar as the partial order 1 in Figure 5, except that Process 2 has fewer transitions involved in them. The conditions of l on the partial order 3 and 4 are

$$(z \leq 0 \wedge s \leq 0 \wedge 12 \leq l \leq 49) \vee (z \geq 1 \wedge s \leq 0 \wedge 12 \leq l \leq 14)$$

and

$$z \leq 0 \wedge s \leq 0 \wedge 30 \leq l \leq 74$$

respectively. These conditions are still not what we expect. This example demonstrates it is not easy to get the proper bounds for time parameters.

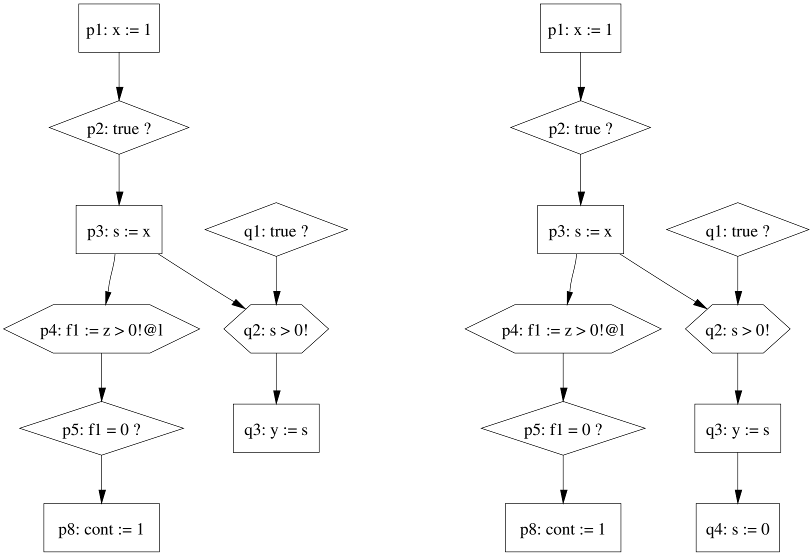


Fig. 10. Partial order 3 (left) and 4 (right)

However, we obtain the promising bounds when we apply the remedy method to the partial order 1. The underlying precondition of the partial order 1 is

$$(z \leq 0 \wedge s \leq 0 \wedge 0 \leq l \leq 92) \vee (z \geq 1 \wedge s \leq 0 \wedge 0 \leq l \leq 14).$$

0 is the lower bound we expect. 92 is believed to be the correct upper bound after we check the time constraints of the partial order 1 carefully. Furthermore, the new condition also tell us under $0 \leq l \leq 14$, timeout can occur even when $z > 0$. This coincides with the time constraints of statement $p4$. The reason why we use this method on the partial order 1 is that sequence $\langle q2, q3, q4, q5\rangle$ is the maximal one before condition $z > 0$ holds. The minimal lower bound is obtained by executing $\langle q2, q3, q4, q5\rangle$ after $p4$ is executed and the maximal upper bound is obtained by executing $p4$ after $\langle q2, q3, q4, q5\rangle$ is executed. Using the partial order 2 generates the same bounds but it takes a much longer time to calculate the precondition for the partial order 2 than for the partial order 1.

In the literature, there are several papers studying parametric model checking [2, 6]. Constraints on parameters can be deduced by parametric model checking as well. The advantage of parametric model checking is that it is an automatic technique to obtain the constraints with respect to a property. But it might search a very large state space and visit irrelevant states. In contrast, our method is not automatic, but requires the users to specify the partial order to compute the constraints. In other words, it involves human intelligence. Hence, the advantage of our method is that it searches far smaller state space than parametric model checking does if users provide an appropriate partial order. Of course, choosing an appropriate partial order may not be achieved easily for a complicated system. Therefore, our method is more suitable for advanced users than for inexperienced users.

5 Conclusion

In this paper we identified time unbalanced partial order in timed systems and gave its definition. This phenomenon is caused by unbalanced projected path pairs and does not exist in untimed systems. The gap between a pair of unbalanced projected paths might force the system to enter a state from which the final state required by the partial order is unreachable. Due to the existence of time unbalanced partial order, testers may not easily distinguish an extendable partial order from an unextendable one, but these two kinds need different treatments. We proposed an algorithm to check whether a partial order is time unbalanced or not and a remedy method to transform an unbalanced partial order into a balanced one so that we can calculate its underlying precondition. We also applied the remedy method to simplify calculating the maximal and the minimal bounds of a time parameter. Generally speaking, on the one hand, time unbalanced partial order can cause troubles to testers so that they must be careful when specifying a partial order; on the other hand, it is helpful to simplify computation in some circumstances.

References

1. R. Alur, Timed Automata, *NATO-ASI 1998 Summer School on Verification of Digital and Hybrid Systems*
2. R. Alur, T. Henzinger, M. Y. Vardi, Parametric Real-time Reasoning, *Proceedings of the 25th ACM Symposium on Theory of Computing*, 592–601, 1993
3. S. Bensalem, D. Peled, H. Qu, S. Tripakis, Automatic Generation of Path Conditions for Concurrent Timed Systems, *1st International Symposium on Leveraging Applications of Formal Methods*, 2004
4. D. L. Dill, Timing assumptions and verification of finite-state concurrent systems, *Automatic Verification Methods for Finite State Systems*, LNCS 407, 1989, 197–212
5. E. L. Gunter, D. Peled, Path Exploration Tool, *5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, LNCS 1579, 405-419, 1999
6. T. Hune, J.M.T. Romijn, M.I.A. Stoelinga, F.W. Vaandrager, Linear parametric model checking of timed automata, *Journal of Logic and Algebraic Programming* 52-53, 183–220, 2002
7. L. Lamport, Time, Clocks, and the Ordering of Events in a Distributed System, *Communications of the ACM*, 21(7), 558–565, 1978
8. D. Peled, H. Qu, Enforcing concurrent temporal behaviors, *Electronic Notes in Theoretical Computer Science*, 113, 65–83, 2005
9. S.Yovine, Model checking timed automata, *Lectures on Embedded Systems*, LNCS 1494, 1998, 114–152

Testing Systems of Concurrent Black-Boxes—An Automata-Theoretic and Decompositional Approach*

Gaoyan Xie** and Zhe Dang***

School of Electrical Engineering and Computer Science,
Washington State University, Pullman, WA 99164, USA
{gxie, zdang}@eecs.wsu.edu

Abstract. The global testing problem studied in this paper is to seek a definite answer to whether a system of concurrent black-boxes has an observable behavior in a given finite (but could be huge) set Bad. We introduce a novel approach to solve the problem that does not require integration testing. Instead, in our approach, the global testing problem is reduced to testing individual black-boxes in the system one by one in some given order. Using an automata-theoretic approach, test sequences for each individual black-box are generated from the system's description as well as the test results of black-boxes prior to this black-box in the given order. In contrast to the conventional compositional/modular verification/testing approaches, our approach is essentially decompositional. Also, our technique is complete, sound, and can be carried out automatically. Our experiment results show that the total number of tests needed to solve the global testing problem is substantially small even for an extremely large Bad.

1 Introduction

Testing a concurrent and component-based system is notoriously difficult[16, 14]. One difficulty comes from the system's nondeterminism and the synchronizations among concurrently running components. Another difficulty lies in the fact that, in a component-based system, its constituent components could be some externally obtained software components (such as COTS products) whose source codes and design details are usually not available. In that case, traditional white-box techniques (like static analysis) are not applicable to analyzing the system. These components can be readily treated as *black-boxes* whose models (both at code level and design level) are unknown. In this paper, we study a testing problem for such a system of concurrent black-boxes.

In our setup, a system of concurrent black-boxes consists of a host system (called the gluer) and a number of black-boxes. Each of the gluer and the black-boxes is called a *unit* (or a component), which is a (possibly nondeterministic and infinite-state) labeled transition system, each of whose labels represents either an observable action or an internal action. All the units in the system run concurrently and synchronize on a number

* The research was supported in part by NSF Grant CCF-0430531.

** Gaoyan Xie is currently affiliated with the University of Massachusetts Dartmouth (gxie@umassd.edu).

*** Corresponding author (zdang@eecs.wsu.edu).

W. Grieskamp and C. Weise (Eds.): FATES 2005, LNCS 3997, pp. 170–186, 2006.

of observable actions. The gluer is a fully specified finite-state unit. For each black-box, however, except for its interface (i.e., the set of its observable actions), everything else is unknown, while its implementation is always available and can be black-box tested. A *global bad behavior* is an observable behavior of the system in a given finite set Bad. Finally, the *global testing problem* studied in this paper is to verify (with a definite answer) that, for the given set Bad, the system does not have a global bad behavior.

A straightforward approach to solve the global testing problem is to perform integration testing over the system as a whole and see if the system exhibits a bad behavior. However, there are fundamental difficulties with this approach. For instance, in some applications [29], integration testing may not be applicable at all. Even when integration testing is possible in some situations, the system itself is often nondeterministic. The combinatorial blow-up on the number of the executions caused by nondeterministic interleavings among the concurrent units in the system generally makes it infeasible to do thorough integration testing, while we are looking for a definite answer to the global testing problem. Due to the same reason, even when one has a way to handle the nondeterminism [30], the size of the given set Bad (which could be very large, e.g., more than 10^{24} in some of our experiments shown later) may also make exhaustive integration testing infeasible.

A less straightforward approach is to combine testing with some formal method. For instance, one can extensively test each black-box alone and try to build [25] a partial model of the black-box from the test results. Then, one can run a formal method like model-checking on the partial system model built from the partial models of the black-boxes to solve the global testing problem. However, this approach is also difficult to implement. For instance, it is hard to choose effective test sequences to build a partial model of a black-box, and it is also hard to know when the tests over a black-box are adequate. Moreover, the partial (and hence approximated) system model might not help us obtain a definite answer to the global testing problem. To avoid the above difficulties, one may also try, using some formal method, to derive an expectation condition over a black-box's behaviors such that: when every black-box behaves as expected, the system guarantees to not have a global bad behavior. Then the expectation conditions can be used to generate test sequence for the black-boxes. However, the interactions among the concurrent black-boxes make it difficult to derive such conditions automatically (see Section 2 for related work on the assume-guarantee style reasoning).

In this paper, we introduce a novel approach (called the "push-in" technique) to solve the problem, which does not entail any integration testing. Instead, in our approach, the global testing problem is reduced to testing individual black-boxes in the system one by one in some given order. Using an automata-theoretic approach, test sequences for each individual black-box are generated from the system's description as well as the test results of black-boxes prior to the black-box in the given order. Suppose that $B_1, \ldots, B_k$ represent the concurrent black-boxes in a system. The first step of our approach is to compute an auxiliary set $\mathcal{A}_1$ of sequences of observable actions for black-boxes $B_1, \ldots, B_k$ and a set $\mathcal{U}_1$ of test sequences for black-box B_1. Then we test the black-box B_1 with test sequences in $\mathcal{U}_1$ and collect all successful test sequences into a surviving set SUV_1. In the second step, from the surviving set SUV_1 and the auxiliary set $\mathcal{A}_1$, we compute the auxiliary set $\mathcal{A}_2$ (for black-boxes $B_2, \ldots, B_k$) and the test sequence set $\mathcal{U}_2$ for black-box B_2. Again, after testing black-box B_2 with test sequences in $\mathcal{U}_2$, we collect all successful testing sequences into a surviving set SUV_2. Subsequent steps follow similarly, and eventually, in the last step (i.e., step k), the global testing problem will be decided from the surviving sets. That is, the system has no global bad behavior iff, for some $1 \leq i \leq k$, the surviving set SUV_i is empty. We also provide a procedure to recover a global bad behavior when the answer to the original problem is "no".

Since the sets (i.e., $\mathcal{U}_i$ and $\mathcal{A}_i$) are provably finite and, in many cases, huge, we use (finite) automata that accept the sets as their symbolic representations, and standard automata operations are used to manipulate these sets. Also, the global testing problem is decomposed into a series of testing problems over each individual black-box in the system. Hence, our approach is an automata-theoretic and decompositional approach. Moreover, the "push-in" technique is both complete and sound, and can be carried out automatically. In particular, we show that the technique is "optimal" in the sense that each test we run over a black-box has the potential to discover a global bad behavior (i.e., we never run useless tests). In general, exhaustive integration testing over a concurrent system is infeasible. However, our experiments show that, using the push-in technique, we can completely solve the global testing problem with a substantially smaller number of tests over the individual black-boxes, even for an extremely large set of Bad (some of our experiments performed only about 10^5 unit tests for a Bad of size more than 10^{24}).

The rest of this paper is organized as follows. In Section 2, previous work related to this paper is discussed. In Section 3, the formal definitions for a system of concurrent black-boxes and its global testing problem are presented. In Section 4, the detail of the push-in technique is shown. In Section 5, a set of experiments are run and the results are analyzed. Finally, Section 6 points out some future work.

2 Related Work

The global testing problem is essentially a verification problem since we are looking for a definite answer. In the area of formal verification, there has been a long history of research on exploiting compositionality in system verification, and a common technique is to follow the "assume-guarantee" reasoning paradigm [20, 27, 19, 7, 2, 9, 8, 3]. However, a successful application of the paradigm depends on the correct assumptions for the components in a system, which are, in general, formulated manually. Several authors suggest solutions to the problem of automated assumption generation [17, 18, 12, 15]. But the solutions require that the source code and/or the finite-state design is available for a unit, which, unfortunately, is not the case in our setup. Although our push-in technique relies on black-box testing instead of an "assume-guarantee" mechanism, it can be extended to a system where a black-box is associated with environmental assumptions.

In the area of software testing, researchers have long recognized the importance of combining formal methods (like model-checking) and testing techniques for system verification. Most work (e.g., [6, 10, 13]) stems from the spirit of specification-based testing, and utilizes model-checkers' capabilities of generating counter-examples from a system's specification to produce test-cases against an implementation. This approach typically works at the unit level and lacks a "control" over the generated test-cases, since, unlike our technique, it does not have an overall and analytical characterization over all the useful (i.e., has the potential to recover a global bad behavior) test sequences. In contrast to our ideas, theoretical work in [25, 34] focuses on complete testing over a *single and finite-state* black-box with respect to a temporal property. The decompositional approaches proposed in [11, 21] for model-checking feature-oriented software designs rely totally on model-checking techniques (no testing) and could cause false negatives. Integration testing of concurrent programs in [30] relies on a specification (unavailable in our model) of a concurrent program.

The quality assurance problem for component-based software has attracted lots of attention in software engineering. However, most work considers the problem from component developers' point of view; i.e., how to ensure the quality of components before they are released (e.g., [24, 33, 32, 28]). This view, however, is fundamentally insufficient: an extensively tested component (by the vendor) may still not perform as expected in a specific deployment environment, since the deployment environments of a component could be quite different and diverse such that they may not be thoroughly tried by the vendor. Our push-in technique approaches this problem from system developers' point of view: how to ensure that multiple components function correctly in a host system where the components are deployed. In our technique, test sequences run on a component are customized to its specific deployment environment. Unlike our approach, frameworks like [4] require a complete specification about the component to be incorporated into a system, which is not always possible.

3 Preliminaries

In this paper, we consider a system of (concurrent) black-boxes, which consists of a host system (called the *gluer*) and a collection of black-box components (simply called *black-boxes*). Each of the gluer and the black-boxes is a *unit*. In the rest of the section, we will present the model of a unit, the model of the system of black-boxes, and the global testing problem for the system.

3.1 The Unit Model

A unit is a nondeterministic and labeled *transition system* T that moves from one state to another while performing an action. Formally, $T = \langle S, s_{\text{init}}, \nabla, R\rangle$, where S is an (infinite and countable) set of states with $s_{\text{init}} \in S$ being the *initial* state, ∇ is a finite set of actions, and $R \subseteq S \times \nabla \times S$ defines the transition relation. In particular, the action set ∇ is partitioned into three disjoint subsets: $\{\epsilon\}$ (an internal action), Π (input actions), and Γ (output actions). Especially, the set $\Sigma = \Pi \cup \Gamma$, i.e., the set of *observable actions* in T, is called the *interface* of T. When the set S of states is a finite set, T is called a *finite-state transition system*.

A *behavior* of T is a sequence of actions in ∇: $a_1 \ldots a_h$ (for some h) such that there is a sequence of states $s_0 \ldots s_h$ with $s_0 = s_{\text{init}}$ and $(s_j, a_j, s_{j+1}) \in R$ for each $0 \leq j \leq h - 1$. An *observable* behavior of T is the result of dropping all the internal actions (i.e., ϵ's) from a behavior. Trivially, the empty string is an observable behavior for any unit T.

A (unit) test sequence α for T is a sequence of observable actions in Σ. A unit T is considered to be a *black-box* if its interface (i.e., Π and Γ) is the only known part in its definition. In this case, we assume that T is testable. That is, there is a black-box testing procedure $\mathbf{BBtest}(T, \cdot)$[1] such that, for any test sequence α, $\mathbf{BBtest}(T, \alpha)$ returns "yes" (i.e., α is *successful*) if α is an observable behavior of the unit T, and, $\mathbf{BBtest}(T, \alpha)$ returns "no" (i.e., α is *unsuccessful*) if otherwise.

For example, consider the black-box Comm in Figure 1, which has seven observable actions (in the figure, we use suffixes ? and ! to distinguish input and output actions respectively). Assume that the black-box is implemented as shown in Figure 5. Clearly, $send\ msg\ ack$ is a successful test sequence to Comm while $send\ msg\ fail$ is not.

Obviously, if one further assumes that the black-box is output deterministic (i.e., an input action sequence uniquely decides the corresponding output action sequence), then a test sequence for the black-box can be simply reduced to a sequence of input actions. However, there are testable units that are not necessarily output deterministic (e.g., [23, 31, 26]). Therefore, to make our algorithms (presented later) more general, we do not apply this assumption (under which, obviously, our algorithm still applies). That's why in our definition, a test sequence is always a sequence of both input actions and output actions.

3.2 The System Model

A system of concurrent black-boxes consists of a gluer G and a number of black-boxes $B_1, \ldots, B_k$, written $Sys = G(B_1, \ldots, B_k)$. The gluer and the black-boxes are all units which run concurrently and synchronize on certain actions. More precisely, G is a fully specified and (nondeterministic) finite-state unit $G = \langle S_0, s^0_{\text{init}}, \nabla_0, R_0 \rangle$, whose interface is $\Sigma_0 = \Pi_0 \cup \Gamma_0$. Each B_i is a black-box unit $B = \langle S_i, s^i_{\text{init}}, \nabla_i, R_i \rangle$, which is testable and whose interface (the only given part of the black-box) is $\Sigma_i = \Pi_i \cup \Gamma_i$. As mentioned earlier, a black-box is not necessarily a finite-state unit. The state sets $S_0, \ldots, S_k$ are all disjoint. But the interfaces $\Sigma_0, \ldots, \Sigma_k$ may not be disjoint: some units may share some common actions.

We use $\Sigma = \Sigma_0 \cup \ldots \cup \Sigma_k$ to denote all the observable actions in the system Sys (this implies that each unit's observable actions are also observable in the system), and use $Sig(a)$, called the *signature* of a, to denote the set of all $0 \leq i \leq k$ such that $a \in \Sigma_i$. Therefore, the signature indicates the units that share action a.

The system Sys, which also works as a labeled transition system, is a Cartesian product of its units. That is, $Sys = \langle S, s_{\text{init}}, \nabla, \mathbf{R} \rangle$, where $S = S_0 \times \ldots \times S_k$ is the system's (global) state set S; each unit starts from its own initial state; i.e., the initial global state s_{init} of the system is $(s^0_{\text{init}}, \ldots s^k_{\text{init}})$; and $\nabla = \{\epsilon\} \cup \Sigma$ with $\Sigma = \Sigma_0 \cup \ldots \cup \Sigma_k$ is the system's action set.

The system's (global) transition relation $\mathbf{R} \subseteq S \times \nabla \times S$ is more complex. A global transition that moves the system from a global state $(s_0, \ldots, s_k)$ to another global state $(s_0^{'}, \ldots, s_k^{'})$ while performing an action $a \in \nabla$ is in $\mathbf{R}$ iff one of the following conditions is satisfied:

[1] The black-box testing procedure can be implemented in practice for a variety of transition systems [5].

- a is an internal action (i.e., ϵ), and exactly one unit in the system performs the internal action while the remaining units do not move; i.e., $\exists 0 \leq i \leq k.\ (s_i, \epsilon, s'_i) \in R_i \wedge \forall 0 \leq j \neq i \leq k.\ s_j = s'_j$,
- a is an observable action (i.e., $a \in \Sigma$), and all the units whose interfaces contain the observable action a synchronize over the action while the remaining units do not move; i.e., $\forall 0 \leq i \leq k.\ (i \in Sig(a) \wedge (s_i, a, s'_i) \in R_i) \vee (i \notin Sig(a) \wedge s_i = s'_i)$.

In other words, at any moment in the system Sys, exactly one unit performs an internal action, exactly one unit performs an observable action that is not shared with any other unit, or multiple units synchronize over a common observable action. It shall be noticed from the above definition that the synchronizations allowed in our model are quite flexible. Not only can the units in a system synchronize over an output/input pair as most other system models allow, they can also synchronize over just an output action or an input action, if only they can perform this (no matter output or input) action at a certain global state. Also, in our model, a synchronization can either occur between a pair of units or among more than two units; thus multi-cast or broadcast is allowed. Certainly in some systems, multi-cast, broadcast, or synchronizations over only an output action or input action may be undesirable. In that case, they can be easily eliminated just by renaming the actions. It shall also be pointed out that, in the system Sys, if a global transition is a synchronization over a pair of output and input actions among some units, these two actions are considered to be one single action, and we do not discriminate whether it is output or input but just treat it as an observable action to the environment.

As defined earlier, a sequence $\alpha \in \Sigma^*$ is an observable behavior of the system Sys of black-boxes if the system, treated as a transition system, has an execution from the initial global state to some global state and, on the execution, α is the observable behavior.

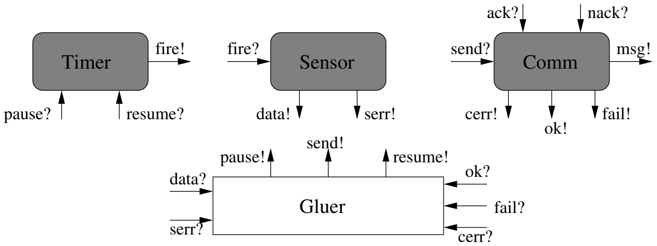


Fig. 1. A Data Acquisition System

For example, consider a data acquisition system shown in Figure 1, which consists of one `Gluer` and three black-box components: `Timer`, `Sensor` and `Comm`. The system works as follows. Once started, the `Timer` keeps signaling a *fire* event when the time interval set runs out; the `Timer` can also be paused (resp. resumed) by an incoming *pause* (resp. *resume*) event. The `Sensor` is supposed to respond to a *fire* event by signaling a *data* event when the sensor's reading is ready; it also signals a *serr* event when something is wrong inside the `Sensor`. The `Comm` component responds to a

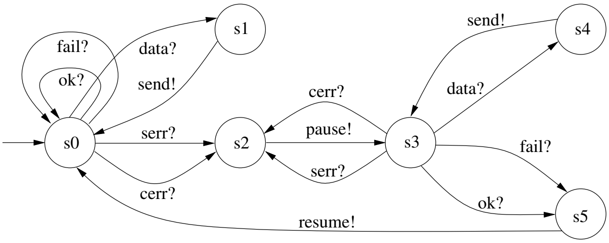


Fig. 2. The Gluer

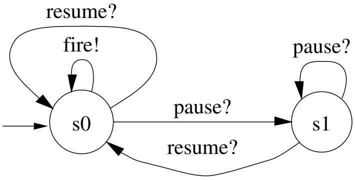


Fig. 3. Internal implementation of Timer

send event to send some data by signaling a *msg* event to some underlying network; it responds to an *ack* (resp. *nack*) event by signaling an *ok* (resp. *fail*) event to indicate that the data associated with a previous *send* event has been transmitted successfully (resp. unsuccessfully) by the underlying network; it signals an *cerr* event when something is wrong inside `Comm`. The `Gluer` (whose transition graph is depicted in Figure 2) simply relays data from `Sensor` to `Comm`; it pauses the `Timer` when something is wrong with the `Sensor` or `Comm`, and after that, it resumes the `Timer` when either an *ok* or *fail* is received from `Comm`. Together, they constitute a data acquidition system, which periodically transmits a reading of the `Sensor` through `Comm` via some underlying communication network. In this system, the `Gluer` and the three components run concurrently and synchronize with each other by sending and receiving those events (here, all synchronizations are over output/input pairs between two units). The internal implementations of the three components are shown in Figure 3, Figure 4, and Figure 5, respectively [2]. It can be seen (though not obviously) that the following sequence is an observable behavior of the system: *fire fire serr pause data send msg ack ok resume fire*, while sequence *fire fire serr data pause send* is not.

When all the black-boxes are fully specified, our system model is roughly equivalent to the IOTS studied in [26]. Our model is also closely related to I/O automata [22] (but ours is not input-enabled) and to interface-automata [9] (but ours, similar to the IOTS, makes synchronizations between units observable at the system level). These observable synchronizations are the key to testing the behavior of a system of concurrent black-boxes, where an abstract model (such as design or source code) of each black-box is unavailable.

[2] Obviously, the push-in technique does not require these transition graphs, which are provided only for readers to understand the system.

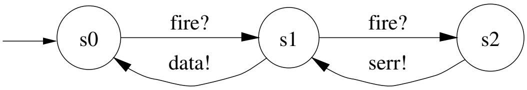


Fig. 4. Internal implementation of Sensor

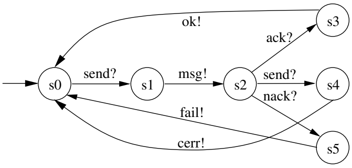


Fig. 5. Internal implementation of Comm

Let $Bad \subseteq \Sigma^*$ be a given set of test sequences that are not supposed to be the observable behaviors of the system Sys. The *global testing problem* is to verify (with a definite answer) that none of the test sequences in Bad is an observable behavior of the system. Clearly, in general, the problem can not be solved completely since the set Bad can be infinite and, for testing, only finitely many test sequences can be run. Therefore, we assume that Bad is a finite set, which can be given as an explicit list of test sequences (e.g., $Bad = \{fire\ fire, fire\ fire\ data, fire\ data\ send\ fire\}$) or as a symbolic representation (e.g., Bad is all sequences in regular expression $fire\ data\ (fire)^*\ send$ whose lengths are between 10 and 30).

4 The Push-In Technique

In this section, we present the "push-in" technique to completely solve the global testing problem, by performing unit testing over each individual black-box in the system. A test sequence is a string or a word. A finite set of test sequences is therefore a regular language and, in this paper, we use a (finite) automaton that accepts the finite set as the symbolic representation of the set. Our push-in technique is automata-theoretic. For each $1 \leq i \leq k$, the technique generates two automata: U_i and A_i. Automaton U_i, called a *unit test sequence automaton*, accepts words in alphabet Σ_i; i.e., it represents a set of test sequences for black-box B_i. Automaton A_i, called an *auxiliary automaton*, accepts words in alphabet $\Sigma_i \cup \ldots \cup \Sigma_k$ (observable actions for the black-boxes $B_i, \ldots, B_k$). Our push-in technique works in the following k steps, where i is from 1 to k:

Step i. The step consists of two tasks:
(Automaton Generation) This task generates the unit test sequence automaton U_i and the auxiliary automaton A_i. We first generate the auxiliary automaton A_i. Initially when $i = 1$, the generation is based on the Sys's description (i.e., the gluer G and the interfaces for $B_1, \ldots, B_k$) and the given set Bad. When $i > 1$, the generation is based on the auxiliary automaton A_{i-1} and the surviving set SUV_{i-1} (see below) obtained from the previous **Step** $i - 1$. If the empty string is accepted by the auxiliary automaton A_i,

then the global testing problem (none of observable behaviors of the system Sys is in Bad) returns “no” (i.e., a bad behavior of the system exists) – no further steps need to run. We then generate the unit test sequence automaton U_i directly from the auxiliary automaton A_i constructed earlier. This task is purely automata-theoretic and does not involve any testing.

(Surviving Set Generation) In this second task, using **BBtest**, we perform unit testing over the black-box B_i for all test sequences accepted by the test sequence automaton U_i (U_i always accepts a finite set). We use SUV_i, called the surviving set, to denote all the successful test sequences. If the surviving set is empty, then the global testing problem returns “yes” (i.e., none of observable behaviors of the system Sys is in Bad). Otherwise, if $i < k$ (i.e., it is not the last step), we goto the following **Step** $i+1$. If $i = k$ (i.e., it is the last step and the surviving set is not empty), then the global testing problem returns “no” (i.e., some observable behaviors of the system Sys is indeed in Bad).

In the rest of this section, we will clarify how Automata Generation and Surviving Set Generation in the k steps can be done. Since our technique heavily depends on automata theory, we would like to first build the theory foundation of our technique before we proceed further.

4.1 Theory Foundation of the Push-In Technique

Let us first make a pessimistic (the name is borrowed from the discussions in [9]) modification of the original system Sys by assuming that each black-box B_i, $1 \leq i \leq k$, can demonstrate *any* observable behavior in Σ_i^* (recalling that Σ_i is the interface of the black-box). The resulting system is denoted by $\hat{Sys}$. Clearly, every observable behavior of Sys is also an observable behavior of $\hat{Sys}$ (but the reverse is not necessarily true).

Notice that $\hat{Sys}$ does not have any black-boxes since the original black-box B_i, after the pessimistic modification, can be considered as a finite state unit $\hat{B}_i$ with only one state, where each action in $\Sigma_i \cup \{\epsilon\}$ is a label on a transition from the state back to the state. According to the semantics definition presented in Section 3.2, it is not hard to see that $\hat{Sys}$ itself, after the composition of the gluer G with all the one-state units $\hat{B}_1, \ldots, \hat{B}_k$, is a finite state transition system with $|G|$ (the number of states in the gluer) states and with actions in $\Sigma \cup \{\epsilon\}$. (Recall that $\Sigma = \Sigma_0 \cup \ldots \cup \Sigma_k$ is the union of all observable actions in the gluer and the black-boxes.) The pessimistic system can also be treated as a pessimistic (finite) automaton by making each state be an accepting state and each ϵ-transition be an ϵ-move. In this way, the language (a subset of Σ^*) accepted by the automaton is exactly all the observable behaviors of the pessimistic system.

As we have mentioned earlier, the set $Bad \subseteq \Sigma^*$ is a finite and hence regular set. Suppose that the symbolic representation of the set is given as an automaton M_{Bad} (whose state number is written $|M_{Bad}|$); i.e., the language accepted by M_{Bad} is exactly the set Bad.

Using a standard Cartesian product construction, one can build an automaton M_{global}, called the global test sequence automaton, to accept the intersection of the language accepted by the pessimistic automaton $\hat{Sys}$ and the language accepted by the automaton M_{Bad}. That is, M_{global} accepts exactly the bad and observable behaviors of the pessimistic system. Clearly, the state number in M_{global} is at most $|G| \cdot |M_{Bad}|$.

For a word $\alpha \in \Sigma^*$, we use $\alpha \downarrow_{\Sigma_i}$, $1 \leq i \leq k$, to denote the result of dropping all symbols not in Σ_i from α. That is, if α is an observable behavior of the system Sys, then $\alpha \downarrow_{\Sigma_i}$ is the corresponding observable behavior of black-box B_i. The theory foundation of our push-in technique can be summarized in the following theorem, which can be shown using the semantics defined in Section 3.2.

Theorem 1. *For any global test sequence α in Σ^*, the following two items are equivalent:*

(1) α is a bad (i.e., in Bad) observable behavior of the system Sys of black-boxes $B_1, \ldots, B_k$,

(2) α is accepted by the global test sequence automaton M_{global}, and each of the following k conditions holds:

(2.1) $\alpha \downarrow_{\Sigma_1}$ is an observable behavior of B_1,

$\vdots$

(2.k) $\alpha \downarrow_{\Sigma_k}$ is an observable behavior of B_k.

We use "class C" to denote all the α's that satisfy Theorem 1 (2). Obviously, the global testing problem (i.e., there is no bad behavior in Sys) is equivalent to the emptiness of class C.

In the push-in technique, the jobs of **Step 1**, . . ., **Step** k are to establish the emptiness of class C using both automata theory and black-box testing. One naive approach for the emptiness is to use Theorem 1 (2) directly: repeatedly pick a global test sequence α accepted by M_{global} (note that M_{global} accepts a finite language) and, using black-box testing, make sure that one of the conditions (2.i), $1 \leq i \leq k$, is false. This naive approach works but inefficiently. This is because, when M_{global} accepts a huge set (such as more than 10^{24} in our experiments shown later), trying every such element is not only infeasible but also unnecessary. Our approach of doing the job aims at eliminating the inefficiency. First, we do not pick a global test sequence α. Instead, we *compute* the test sequences run on black-box B_i from the testing *results* on black-box B_{i-1} in the previous **Step** $i - 1$. As we have mentioned at the beginning of this section, each **Step** i has two tasks to perform: Automata Generation and Surviving Set Generation, which are presented in detail as follows.

4.2 Automata Generation in Step i

This task in **Step** i is to generate two automata: the unit test sequence automaton U_i and the auxiliary automaton A_i.

Initially when $i = 1$, A_1 is constructed as $A_1 = M_{global} \downarrow_{\Sigma_1 \cup \ldots \cup \Sigma_k}$, i.e., the result of dropping every transition in M_{global} that is labeled with an observable action not in $\Sigma_1 \cup \ldots \cup \Sigma_k$. U_1 is constructed as the automaton $U_1 = A_1 \downarrow_{\Sigma_1}$ (i.e., the result of dropping every transition in A_1 that is labeled with an observable action not in Σ_1). Observe that A_1 accepts the language $\mathcal{A}_1 = \{\alpha \downarrow_{\Sigma_1 \cup \ldots \cup \Sigma_k}\colon \alpha \text{ accepted by } M_{global}\}$ and U_1 accepts the language $\mathcal{U}_1 = \{\alpha \downarrow_{\Sigma_1}\colon \alpha \text{ is in } \mathcal{A}_1\}$. The state number in either of the two automata, in worst cases, is $|M_{global}|$.

When $i > 1$, the two automata A_i and U_i are constructed from the auxiliary automaton A_{i-1} and the surviving set SUV_{i-1} obtained in the previous step. To construct A_i, we first build an automaton suv_{i-1} to accept the finite set SUV_{i-1}. Then, we build an intermediate automaton M_{i-1} that works as follows: on an input word in $(\Sigma_{i-1} \cup \ldots \Sigma_k)^*$, M_{i-1} starts simulating A_{i-1} and suv_{i-1} on the word, in parallel. During the simulation, whenever suv_{i-1} reads an input symbol that is not in Σ_{i-1} (note that suv_{i-1} only accepts words in Σ^*_{i-1}), it skips the input symbol. M_{i-1} accepts the input word when both A_{i-1} and suv_{i-1} accept. Finally, the auxiliary automaton A_i is constructed as $A_i = M_i \downarrow_{\Sigma_i \cup \ldots \Sigma_k}$. The unit test sequence automaton U_i is constructed as $U_i = A_i \downarrow_{\Sigma_i}$.

One can show that each of the two automata A_i and U_i has, in worst cases, a state number of $|A_{i-1}| \cdot |suv_{i-1}|$. Also, A_i accepts the language $\mathcal{A}_i = \{\alpha \downarrow_{\Sigma_i \cup \ldots \cup \Sigma_k} : \alpha \in (\Sigma_{i-1} \cup \ldots \cup \Sigma_k)^*$ is in $\mathcal{A}_{i-1}$ and $\alpha \downarrow_{\Sigma_{i-1}}$ is in $SUV_{i-1}\}$ and U_i accepts the language $\mathcal{U}_i = \{\alpha \downarrow_{\Sigma_i} : \alpha \in (\Sigma_i \cup \ldots \Sigma_k)^*$ is in $\mathcal{A}_i\}$.

As we have mentioned earlier, when the empty string is accepted by the auxiliary automaton A_i (a standard membership algorithm can be used to validate the acceptance), our push-in technique will return a "no" answer on the global testing problem (i.e., the system does have a bad observable behavior) and no further steps need to run.

4.3 Surviving Set Generation in Step i

The surviving set SUV_i is the set of all successful unit test sequences $\alpha \in \mathcal{U}_i$; i.e., $SUV_i = \{\alpha \in \Sigma_i^* : \alpha \in \mathcal{U}_i$ and α is an observable behavior of black-box $B_i\}$.

A straightforward way to obtain the set is to run the black-box testing procedure **BBtest** over the black-box B_i with every test sequence in $\mathcal{U}_i$. This is, however, not efficient, in particular when the set $\mathcal{U}_i$ is huge. Observable behaviors of a unit are prefix-closed: if α is not an observable behavior of B_i, then, for any β, $\alpha\beta$ can not be (i.e., test sequence $\alpha\beta$ need not be run). With prefix-closeness and $\mathbf{BBtest}$, we use the following automata-theoretic procedure to generate the surviving set SUV_i.

Recall that $\mathcal{U}_i$ is a finite set of unit test sequences and, as a regular language, accepted by the unit test sequence automaton U_i. Let m be the maximal length of all test sequences in $\mathcal{U}_i$ (the length can be obtained using a standard longest path algorithm over the transition graph of automaton U_i). Our procedure consists of the following m jobs. Each Job_j, where j is from 1 to m, is to identify (using black-box testing) all the successful test sequences (with length j) which are prefixes (which are not necessarily proper) of some test sequences in $\mathcal{U}_i$. In order to do this efficiently, the job makes use of the previous testing results in Θ_{j-1}. More precisely, each Job_j has two parts (by assumption, let Θ_0 contain only the empty word.):

- Define P_j to be the set of all the prefixes with length j of all the unit test sequences in $\mathcal{U}_i$. Calculate the set $\hat{P}_j \subseteq P_j$ such that each element in $\hat{P}_j$ has a prefix (with length $j-1$) in Θ_{j-1}. To implement this part, one can first construct an automaton (from automaton U_i) to accept the language P_j. Then, construct another automaton to accept the set Θ_{j-1}. Finally, an automaton M can be constructed from these two automata to accept the language $\hat{P}_j$. All the constructions are not difficult and do not involve testing.

– Using **BBtest**, generate the set Θ_j that consists of all the successful test sequences over black-box B_i in $\hat{P}_j$. Hence, one only runs test sequences in $\hat{P}_j$ instead of the entire P_j, thanks to the previous testing results in Θ_{j-1}.

It is left to the reader to verify that, after the jobs are completed, the surviving set SUV_i can be obtained as $\mathcal{U}_i \cap (\cup_{0\leq j\leq m}\Theta_j)$. Again, this set can be accepted by an automaton, treated as a symbolic representation of the set, constructed from automaton U_i and the automata built in the above jobs to accept Θ_j, $1 \leq j \leq m$. One can choose the procedure to output the explicit set SUV_i or its symbolic representation suv_i.

4.4 Correctness and Bad Behavior Generation

Since the global testing problem is equivalent to the emptiness of class C, we only need to show that the emptiness is answered correctly with the push-in technique. Clearly, the technique always terminates with a yes/no answer. It returns "yes" only at some **Step** i, $1 \leq i \leq k$, whose surviving set $SUV_i = \emptyset$. It returns "no" only

CASE 1. At some **Step** i, $1 \leq i \leq k$, when the auxiliary automaton A_i accepts the empty word, or

CASE 2. At the last **Step** k when $SUV_k \neq \emptyset$.

In these two cases, in order to demonstrate a global bad behavior of the system, we first define an operation called $\text{select}_j(\cdot)$, $1 \leq j \leq k$. Given a sequence α_j, the operation returns a sequence α_{j-1} (when $j = 1$, it simply returns α_j) satisfying the following conditions: $\alpha_{j-1} \in \mathcal{A}_{j-1}$, $\alpha_{j-1} \downarrow_{\Sigma_{j-1}} \in SUV_{j-1}$ and $\alpha_{j-1} \downarrow_{\Sigma_j \cup \ldots \Sigma_k} = \alpha_j$. The returned sequence α_{j-1} may not be unique. In this case, any sequence (such as a shortest one) satisfying the conditions will be fine. Now, we define another operation called $\mathbf{BadGen}_j(\cdot)$, $1 \leq j \leq k$, as follows. Given a sequence α_j, we first calculate $\alpha_{j-1} = \text{select}_j(\alpha_j)$. Then, we calculate $\alpha_{j-2} = \text{select}_{j-1}(\alpha_{j-1})$, and so on. Finally, we obtain α_1. At this time, the operation $\mathbf{BadGen}_j(\alpha_j)$ returns any sequence α satisfying the following conditions: α is accepted by M_{global} and $\alpha \downarrow_{\Sigma_1 \cup \ldots \Sigma_k} = \alpha_1$. All these operations can be easily implemented through automata constructions.

Coming back to bad behavior generation, in CASE1, we return $\mathbf{BadGen}_i(\lambda)$ (where λ is the empty sequence) as a global bad behavior. In CASE2, we simply pick any sequence α_k from SUV_k and return $\mathbf{BadGen}_k(\alpha_k)$ as a global bad behavior.

One can show that our technique is indeed correct:

Theorem 2. *If the class C is empty then the push-in technique returns "yes", otherwise it returns "no". When the technique returns yes, it shows that the system doesn't have any of the global bad behaviors in* BAD*, otherwise it indicates that the system does exhibit bad behaviors in* BAD.

In each step of our algorithm, one can use standard algorithms in automata theory to make the obtained automata like U_i's and A_i's smaller. The algorithms include eliminating unreachable states and/or minimization. Additionally, the algorithms as well as all the automata constructions mentioned in the push-in technique can be implemented using existing automata manipulation tools like Grail [1].

From the correctness theorem, we know that the push-in technique is sound and complete. However, one question still remains unsolved: Are test sequences (for black-box B_i) in each $\mathcal{U}_i$ more than necessary (in solving the global testing problem)? We can show that each $\mathcal{U}_i$ derived from our push-in technique is "optimal" in the following sense. Suppose that we have completed the first $i-1$ **Step**s (i.e., the black-boxes $B_1, \ldots, B_{i-1}$ have been tested) and have obtained $\mathcal{U}_i$ to start the subsequent steps (i.e., the remaining black-boxes $B_i, \ldots, B_k$ are not tested yet). Each test sequence α_i in $\mathcal{U}_i$ has to be run, since one can show the following two statements: There are black-boxes $B_i^*, \ldots, B_k^*$, such that α_i is a successful (resp. unsuccessful) test sequence for B_i^* and the system $G(B_1, \ldots, B_{i-1}, B_i^*, \ldots, B_k^*)$ has (resp. does not have) a global bad behavior.

Table 1. Experiment Results: Counts of Test Sequences

		maxlength=10				maxlength=20				maxlength=30			
	step$_i$	$\#A_i$	$\#U_i$	$\#SUV_i$	TC_i	$\#A_i$	$\#U_i$	$\#SUV_i$	TC_i	$\#A_i$	$\#U_i$	$\#SUV_i$	TC_i
case 1	step$_1$	$1.06X10^7$	148	47	68	$7.16X10^{15}$	$8.06X10^4$	3533	4572	$2.16X10^{24}$	$4.14X10^7$	$2.23X10^5$	$2.87X10^5$
	step$_2$	$3.05X10^6$	548	12	41	$6.92X10^{14}$	$4.62X10^5$	177	393	$1.13X10^{23}$	$2.43X10^8$	1331	2940
	step$_3$	$4.78X10^4$	$4.78X10^4$	7	39	$1.15X10^{12}$	$1.15X10^{12}$	58	297	$1.81X10^{19}$	$1.81X10^{19}$	274	1577
case 2	step$_1$	$1.38X10^7$	386	73	121	$5.90X10^15$	$2.61X10^5$	6697	9384	$1.59X10^{24}$	$1.42X10^8$	$4.74X10^5$	$6.30X10^5$
	step$_2$	$3.12X10^6$	142	13	25	$4.94X10^{14}$	$5.91X10^4$	93	203	$6.99X10^{22}$	$2.53X10^7$	645	1356
	step$_3$	$7.25X10^5$	$7.25X10^5$	0	47	$1.11X10^{13}$	$1.11X10^{13}$	0	277	$1.48X10^{20}$	$1.48X10^{20}$	0	1259
case 3	step$_1$	$1.38X10^7$	386	73	121	$5.90X10^{15}$	$2.61X10^5$	6697	9384	$1.59X10^{24}$	$1.42X10^8$	$4.74X10^5$	$6.30X10^5$
	step$_2$	$3.12X10^6$	142	13	25	$4.94X10^{14}$	$5.91X10^4$	93	203	$6.99X10^{22}$	$2.53X10^7$	645	1356
	step$_3$	$7.25X10^5$	$7.25X10^5$	0	47	$1.11X10^{13}$	$1.11X10^{13}$	13	359	$1.48X10^{20}$	$1.48X10^{20}$	129	2577
case 4	step$_1$	$1.30X10^6$	178	32	76	$3.51X10^{15}$	$2.20X10^5$	5507	8197	$1.65X10^{24}$	$1.36X10^8$	$4.44X10^5$	$6.00X10^5$
	step$_2$	$1.02X10^5$	97	0	14	$9.54X10^{13}$	$1.70X10^5$	0	128	$2.39X10^{22}$	$1.22X10^8$	0	906
	step$_3$	0	0	0	0	0	0	0	0	0	0	0	0

5 Experiments

All the experiments were performed on a PC with a 800MHz Pentium III CPU and 128MB memory. The Grail [1] tool was used to perform almost all the automata operations[3]. The entire experiment process was driven by a Perl script and carried out automatically. Our experiments were run on the system of black-boxes shown in Figure 1. In the experiments, we designated black-boxes `Timer`, `Sensor` and `Comm` as B_1, B_2, and B_3, respectively. The internal implementations of the black-boxes are shown in Figures 3, 4 and 5, on which the unit testing in the experiments was performed. We have totally run twelve experiments (each experiment is a complete execution of the push-in technique), which are divided into four cases. Each of the four cases consists of three experiments, which are illustrated in detail as follows.

Case 1. Firstly, we wish that whenever a $pause$ event takes place, there should be no more $send$ until a $resume$ occurs. The corresponding bad behaviors are specified as a regular expression, $\Sigma^* p(\Sigma - \{r\})^* s \Sigma^*$, where Σ is the set of all the twelve events in the system; p, r, and s stand for the $pause$, $send$, and $resume$, respectively (such abbreviation will be used throughout this section). For the first experiment run in this case, we chose the Bad to be all words in the regular expression that are not longer

[3] We implemented (in C) three additional operations to manipulate automata with ϵ-moves and to count the number of words in a finite language accepted by an automaton, which are not provided in Grail.

than 10 (denoted by "maxlength=10"). The remaining two experiments were run with "maxlength=20" and "maxlength=30", respectively. To understand the results shown in Table 1, we go through the third experiment (i.e., "maxlength=30"). The results of the experiment are shown in the box at the right upper corner in the table (i.e., under the four columns associated with "maxlength=30" and in the three rows ("step_1", "step_2", "step_3") associated with "case 1"). The three steps in the experiment correspond to the three **Step**s (since there are three black-boxes) in the push-in technique. The auxiliary automaton A_1 calculated in **Step 1** accepts totally $\#A_1 = 2.16 \times 10^{24}$ test sequences. The unit test sequence automaton U_1 accepts $\#U_1 = 4.14 \times 10^7$ test sequences. Using the black-box testing procedure in Section 4.3, we actually only performed $TC_1 = 2.87 \times 10^5$ unit tests over B_1 (the `Timer`), among which $\#SUV_1 = 2.23 \times 10^5$ tests survived. In **Step 2** and **Step 3**, we obtained $\#A_2, \#U_2, \#A_3, \#U_3$ similarly as shown in the table. In particular, we actually performed $TC_2 = 2940$ unit tests over the `Sensor` in **Step 2** and $TC_3 = 1577$ unit tests over the `Comm` in **Step 3**. Since the last surviving set SUV_3 is not empty ($\#SUV_3 = 274$), the experiment detects a global bad behavior specified in this case.

Notice that the total number of unit tests run in this experiment is $TC_1+TC_2+TC_3$, which is not more than 2.92×10^5. This number essentially indicates the actual "cost" of the experiment in deciding whether there is a global bad behavior specified in the case and whose length is bounded by 30. This number is quite good considering the astronomical number $\#A_1 = 2.16 \times 10^{24}$ which would be the number of integration test sequences if one run integration testing, since $M_{global} = A_1$ in the system. The other two experiments ("maxlength=10" and "maxlength=20") also detected a global bad behavior and results are shown in the first three rows under "maxlength=10" and "maxlength=20" in Table 1 (the costs of these two experiments, which are 148 and 5262 respectively, become much smaller).

Case 2. The detected bad behaviors are due to the concurrency nature of these black-boxes: a $fire$ was issued before the $pause$ is sent to `Timer`, which eventually leads to another $send$. For instance, a global bad behavior could be like the following: $fire\ data\ send\ msg\ fire\ data\ send\ cerr\ fire\ data\ pause\ send$. From this observation, we believed that the system might also have other bad behaviors: after a $cerr$ takes place, there could be another $cerr$ coming before a $resume$ occurs. Such bad behaviors are encoded by $\Sigma^* c(\Sigma - \{r\})^* c\Sigma^*$. The three experiments in this case, however, did not detect such bad behaviors (i.e., $\#SUV_3 = 0$ for all lengths, shown in the third row "step_3" associated with "case 2" in Table 1).

Case 3. Based upon the experiments in the previous case, we carefully studied the system and realized that the implementation of `Comm` might be wrong: after an error occurs (i.e., a $cerr$ outputs), `Comm` is supposed to retain its state prior to the output of the $cerr$, while it does not. After correcting this bug (by making the internal implementation of `Comm`, shown in Figure 5, move to state $s2$ instead of $s0$ after a $cerr$ is output), in this case, we run the three experiments again. The experiments detected bad behaviors only with length more than 10 (i.e., $\#SUV_3 = 0$ when maxlength is 10 and $\#SUV_3 > 0$ when maxlength is 20 and 30, shown in Table 1).

Case 4. Now we want to test that: after an error occurs in `Sensor` (i.e., a $serr$ is issued), there will be at most one more $fire$ issued before a $resume$ occurs. The corresponding bad behaviors are encoded by $\Sigma^* serr(\Sigma - \{r\})^* f(\Sigma - \{r\})^* f(\Sigma - \{r\})^* r\Sigma^*$, where f stands for $fire$. Our experiments did not detect any of such behaviors for all the three choices of maxlength: 10, 20, 30. In fact, in the experiments, no testing over `Comm` was needed. This is because, shown in the last three rows of Table 1, $\#SUV_2$ is 0 for all the three choices.

We measured the total time that our script used for automata manipulations in each of the twelve experiments, shown in Table 2. In the table, the "result" shows whether a global bad behavior was detected in an experiment; i.e., "$\times$" (resp. "$\surd$") indicates "detected" (resp. "not detected"). As shown in the table, the total time is within a minute for all the four experiments with "maxlength=10". For "maxlength=20", the time is still acceptable (within an hour). When the maxlength is increased to 30, the time is still within our patience (which was set to be 24 hours). Yet, our script could not finish within the patience for any experiment when we tried to push maxlength to 40. Even though determinization and minimization are optional in our push-in technique, we made them mandatory in our experiments. In this way, we can cross-compare the sizes of the automata obtained in each step of the experiments. The largest size of all the automata constructed in the twelve experiments, after determinization and minimization, is with 726 states and 2138 transitions. In an experiment with maxlength=40, the script tried to make an automaton (with 1182 states) deterministic and failed to do so within our patience.

Exhaustive integration testing over a concurrent system is in general infeasible. However, the experiments show that, using the push-in technique, we can completely solve the global testing problem with a substantially smaller number of tests over each individual black-box only, even for an extremely large set of Bad. For instance, the total number of unit tests (TC_i's) performed in each of the four experiments with "maxlength=30" is in the order of 10^5, while each Bad is in the order of 10^{24} (notice that each Bad is always larger than each $\#A_1$, shown in Table 1).

Table 2. Experiment Results: Time Efficiency

	maxlength=10		maxlength=20		maxlength=30	
Cases	time	result	time	result	time	result
Case 1	$\sim$25s	$\times$	$\sim$40m	$\times$	$\sim$19h	$\times$
Case 2	$\sim$34s	$\surd$	$\sim$58m	$\surd$	$\sim$18h	$\surd$
Case 3	$\sim$36s	$\surd$	$\sim$56m	$\times$	$\sim$18h	$\times$
Case 4	$\sim$17s	$\surd$	$\sim$22m	$\surd$	$\sim$5h	$\surd$

6 Future Work

This paper presents an automata-theoretic and decompositional technique to testing a system of concurrent black-boxes, which is automatic, sound, and complete. Our technique can be generalized to many other forms of bad behavior specifications (i.e., the finite set Bad). For instance, we may that specify that Bad consist of all observable sequences not longer than 40, each of which can make the gluer enter a given (undesired) state. But the exact formalisms for bad behavior specifications need further investigation. Our model of the system is based on synchronized communications. Therefore, it would be interesting to see whether the approach can be generalized to some forms of asynchronous (e.g., shared-variable) systems. Black-boxes in our model are event-driven; it is also worthwhile to study other decompositional testing approaches for data-driven black-boxes. Sometimes, our push-in technique fails to complete, due to an extremely large bad behavior set Bad (e.g., our experiments with "maxlength=40" shown earlier, whose global test sequences deduced from Bad are roughly as many as 10^{33}). In this case, we need study methods to (symbolically) partition the set into smaller subsets such that the push-in technique can be run over each smaller subset. In this way, a global bad behavior could instead be found. In our definition of the push-in technique, there is not a pre-defined ordering in testing the black-boxes. For instance, in our experiments, the ordering was `Timer`, `Sensor`, `Comm`, based on the size of a black-box's interface. Clearly, more studies are needed to clarify the relationship between the efficiency of our technique and the choices of the ordering.

References

1. Grail homepage. http://www.csd.uwo.ca/research/grail/.
2. Martn Abadi and Leslie Lamport. Composing specifications. *TOPLAS*, 15(1):73–132, 1993.
3. Rajeev Alur, Thomas A. Henzinger, Freddy Y. C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. MOCHA: Modularity in model checking. In *CAV'98*, volume 1427 of *LNCS*, pages 521–525. Springer, 1998.
4. A. Bertolino and A. Polini. A framework for component deployment testing. In *ICSE'03*, pages 221–231. IEEE Press, 2003.
5. Ed Brinksma and Jan Tretmans. Testing transition systems: An annotated bibliography. In *Proc. 4th Summer School on Modeling and Verification of Parallel Processes*, pages 187–195. Springer-Verlag, 2001.
6. J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using modelchecking. WVU Technical Report #NASA-IVV-96-022.
7. E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *LICS'89*, pages 353–362. IEEE Press, 1989.
8. A. Coen-Porisini, C. Ghezzi, and R. A. Kemmerer. Specification of realtime systems using ASTRAL. *TSE*, 23(9):572–598, 1997.
9. Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *FSE'01*, pages 109–120. ACM Press, 2001.
10. A. Engels, L.M.G. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In *TACAS'97*, volume 1217 of *LNCS*, pages 384–398. Springer, 1997.
11. Kathi Fisler and Shriram Krishnamurthi. Modular verification of collaboration-based software designs. In *ESEC/FSE'01*, pages 152–163. ACM Press, 2001.
12. C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN'03*, volume 2648 of *LNCS*, pages 213–225. Springer, 2003.
13. Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. In *ESEC/FSE'99*, volume 1687 of *LNCS*, pages 146–163. Springer, 1999.

14. S. Ghosh and P. Mathur. Issues in testing distributed component-based systems. In *First ICSE Workshop on Testing Distributed Component-Based Systems*, 1999.
15. Dimitra Giannakopoulou, Corina S. Pasareanu, and Jamieson M. Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *ICSE'04*, pages 211–220. IEEE Press, 2004.
16. Mary Jean Harrold. Testing: a roadmap. In *Proceedings of the conference on the future of software engineering*, pages 61–72. ACM Press, 2000.
17. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, , and Shaz Qadeer. Thread-modular abstraction refinement. In *CAV'03*, volume 2725 of *LNCS*, pages 262–274. Springer, 2003.
18. Ralph D. Jeffords and Constance L. Heitmeyer. A strategy for efficiently verifying requirements. In *FSE'03*, pages 28–37. ACM Press, 2003.
19. C.B. Jones. Tentative steps towards a development method for interfering programs. *TOPLAS*, 5(4):596–619, 1983.
20. Leslie Lamport. Specifying concurrent program modules. *TOPLAS*, 5(2):190–222, 1983.
21. Harry Li, Shriram Krishnamurthi, and Kathi Fisler. Verifying cross-cutting features as open systems. *ACM SIGSOFT Software Engineering Notes*, 27(6):89–98, 2002.
22. N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989.
23. Lev Nachmanson, Margus Veanes, Wolfram Schulte, Nikolai Tillmann, and Wolfgang Grieskamp. Optimal strategies for testing nondeterministic systems. In *ISSTA'04*, pages 55–64. ACM Press, 2004.
24. Alessandro Orso, Mary Jean Harrold, and David S. Rosenblum. Component metadata for software engineering tasks. In *EDO'00*, volume 1999 of *LNCS*, pages 129–144. Springer, 2000.
25. Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. In *FORTE/PSTV'99*, pages 225–240. Kluwer, 1999.
26. Alexandre Petrenko, Nina Yevtushenko, and Jia Le Huo. Testing transition systems with input and output testers. In *TestCom'03*, volume 2644 of *LNCS*, pages 129 – 145. Springer, 2003.
27. A. Pnueli. In transition from global to modular temporal reasoning about programs, 1985. In K.R. Apt, editor, Logics and Models of Concurrent Systems, sub-series F: Computer and System Science.
28. D. Rosenblum. Adequate testing of componentbased software. Department of Information and Computer Science, University of California, Irvine, Technical Report 97-34, August 1997.
29. C. Szyperski. Component technology: what, where, and how? In *ICSE'03*, pages 684–693. IEEE Press, 2003.
30. C. Tai and R. H. Carver. Testing of distributed programs. In *Parallel and Distributed Computing Handbook*, pages 955–978. McGraw-Hill, 1996.
31. Jan Tretmans and Ed Brinksma. Torx: Automated model-based tesing. In *First European Conference on Model-Driven Software Engineering*, pages 31–43, 2003.
32. J. Voas. Developing a usage-based software certification process. *IEEE Computer*, 33(8):32–37, August 2000.
33. John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA'02*, pages 218–228. ACM Press, 2002.
34. Gaoyan Xie and Zhe Dang. An automata-theoretic approach for model-checking systems with unspecified components. In *Formal Approaches to Software Testing*, volume 3395 of *LNCS*, pages 155–169, 2004.

Automated Generation of Positive and Negative Tests for Parsers

Sergey Zelenov and Sophia Zelenova

Institute for System Programming of Russian Academy of Sciences
{zelenov, sophia}@ispras.ru
http://www.ispras.ru/~RedVerst/

Abstract. In this paper we describe a specification-based approach to automated generation of both positive and negative test sets for parsers. We propose coverage criteria definitions for such test sets and algorithms for generation of the test sets with respect to proposed coverage criteria. We also present practical results of the technique application to testing syntax analyzers of several languages including C and Java.

Keywords: specification-based test generation, coverage criterion, compiler testing, parser, positive tests, negative tests, mutation testing, formal language, BNF grammar.

1 Introduction

Compilers are the most important tools used in software development. Reliability and correctness of compilers is a question of vital importance. Indeed, correctness of any application depends on correctness of a compiler the application was compiled by. Input data for a compiler has a very complicated structure. Besides, transformations performed by compilers are also very sophisticated. Thus, all phases of compiler testing (test selection, test running, SUT[1] outcome analysis) need automation.

Syntax analysis is the very first phase of the compilation process. Correctness of functionality of the other compilation phases (semantics checking, optimizations, code generation) depends on correctness of syntax analysis. So, syntax analyzer testing is a base for testing all other phases of compilation.

Grammar description in the form of BNF is a usual way to define syntax formally. In fact, BNF-description is a specification of syntax analyzer's functionality. Thus, the specification-based testing approach (see [15]) is the most winning solution in this area. Existence of a formal description allows an automation of test selection. Therefore, testing effort is reduced. Besides, systematic character of testing increases confidence in test results.

Many authors have been investigating the problem of grammar-based test selection for syntax analyzers. Fundamental paper [17] introduces the following

[1] The abbreviation *SUT* stands for "system under test".

W. Grieskamp and C. Weise (Eds.): FATES 2005, LNCS 3997, pp. 187–202, 2006.

coverage criterion for positive test[2] sets: For each rule of the grammar, the test set should contain a sentence that uses this rule in some derivation tree. In the paper, Purdom proposed an algorithm to generate the smallest test set meeting this criterion. However, the Purdom's criterion was found unsatisfactory. Lämmel in [9] showed that test sets generated by Purdom's algorithm does not detect elementary errors. Lämmel proposed the following stronger coverage criterion: For each pair of rules of a grammar a test set should contain a sentence that uses the first rule immediately after the second rule in some derivation tree.

There are also several probabilistic approaches (see [6, 12, 10, 11]). These approaches define no coverage criteria. Therefore, the following problem rises: When should a generator finish generating tests? To solve this problem, authors often introduce probabilities of rule occurrences in derivation trees. When a rule is used in a derivation tree of the next test, the corresponding probability decreases. In any case, there is a termination problem for these approaches. Besides, random nature of a generation process can not guarantee systematic testing.

All approaches quoted above concern generation of positive tests for syntax analyzer. Nowadays, there are very few works that propose techniques to generate negative tests[3] for syntax analyzer. However, negative tests are very important because admission of an incorrect lexeme sequence by syntax analyzer may lead to abnormal termination of the compilation.

Harm and Lämmel [7] have made a hypothesis that negative tests for syntax analyzer may be generated with the help of a positive tests generator using mutation testing techniques (see [5, 13]). The main idea is as follows. First, one should modify the original grammar in order to obtain a number of grammars (mutants) defining languages that are close to but not equivalent to the original one. Next, for each mutant grammar one should generate positive tests in order to obtain potentially negative tests. General problems of this approach are:

- A mutant grammar may be equivalent to the original one. Such mutants must be detected and must not be used in the test generation process.
- Tests generated for a mutant grammar may belong to the original language. In other words, some of potentially negative tests in fact may be positive. All such tests must be removed from the negative test set.

In general case both these problems are very difficult. The second one may be solved by using a reference syntax analyzer of the same language. Unfortunately, such a reference analyzer may not be accessible.

In this paper we describe coverage criteria for syntax analyzer testing on the basis of classical syntax analysis algorithms (see [1]). Such an approach seems suitable because we want to obtain test sets for syntax analyzers, and so, we should estimate quality of test sets with respect to characteristics of the SUT (i.e. syntax analyzer) such as, for example, functional coverage or code coverage (see [2]). The technique we describe is developed in the movement

[2] A sentence is called a *positive test* if it is a sentence of the target formal language.

[3] A sentence is called a *negative test* if it is not a sentence of the target formal language.

of our general model-based approach to compiler testing (see [18, 8, 16]). We consider classical algorithms of syntax analysis as models of syntax analyzer's behavior. As mentioned above, there are very few works concerning the problem of negative tests generation. This paper proposes a solution of this problem.

The remainder of the paper is organized as follows. In Section 2 some basic notions needed in the main part of the paper are spelled out. In Section 3 our testing approach is described. In Section 4 experimental results are presented. In Section 5 the paper is concluded.

2 Preliminaries

Let us start with some standard definitions concerning formal languages. More strict and detailed consideration of facts presented here may be found in well-known book of A .Aho, R. Sethi, and J. D. Ullman [1].

A context-free grammar G of a formal language is a quadruplet $(\mathcal{T}, \mathcal{N}, \mathcal{P}, S)$, where $\mathcal{T}$ is a set of terminals, $\mathcal{N}$ is a set of nonterminals, $\mathcal{P}$ is a finite set of productions or rules, $S \in \mathcal{N}$ is called start symbol.

By $\mathfrak{L}_G$ denote a language generated by the grammar G.

A grammar $G' = (\mathcal{T}', \mathcal{N}', \mathcal{P}', S')$ is called an *augmented grammar* of G if $\mathcal{T}' = \mathcal{T}$, $\mathcal{N}'$ is equal to $\mathcal{N}$ supplemented by a new nonterminal S' (a new start symbol), and $\mathcal{P}'$ is equal to $\mathcal{P}$ supplemented by a new production $S' \to S$.

A sequence of grammar symbols (terminals and nonterminals) is called a *sentential form* of G. In this paper Greek letters from beginning of the alphabet (α, β, ...) stand for sentential forms. The empty form is denoted by ε.

A sentential form is a *right sentential form* if it has a rightmost derivation.

Example. Consider the following grammar:

$$\begin{aligned} S &\to AB \\ A &\to cd \\ B &\to eCf \\ C &\to Ae \end{aligned}$$

In this grammar the sentential form cdB does not have a rightmost derivation, i.e. this form can not be obtained by series of expansions of rightmost nonterminals. The form $AeCf$ is an example of right sentential form. ▷

Let α be a right sentential form. A subsequence $\beta \subseteq \alpha$ is called a *handle* if β may be reduced to some nonterminal such that the sentential form obtained from α by this reduction may be reduced to the start symbol.

Example. Consider the grammar from the previous example. As mentioned above, the sequence $AeCf$ is a right sentential form. It includes two subsequences Ae and eCf, which may be reduced to nonterminals C and B respectively. Nevertheless, only the sequence eCf may be considered as a handle because the sentential form CCf can not be derived from the start symbol. ▷

A prefix of a right sentential form is called a *viable prefix* of the form if it does not exceed the right bound of the rightmost handle of the form.

A production from a grammar G is called an *item* of the grammar G if it has a dot in some position of its right part. By $\mathfrak{P}$ denote the set of all items of G.

Example. There are 4 items in the production $A \to XYZ$: $A \to \bullet XYZ$, $A \to X \bullet YZ$, $A \to XY \bullet Z$, and $A \to XYZ\bullet$. ▷

An item of a grammar G is called *kernel* if it has a dot not at the beginning of the right part of the production or if it is equal to the item $S' \to \bullet S$ of the augmented grammar G'.

Let I be a set of items. Let J be a minimal set of items such that the following conditions hold: (i) $I \subseteq J$; (ii) for any item $A \to \alpha \bullet B\beta \in I$ and any production $B \to \gamma$ the item $B \to \bullet\gamma$ belongs to J. Then J is called a *closure* of I and is denoted by $closure(I)$.

Consider a pair (I, X), where I is a set of items of a grammar G and X is a grammar symbol (terminal or nonterminal). We define the function *goto* by the rule $goto(I, X) = closure(\{A \to \alpha X \bullet \beta \mid A \to \alpha \bullet X\beta \in I\})$.

Consider an augmented grammar $G' = (\mathcal{T}, \mathcal{N}', \mathcal{P}', S')$ with $\mathcal{N}' = \mathcal{N} \cup \{S'\}$ and $\mathcal{P} = \mathcal{P} \cup \{S' \to S\}$. Let $I_0 = closure(\{S' \to \bullet S\})$. Starting with I_0, we construct a system of item sets $I_0, \ldots, I_N$ such that for any pair (I_k, X), where $k = 0, \ldots, N$ and X is a grammar symbol, there exists an index $j = 0, \ldots, N$ such that $goto(I_k, X) = I_j$. Such a system of item sets is called a *canonical item set system.* Using a canonical system $I_0, \ldots, I_N$, we create a finite state machine $\mathfrak{A}$ for recognition of viable prefixes. Namely, we take canonical item sets I_j as states s_j of $\mathfrak{A}$ and specify transitions by the function *goto.*

There are two well-known syntax analysis algorithms: LL-analysis and LR-analysis (see [1]). An LL-analyzer constructs the leftmost derivation of a target language sentence using transition diagrams or predictive analysis tables. A non-recursive implementation of an LL-analyzer uses a stack and a predictive analysis table. Initially, the stack contains the "end of line character" \$ and the start symbol of the grammar. At each step the analyzer deals with a current input symbol a and a symbol X on the top of the stack. Behavior of the analyzer is determined by these two symbols as follows:

- if $X = a = \$$, then the analyzer finishes successfully;
- if $X = a \neq \$$, then the analyzer pops the symbol X from the stack and pass on to the next input symbol;
- if X is a nonterminal, then the analyzer is looking for an expansion of symbol X such that the symbol a is acceptable for this expansion. After that, the symbol X in the stack is replaced by inverted sequence of symbols of the expansion. For example, if the expansion has the form $X \to ABC$, then the analyzer replaces X by the sequence CBA. As a result, the symbol A is on the top of the stack. Conflicts happening in expansion search process may be resolved with the help of look-ahead input symbols.

Now we consider an LR-analyzer, which also uses a stack. Such an analyzer has two main operations:

- to shift a new input symbol from the input stream to the stack;
- to reduce several successive symbols from the top of the stack to some nonterminal.

At each step the stack contains a viable prefix of some right sentential form. Any shift/reduce action pushes to the stack a state symbol s_j that corresponds to a current viable prefix. The analyzer makes a decision about shift/reduce action depending on the pair (symbol s_j; current input symbol). If the stack contains the start symbol of the grammar G, then the analyzer finishes.

3 Technique Description

3.1 Positive and Negative Tests for Parser

In this paper a *parser* means a boolean function that is defined at sequences of terminal symbols and is equal to 'true' iff a sequence belongs to given formal language and is equal to 'false' otherwise. In fact, parser implementations may also have some additional features (for example, parser may construct derivation tree or output error messages) but in this paper we ignore such features.

A test for a parser is called *positive* if the parser is required to return 'true' on this test. In other words, a positive test is a sentence of the target language.

A test for a parser is called *negative* if the parser is required to return 'false' on this test. In other words, a negative test is a terminal sequence that is not a sentence of the target language.

To construct a positive test, it is enough to derive some terminal sentential from the start grammar symbol with the help of grammar productions with restricted production recursion depth.

As for negative tests, there are two problems:

- how much difference should be between negative tests and sentences of the target language?
- how to create a sentence that is certainly not a sentence of the target language?

First, we answer the second question.

Consider a grammar $G = (\mathcal{T}, \mathcal{N}, \mathcal{P}, S)$. For any symbol $X \in \mathcal{T} \cup \mathcal{N}$ we define a set $\mathcal{U}_X$ of occurrences of the symbol X in G. This set contains all pairs

$$(\text{production } p \in \mathcal{P}; \text{ number } i \text{ of the symbol } X \text{ in the production } p)$$

such that a symbol standing on i-th position of the right part of p is the symbol X. A pair $(p, i) \in \mathcal{U}_X$ is called an *occurrence of the symbol X in the production p*.

Let t be a terminal. For any occurrence $u \in \mathcal{U}_t, u = (p, i), p = X \to \alpha t \beta$ of t in G we construct a set $\mathcal{F}_u$ of terminals $t' \in \mathcal{T}$ such that there exists a derivation

$$S \overset{*}{\Rightarrow} \gamma X \delta \overset{p}{\Rightarrow} \gamma \alpha t \beta \delta \overset{*}{\Rightarrow} \alpha' t t' \beta'.$$

Here Greek letters stand for some subsentential forms, i.e. sequences of nonterminals and terminals. If there exists a derivation $S \overset{*}{\Rightarrow} \gamma X \overset{p}{\Rightarrow} \gamma\alpha t$ of a sentence that ends with terminal t, then the set $\mathcal{F}_u$ contains the empty sequence ε.

By $\mathcal{F}_t$ denote the set $\bigcup\limits_{u \in U_t} \mathcal{F}_u$. In other words, the set $\mathcal{F}_t$ is a set of terminals that may be an immediate continuation of the terminal t.

Consider the set of terminals that can not immediately follow t:

$$\mathfrak{N}_t = (\mathcal{T} \cup \{\varepsilon\}) \setminus \mathcal{F}_t.$$

Statement 1. A terminal sequence containing a subsequence tt' with $t' \in \mathfrak{N}_t$ is not a sentence of the formal language generated by the grammar G.

Proof. This is clear from the definition of the set $\mathfrak{N}_t$. ▷

Let $\alpha = t_1 \ldots t_n$ be a terminal sequence such that there exists a derivation $S \overset{*}{\Rightarrow} \beta\alpha\gamma$. For the sequence α, we a define the set $\mathfrak{N}_\alpha$ of terminals that can not immediately follow α. More preciesely, $t' \in \mathfrak{N}_\alpha$ iff there exist no derivation $S \overset{*}{\Rightarrow} \beta\alpha t'\gamma$. Hence, each sequence $\beta\alpha t'\gamma$ with $t' \in \mathfrak{N}_\alpha$ is not a sequence of formal language generated by the grammar G.

So, we can create terminal sequences that certainly do not belong to the target formal language. In the next subsection we discuss the following problem: How to obtain a representative set of negative tests for parser.

3.2 Coverage Criteria

The main operation performed by LL-parsers and LR-parsers is making decision about further actions on the basis of some incomplete data (that is a delivered part of the input stream). An LL-parser makes such a decision on the basis of a pair (nonterminal on the top of the stack; current input symbol); An LR-parser makes such a decision on the basis of a pair (state symbol on the top of the stack; current input symbol). Therefore we formulate the following coverage criteria for positive test set:

(PLL) All pairs

(nonterminal A; acceptable input token t)

must be covered. A pair (A, t) is covered iff a test set contains a terminal sequence that is contained in the target language and has a derivation $S \overset{*}{\Rightarrow} \alpha A\beta \Rightarrow \alpha t\gamma\beta$. In other words, in the course of processing that sequence an LL-parser reaches a situation, when A is on the top of the stack and t is a current input symbol. A variant of this criterion is presented in [4].

(PLR) All pairs

(state symbol s_i;
transition from the state s_i marked by a grammar symbol X)

must be covered. A pair (s_i, X) is covered iff a test set contains a terminal sequence that is contained in the target language and has a derivation

$S \overset{*}{\Rightarrow} \alpha X \beta$ such that the prefix α corresponds to s_i. In other words, in the course of processing that sequence an LR-parser reaches a situation, when s_i is on the top of the stack and some prefix of the input stream is a terminal sequence that may be reduced to X.

In a similar manner, we formulate the following coverage criteria for negative test set (these two criteria have a parameter r that stands for a quantity of "correct" terminals prefiwing a "wrong" terminal):

(NLL$_R$) Let A be a nonterminal. A terminal sequence $t_1 \dots t_r$ is called an *acceptable terminal pre-sequence of* A if there exists a sentential form $\alpha t_1 \dots t_r A \beta$ derived from the start grammar symbol. Consider a union of sets $\mathfrak{N}_{t_1 \dots t_r}$ for all acceptable terminal pre-sequences of the symbol A with the length $r \leq R$. The criterion is: All pairs (A, t') must be covered, where t' is contained in the considered union. A pair (A, t') is covered iff a test set contains a terminal sequence that is not a sentence of the target language and in the course of processing this sequence an LL-parser reaches a situation, when A is on the top of the stack and the "wrong" symbol t' is a current input symbol.

(NLR$_R$) Let s_i be a state symbol of the FSM $\mathfrak{A}$ recognizing viable prefixes. A terminal sequence $t_1 \dots t_r$ is called an *acceptable terminal pre-sequence of* s_i if there exists a sentential form $\alpha t_1 \dots t_r \beta$ derived from the start grammar symbol such that the prefix $\alpha t_1 \dots t_r$ corresponds to s_i. Consider a union of sets $\mathfrak{N}_{t_1 \dots t_r}$ for all acceptable terminal pre-sequences of s_i with the length $r \leq R$. The criterion is: All pairs (s_i, t') must be covered, where t' is contained in the considered union. A pair (s_i, t') is covered iff a test set contains a terminal sequence that is not a sentence of the target language and in the course of processing this sequence an LR-parser reaches a situation, when s_i is on the top of the stack and t' is a current input symbol.

In order to obtain a situation (A, t') for the criterion (NLL) and a situation (s_i, t') for the criterion (NLR), a negative test set must contain a terminal sequence a with prefix $t_1 \dots t_r t'$ such that an LL- or LR-parser obtains the required symbol sequence in the stack in the course of processing the prefix $t_1 \dots t_r$. So, to create negative tests, we modify sentences of the target language by insertion or replacement of some terminals such that each modified sequence have "wrong" prefix $t_1 \dots t_r t'$ with $t' \in \mathfrak{N}_{t_1 \dots t_r}$.[4]

[4] There is a connection between the proposed approach and mutation testing. Namely, for each terminal sequence that is a negative test obtained as described above we can construct a mutant grammar such that this negative test is a sentence of a formal language generated by this mutant grammar.

One of principles of mutation testing is the following *coupling effect*: Complex faults are coupled to simple faults in such a way that a test set that detects all simple faults in a program will detect most complex faults (see [14]). According to the principle of coupling effect, we can apply only simple mutations. Note that the approach proposed in this paper is completely agree with this principle.

At the end of this subsection, we define two useful coverage criteria for some special grammars.

Let a canonical item set system of a grammar G meets the following condition: If I_i and I_j are two different item sets from the canonical system, then kernel item subsets of sets I_i and I_j do not intersect. Note that for this kind of grammar the following condition holds: If all grammar items are covered, then all pairs (state of the FSM $\mathfrak{A}$; transition from this state) are also covered. Consider the following coverage criterion for positive test set:

(WPLR) All pairs

$$\text{(item } \pi = B \to \alpha \bullet X\beta \text{ of a grammar } G; \\ \text{terminal } t \text{ acceptable for the symbol } X \text{ as a first terminal)}$$

must be covered. A pair (π, t) is covered iff a test set contains a sentence of the target language with a derivation $S \overset{*}{\Rightarrow} \gamma B\delta \overset{*}{\Rightarrow} \gamma\alpha X\beta\delta \overset{*}{\Rightarrow} \gamma\alpha t\mu\beta\delta$.

In case of grammars of the specified kind, this criterion is stronger then (PLR). Indeed, it is easy to show that each state of the FSM $\mathfrak{A}$ may be designated by its kernel item subset. Thus, since kernel item subsets of different states do not intersect, then if all items are covered, then all states are also covered.

Similarly, consider the following coverage criterion for negative test set:

(WNLR$_R$) Let $\pi = B \to \alpha \bullet \beta$ be an item of a grammar G. A terminal sequence $t_1 \ldots t_r$ is called a *terminal pre-sequence acceptable for the item* π iff there is a terminal sequence $\mu t_1 \ldots t_r \bullet \lambda$ with a derivation

$$S \overset{*}{\Rightarrow} \gamma B\delta \overset{\pi}{\Rightarrow} \gamma\alpha \bullet \beta\delta \overset{*}{\Rightarrow} \mu t_1 \ldots t_r \bullet \lambda,$$

i.e. $\mu t_1 \ldots t_r$ is derived from $\gamma\alpha$ and λ is derived from $\beta\delta$. Consider a union of sets $\mathfrak{N}_{t_1 \ldots t_r}$ for all acceptable terminal pre-sequences for π with the length $r \leq R$. The criterion is: All pairs (π, t') must be covered, where t' is contained in the considered union. A pair (π, t') is covered iff a test set contains a terminal sequence that is not a sentence of the target language and one of its prefixes has a form $\mu t_1 \ldots t_r t'$, where $t_1 \ldots t_r$ is a terminal pre-sequence acceptable for π such that $t' \in \mathfrak{N}_{t_1 \ldots t_r}$.

3.3 Test Sets

In this subsection we describe techniques for automatic generation test sets meeting the coverage criteria introduced above.

Positive Tests. Let $G = (\mathcal{T}, \mathcal{N}, \mathcal{P}, S)$ be a grammar. Let B be a nonterminal, A be a grammar symbol. By $\mathcal{U}_A^B$ denote a set of all occurrences (p, i) of A in G

such that $p = B \to \alpha A \beta$. Let us enumerate elements of $\mathcal{U}_A^B$ in the order in which they occur in the text of BNF. By $A^{(B,j)}$ denote a j-th element of $\mathcal{U}_A^B$.

Example. Let a nonterminal B be defined by the following productions:

$$\begin{gathered} B \to ADE \\ B \to CADA \\ B \to KB. \end{gathered}$$

Here we have three occurrences of A: $A^{(B,1)}$ from the first production ($B \to A_1DE$), $A^{(B,2)}$ and $A^{(B,3)}$ from the second production ($B \to CA_2DA_3$). ▷

Let us introduce the following relation: We write $A \prec B$ if A appears in the right part of some production for the nonterminal B.

Statement 2. For any grammar symbol A derived from the start symbol S there exists a chain $A \prec B_1 \prec \ldots \prec B_k \prec S$ such that all symbols B_i are nonterminals, $B_i \neq S$ and $B_i \neq B_j$ for all $i \neq j$.

Proof. By the assumption, A is derived from S. This means that there exists a chain $A \prec B_1 \prec \ldots \prec B_k \prec S$ for some non-terminals $B_1, \ldots, B_k$. Without loss of generality we can assume that this chain has the minimal length. Suppose that $B_s \neq B_{s'}$ for all $s \neq s',\ s, s' = i, \ldots, k$ and $B_{i-1} = B_l$ for some $l \in \{i, \ldots, k\}$. So we can construct a new chain $A \prec B_1 \prec \ldots \prec B_{i-2} \prec B_l \prec \ldots \prec B_k \prec S$, which is shorter than the original chain $A \prec B_1 \prec \ldots \prec B_k \prec S$. This contradiction concludes the proof. ▷

The Statement 2 shows a way to create a test set that meets the criterion (PLL). Construct a chain $A \prec B_1 \prec \ldots \prec B_k \prec S$ by the following algorithm:

1. For the symbol A, find a nonterminal set N_A such that for every nonterminal from N_A there exists its defining production with the symbol A in the right part. If $S \in N_A$, then the desired chain is constructed.
2. For each element $B_1 \in N_A$, find a nonterminal set N_{A,B_1} such that for each $B_2 \in N_{A,B_1}$ we have $B_2 \neq B_1$ and there exists a defining production of B_2 with B_1 in the right part. If $S \in N_{A,B_1}$, then the desired chain is constructed.

...

s. For each element $B_{s-1} \in N_{A,B_1,\ldots,B_{s-2}}$, find a nonterminal set $N_{A,B_1,\ldots,B_{s-1}}$ such that for each $B_s \in N_{A,B_1,\ldots,B_{s-1}}$ we have

$$B_s \notin \{B_1, \ldots, B_{s-1}\}$$

and there exists a defining production of B_s with B_{s-1} in the right part. If $S \in N_{A,B_1,\ldots,B_{s-2}}$, then the required chain is constructed.

...

By Statement 2, this algorithm finishes after creating the desired chain. Note that each chain $A \prec B_1 \prec \ldots \prec B_k \prec S$ corresponds to some derivations of sentential forms containing A. We can obtain these derivations by restricting occurrences of grammar symbols $A, B_1, \ldots, B_k$ in productions that define the relation $\prec$. Each resulting sentential form looks like $\alpha A \beta$, where α and β are some sequences of grammar symbols.

Let us define the function $first(x)$ returning a set of expansions of x:

1. If $x = t$ is a terminal, then $first(t) = \{t\}$.
2. If $x = A$ is a nonterminal, then $first(A) = \bigcup_{p=A\to\alpha} first(\alpha)$.
3. If $x = \alpha = X_1 \ldots X_n$ is a sentential form then let
$$C_\alpha = \{\beta X_2 \ldots X_n \mid \beta \in first(X_1)\}.$$
If X_1 can not expand to the empty sequence ε, then $first(\alpha) = C_\alpha$.
If X_1 have an empty expansion, then $first(\alpha) = C_\alpha \cup first(X_2 \ldots X_n)$.

In the course of calculation of the function $first$, recursion for nonterminals interrupts in the following way. If a recursive call occurs for the same nonterminal, then the result of this call is equal to the empty set.

Statement 3. For any terminal t that is acceptable for A as a first expansion symbol, the set $first(A)$ contains an expansion beginning with t.

Proof. It follows by the construction. ▷

If we have a sentential form $\alpha A\beta$ derived from the start symbol and a set $first(A)$, then we can construct a set of forms $\alpha\gamma\beta$, where $\gamma \in first(A)$. For any form of this kind we fix a terminal sequence derived from this form. By $\mathfrak{T}_A$ denote a set of fixed terminal sequences.

Statement 4. A positive test set $\bigcup_{A\in\mathcal{N}} \mathfrak{T}_A$ meets the criterion (PLL).

Proof. It follows from Statement 3 and construction of sets $\mathfrak{T}_A$. ▷

At the end of this subsection, we describe, how to select positive test sets meeting the criteria WPLR and PLR.

Let B be a nonterminal of a grammar G. By $\mathcal{D}_k(B)$ denote a set of sentential forms $\alpha B\beta$ derived from the start symbol such that theirs minimal derivation chains include at most k occurrences of every production.

Let $\pi = B \to \lambda \bullet X\mu$ be an item of the grammar G. Suppose that
$$first(\pi) = \{\lambda\gamma\mu \mid \gamma \in first(X)\}.$$
By definition, put $\zeta = \alpha B\beta \in \mathcal{D}_k(B)$. For each sentential form $\alpha\lambda\gamma\mu\beta$ with $\lambda\gamma\mu \in first(\pi)$ let us fix some terminal sequence derived from this form. By $\mathfrak{T}_{\pi,\zeta}$ denote a set of the fixed sequences. It is clear that each sequence from $\mathfrak{T}_{\pi,\zeta}$ is a sentence of the target language. Thus, $\mathfrak{T}_{\pi,\zeta}$ is a set of positive tests.

Statement 5. For each nonterminal $B \in \mathcal{N}$ of the grammar G let us fix some sentential form $\zeta_B \in \mathcal{D}_k(B)$. Then the positive test set
$$\bigcup_{\pi\in\mathfrak{P},\pi=B\to\ldots} \mathfrak{T}_{\pi,\zeta_B}$$
meets the criterion (WPLR); recall that $\mathfrak{P}$ is a set of all items of G.

Proof. It follows from Statement 3. ▷

Statement 6. There exists a number K such that for each $k > K$ a set

$$\bigcup_{\pi \in \mathfrak{P}} \bigcup_{\zeta \in \mathcal{D}_k(B)} \mathfrak{T}_{\pi,\zeta}$$

meets the criterion (PLR).

Proof. It is easy to show that if k tends to infinity, then $\mathcal{D}_k(B)$ tends to a set of all sentential forms $\alpha B \beta$. Therefore, there exists a number K such that for each $k > K$ the set $\mathcal{D}_k(B)$ covers all states of the FSM $\mathfrak{A}$ recognizing viable prefixes. Since we take the union for all items of G, all pairs (s_i, X) are also covered, where s_i is a state symbol and X is a grammar symbol. ▷

Negative Tests. As mentioned above, the most natural way to create negative tests is a mutation of sentences of the target language. Mutation of a sentence may be performed by replacement of some (may be empty) subsequence of this sentence by some "wrong" terminal sequence.

Let $\alpha = t_1 \dots t_n$ be a sentence of the target language. Let the terminal t_i be marked. Let us modify the sentence α as follows. Consider a set of terminal sequences $t_1 \dots t_i t' t_{i+1} \dots t_n$ for all $t' \in \mathfrak{N}_{t_i}$. Such a modification of the sentence α is called a *mutation of first kind* and is denoted by $\mathrm{mut}_1(\alpha, i)$. The mutation of first kind *inserts* "wrong" symbols into a sentence. Let us modify the sentence α by another way as follows. Consider a set of terminal sequences $t_1 \dots t_i t' t_{i+2} \dots t_n$ for all $t' \in \mathfrak{N}_{t_i}$. Such a modification of the sentence α is called a *mutation of second kind* and is denoted by $\mathrm{mut}_2(\alpha, i)$. The mutation of second kind *replaces* one of terminals by "wrong" symbols. Let us also define mutations of first and second kind for $i = 0$. In this case, a "wrong" symbol t' belongs to a set of symbols that are not acceptable as a first symbol, i.e.

$$t' \in \mathcal{T} \setminus \{t \mid \exists \beta = x_1 \dots x_n \in \mathfrak{L}_G, x_1 = t\}.$$

The mutation operations are defined as above: $\mathrm{mut}_1(t_1 \dots t_n, 0) = \{t' t_1 \dots t_n\}$, $\mathrm{mut}_2(t_1 \dots t_n, 0) = \{t' t_2 \dots t_n\}$.

By Statement 1, both mutation operations give us negative tests. Now we describe, how to construct a set of sentences with marked terminals. Next, we describe, how to select negative test sets meeting criteria (NLL$_1$) and (NLR$_1$).

Let $\pi = D \to B_1 \dots B_i \bullet B_{i+1} \dots B_n$ be an item of a grammar G.

Consider the following algorithm for construction of a set of sentences with marked terminals.

1. Let M_π be the empty set.
2. Construct a sentential form $\alpha D \beta \overset{*}{\Leftarrow} S$ derived from the start symbol as is described above. Let γ be a non-empty sequence from $first(B_{i+1} \dots B_n)$. Add all sentential forms $\alpha \gamma \beta$ with marked first symbol of the sequence γ to the set M_π. By the construction of the set $first(\dots)$, the first symbol of the sequence γ is a terminal. If the sequence $B_{i+1} \dots B_n$ have an empty expansion, then go to the step 3. Otherwise, M_π is constructed.

3. For each item $\pi' = X \to \lambda D \bullet \mu$, add all sentential forms from the set $M_{\pi'}$ to the set M_π. Here recursion interrupts as follows. If a recursive call occurs for the same item, then the result of this call is equal to the empty set.

Consider a grammar $\mathfrak{G}$ obtained from the grammar G as follows. Terminals and nonterminals of the grammar $\mathfrak{G}$ are the same as of G. If the grammar G contains a production $A \to B_1 \dots B_n$, then the grammar $\mathfrak{G}$ contains the production $A \to B_n \dots B_1$, where A is a nonterminal and B_i are grammar symbols. The grammar $\mathfrak{G}$ has only these productions.

It is clear that each sentence of the language generated by $\mathfrak{G}$ is an inversion of some sentence of the language generated by G.

Take an item $\bar{\pi} = D \to B_n \dots B_{i+1} \bullet B_i \dots B_1$ of $\mathfrak{G}$ corresponding to an item π of G. Consider the set $M_{\bar{\pi}}$ in $\mathfrak{G}$. Let $N_\pi = \{(X_1 \dots X_n, i) \mid (X_n \dots X_1, i) \in M_{\bar{\pi}}\}$. Each sentential form from N_π may be expanded to one or more target language sentences with marked terminals. By $\mathfrak{S}_\pi$ denote the obtained set of marked sentences. Now one can apply mutation operations of first and second kind to sentences of this set.

Statement 7. Let A be a nonterminal, let $\mathfrak{P}_A$ be a set of items $D \to B_1 \dots B_i \bullet B_{i+1} \dots B_n$ such that $B_{i+1} = A$. Then the negative test sets

$$\mathfrak{S}_j = \bigcup_A \bigcup_{\pi \in \mathfrak{P}_A} \mathrm{mut}_j(\mathfrak{S}_\pi), \quad j = 1, 2$$

meet the coverage criterion (NLL$_1$).

Proof. Suppose that there exists a derivation $S \overset{*}{\Rightarrow} \alpha t A \beta$.

The nonterminal A appears at some step of this derivation, i.e.

$$S \overset{*}{\Rightarrow} \gamma D \delta \overset{p}{\Rightarrow} \gamma B_1 \dots B_i A B_{i+2} \dots B_n \delta \overset{*}{\Rightarrow} \alpha t A \beta,$$

where $p = D \to B_1 \dots B_i A B_{i+2} \dots B_n$. By the assumption, a set $M_{\bar{\pi}}$ may be constructed for an item $\bar{\pi} = D \to B_n \dots B_{i+2} A \bullet B_i \dots B_1$ of $\mathfrak{G}$. Therefore, if the set $M_{\bar{\pi}}$ contains a sequence $\lambda A t \mu$ then the pair (A, t') with $t' \in \mathfrak{N}_t$ is covered.

Note that the original derivation of the sentence $\alpha t A \beta$ corresponds to the following derivation in the grammar $\mathfrak{G}$:

$$S \overset{*}{\Rightarrow} \delta' D \delta' \overset{p}{\Rightarrow} \delta' B_n \dots B_{i+2} A B_i \dots B_1 \gamma' \overset{*}{\Rightarrow} \beta' A t \alpha'.$$

If the sequence $B_i \dots B_1$ can not expand to the empty sequence ε, then there exists a sequence in $first(B_i \dots B_1)$ that starts with t, i.e. $M_{\bar{\pi}}$ contains a sentential form $\lambda A t \mu$, as desired. If $B_i \dots B_1$ may expand to ε, then the proof is similar. ▷

Statement 8. Negative test sets

$$\mathfrak{S}_j = \bigcup_A \bigcup_{\pi \in \mathfrak{P}_A} \mathrm{mut}_j(\mathfrak{S}_\pi) \qquad (j = 1, 2)$$

defined in Statement 7 meet the coverage criterion (WPLR$_1$).

Proof. Is similar to proof of Statement 7. $\triangleright$

Let $\pi = D \to B_1 \dots B_i \bullet B_{i+1} \dots B_n$ be an item of G. Note that at the second step of construction of M_π, we choose a sentential form $S \overset{*}{\Rightarrow} \alpha D \beta$. So, the set M_π depends on this sentential form. Hence, the set $\mathfrak{S}_\pi$ depends on this sentential form too. Let $\mathcal{D}_k(D)$ be a set of sentential forms $\alpha D \beta$ such that theirs minimal derivation chains include at most k occurrences of each production.

Statement 9. There exists a number K such that for each $k > K$ sets

$$\mathfrak{S}_{jk} = \bigcup_{\pi \in \mathfrak{P}} \bigcup_{\alpha D \beta \in \mathcal{D}_k} \mathrm{mut}_j(\mathfrak{S}_\pi) \qquad (j = 1, 2)$$

meet the coverage criterion (NLR$_1$).

Proof. It is easy to show that if k tends to infinity, then $\mathcal{D}_k(D)$ tends to a set of all sentential forms $\alpha D \beta$. Therefore, there exists a number K such that for each $k > K$ all states of the FSM $\mathfrak{A}$ recognizing viable prefixes are covered. So, all pairs (s_i, t') are also covered, where s_i is a state symbol and t' is a "wrong" terminal. $\triangleright$

4 Experimental Results

We implemented the following generators:

- A generator G_P that creates sets of positive tests for parsers meeting the criterion PLL.
- A generator G_N that creates sets of negative tests for parsers meeting the criteria WNLR$_1$ and NLL$_1$.

Both generators take a specification of a target language (namely, BNF-grammar) as an input. The generator G_P has parameters that allow to control test set size. The generator G_N has a parameter that allows to control kind of mutation.

Besides, for each generated negative test the generator G_N provides an information about location and kind of an error in the test. Thus, if the parser under test can output proper error diagnostics, then one can use the generated negative tests in order to test correctness of the parser's error diagnostics.

We applied the generators to generate tests for the following programming languages:

- C – an extension of ANSI C implemented in the GCC compiler (see [19]);
- mpC – a high-level parallel language, an extension of ANSI C (see [21]);
- Java (version 1.4);
- J@va – a specification extension of Java designed in the Institute for System Programming of Russian Academy of Sciences (see [3]).

Table 1 reports some properties of the used BNF-grammars.

Tables 2, 3, 4 report some properties of generated test sets.

Table 1. Properties of grammars used for test generation

Property	**C**	**Java**	**J@va**
Number of non-terminals	70	135	174
Number of tokens	93	101	140
Minimal cardinality of $\mathfrak{N}_t$	1	19	34
Maximal cardinality of $\mathfrak{N}_t$	93	101	140
Average cardinality of $\mathfrak{N}_t$	68	79	119

Table 2. Number of generated tests

Generator	**C**	**Java**	**J@va**
G_P	137307	137148	219221
G_N	43448	71943	145758

Table 3. Generation time

Generator	**C**	**Java**	**J@va**
G_P	19 m 30 s	19 m 51 s	43 m 08 s
G_N	3 m 01 s	7 m 14 s	19 m 25 s

Table 4. Average size of generated tests (bytes)

Generator	**C**	**Java**	**J@va**
G_P	58	68	112
G_N	219	220	266

The obtained tests for Java and J@va have been run on J@va-parser developed in the Institute for System Programming of Russian Academy of Sciences (see [3]). As a result, the positive tests have detected 8 errors in the parser. The positive tests for mpC have detected 12 errors in the mpC compiler. The generated tests for C have been run on the GCC compiler (see [19]). For 112 positive tests, the compiler get caught in an endless loop. The generated tests for C have been also run on C-parser developed in the Institute for System Programming of Russian Academy of Sciences (see [20]). The negative tests have detected 2 errors in the parser.

5 Conclusion

In this paper we proposed coverage criteria for positive and negative tests for parsers by using model-based and specification-based testing approach.

One of the main advantage of the negative tests generation technique described in the paper is that the technique guarantees negativeness of generated tests. This allows to perform testing without any reference parser.

We described algorithms for generation of tests sets that meet the proposed coverage criteria. We have developed corresponding generators of positive and negative tests. Tests generated for different programming languages have been applied to testing a number of parsers and compilers. As a result, errors in the systems under test have been detected.

So, experimental results are quite encouraging. They prove adequacy of the proposed coverage criteria. We believe that the technique described in the paper will be useful in commercial compiler testing projects.

References

1. Aho, A. V., Sethi, R., Ullman, J. D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (1986)
2. Beizer, B.: Software Testing Techniques. van Nostrand Reinhold (1990)
3. Bourdonov, I. B., Demakov, A. V., Jarov, A. A., Kossatchev, A. S., Kuliamin, V. V., Petrenko, A. K., Zelenov, S. V.: Java Specification Extension for Automated Test Development. Proceedings of PSI'01. LNCS **2244**. (2001) 301–307.
4. Demakov, A. V., Zelenova, S. A., Zelenov, S. V.: Testing of Parsers of Texts in Formal Languages. Programming Systems and Tools, **2**. Moscow (2001) 150–156 (in Russian)
5. DeMillo, R. A., Offut, A. J.: Constraint-Based Automatic Test Data Generation. IEEE Transactions on Software Engineering, **17**, No. 9. (1991) 900–910
6. Guilmette, R. F.: TGGS: A flexible system for generating efficient test case generators. (1999)
7. Harm, J., Lämmel, R.: Two-dimensional Approximation Coverage. Informatica Journal, **24**, No. 3. (2000)
8. Kossatchev, A. S., Petrenko, A. K., Zelenov, S. V., Zelenova, S. A.: Application of Model-Based Approach for Automated Testing of Optimizing Compilers. Proceedings of the International Workshop on Program Understanding. Novosibirsk (2003) 81–88
9. Lämmel, R.: Grammar testing. In Proc. of Fundamental Approaches Software Engineering, **2029**. (2001) 201–216.
10. Maurer, P. M.: Generating test data with enhanced context-free grammars. IEEE Software. (1990) 50–55
11. Maurer, P. M.: The design and implementation of a grammar-based data generator. Software Practice and Experience, **22**, No. 3. (1992) 223–244
12. McKeeman, W.: Differential testing for software. Digital Technical Journal, **10**, No. 1. (1998) 100–107
13. Offut, A. J., Lee, S. D.: An Empirical Evaluation of Weak Mutation. IEEE Transactions on Software Engineering, **20**, No. 5. (1994) 337–344
14. Offut, A. J., Untch, R. H.: Mutation 2000: Uniting the Orthogonal. Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries. San Jose, CA (2000) 45–55
15. Petrenko, A. K.: Specification Based Testing: Towards Practice. LNCS, **2244**. (2001) 287–300

16. Petrenko, A. K. et al.: Compiler Testing Based on the Formal Model of the Language. Preprint of the Keldysh Institute of Applied Mathematics, **45** (1992) (in Russian)
17. Purdom, P.: A Sentence Generator For Testing Parsers. BIT, **2** (1972) 336–375
18. Zelenov, S. V., Zelenova, S. A., Kossatchev, A. S., Petrenko, A. K.: Test Generation for Compilers and Other Formal Text Processors. Programming and Computer Software, **29**, No. 3. Moscow–New-York (2003) 104–111
19. GCC. `http://gcc.gnu.org/`
20. CTesK. `http://unitesk.com/products/ctesk/`
21. The mpC Programming Language Specification. The Institute for System Programming of Russian Academy of Science. `http://www.ispras.ru/~mpc`

Testing from Algebraic Specifications: Test Data Set Selection by Unfolding Axioms

Marc Aiguier[1], Agnès Arnould[2], Clément Boin[1],
Pascale Le Gall[1], and Bruno Marre[3]

[1] Université d'Évry-Val d'Essonne, LaMI CNRS UMR 8042,
523 pl. des Terrasses F-91025 Évry Cedex, France
{aiguier, cboin, legall}@lami.univ-evry.fr
[2] Université de Poitiers, SIC, CNRS FRE 2731,
SP2MI, F-86962 Futuroscope Cedex, France
arnould@sic.univ-poitiers.fr
[3] CEA/DRT/LIST/DTSI/SLA Saclay,
F-91191 Gif sur Yvette Cedex
bruno.marre@cea.fr

Abstract. This paper deals with test data set selection from algebraic specifications. Test data sets are generated from selection criteria which are usually defined to cover specification axioms. The unfolding selection criterion consists in covering the input domain of an operation using case analysis. The unfolding procedure can be iterated in order to split input domains of operations into finer subdomains. In this paper we propose to extend an unfolding procedure previously developed in [5, 19] that could only be performed on very low level, i.e. executable specifications. On the contrary, our new unfolding procedure can be applied to any positive conditional specification. We show that our unfolding procedure is sound (no test is added) and complete (no test is lost) with respect to the starting reference test data set.

Keywords: Specification-based testing, algebraic specifications, selection criteria, unfolding, proof tree normalization, conditional rewriting.

1 Introduction

Specification-based testing, or black-box testing, consists in the dynamic verification of the specification requirements. Moreover, formal specifications are of great help for this task since they allow the design of well-founded and powerful tools for test case generation and for test execution. Test cases are then automatically generated from selection criteria. These criteria are chosen by experts according to either the application domain or the criticity level. Generally, criteria for specification-based testing allow to cover the specification requirements (e.g. axioms, transitions or states). In order to provide a success/failure verdict (oracle problem), test execution tools apply test inputs and analyse the outputs by comparison with the expected results defined from the formal specification.

W. Grieskamp and C. Weise (Eds.): FATES 2005, LNCS 3997, pp. 203–217, 2006.

Several approaches have been proposed, each one depending on the choice of formalisms: labelled transition systems [15], model based specifications such as the B method, VDM or Z [10, 16], synchronous reactive languages as LUSTRE [20] and algebraic specifications [5, 12, 6, 2, 3, 14, 17, 18, 11, 9, 8]. In the framework of testing from algebraic specifications, decision procedures interpret test outputs such that the resulting verdicts fit on the notion of program correctness. Comparing test outputs with expected results may be a complex task when some information is missing (the oracle problem). Different observational approaches [7] have been proposed to cope with similar problems arising with specification refinement. Previous works [5, 12] and more particularly [4, 14, 2] provide a formal framework for a pure black-box testing from algebraic specifications. Test cases are observable formulas which can be computed by the program under test and interpreted as "true" or "false". Correctness of a program under test with respect to a specification is then defined up to some observational equivalence depending on the set of observable formulas.

In this paper, we are interested in the process of selecting test sets from algebraic specifications, more precisely from positive conditional algebraic specifications. Test data sets are generated from selection criteria which are usually defined to cover specification axioms. The unfolding selection criterion consists in covering the input domain of an operation using case analysis based on the form of the axioms. The unfolding procedure can be iterated in order to split input domains of operations into finer subdomains. In this paper we propose to extend an unfolding procedure previously developed in [5, 19] that could only be performed on very low level, i.e. executable specifications. On the contrary, our new unfolding procedure can be applied to any positive conditional specification.

A selection criterion has to be viewed as the coverage of some formulas which represent some test objectives, such as the axioms of the specifications. There are two main strategies to select test cases: one that performs any selection of test cases based on some deterministic choice or on a distribution on the considered input domain (random testing) and one that performs a selection of test cases in order to cover subdomains identified by a domain coverage (partition testing). In the latter case, subdomains partition the initial domain and correspond to the various cases addressed by the specification. Concerning random testing, it has been advocated by several works [6, 9] since either it is really easy to implement or it brings a quantitative evaluation of the testing process. The widely well-known drawback of random testing is the case of a subdomain with a low probability level but with a high probability level of failure rate. Within the framework of testing from algebraic specifications, such an unlikely subdomain arises with conditional axioms of the form $\varphi(X) \Rightarrow \psi(X)$ where X is a variable vector. If the subdomain making true the condition $\varphi(X)$ has a low probability, then random testing can miss the verification of $\psi(X)$ which is precisely required on this problematic subdomain [5, 8]. On the contrary, partition testing is based on a case analysis of the formula under test. The formula under test is preprocessed in order to reveal pertinent subdomains. For example, [10] translates formula under test into an equivalent disjunctive normal form, each conjunction representing a test subdomain. Another formula translation consists in applying a proof strategy such that the remaining lemmas represent a test subdomain [8]. [5, 19] have given importance to case analysis by unfolding specification axioms. It consists in splitting the input domain of an operation from specification axioms. Selection criteria based on axiom unfolding allow the tester to progressively refine the coverage domain in order to control the size of the resulting test set.

The paper is organized as follows. In Section 2, we recall standard notations about algebraic positive conditional specifications. In order to be as self-contained as possible, Section 3 gives relevant definitions of [14] concerning our framework of testing and defines selection criteria and their associated properties. In Section 4, we recall the previous unfolding procedure defined for a restricted class of conditional specifications, the executable ones, for which each computation has a unique normal form. Section 5 introduces an extension of this unfolding procedure allowing us to define a selection criterion for the class of all positive conditional specifications. Both unfolding procedures perform a case analysis on specification axioms defining the operations. We will show that both unfolding selection criteria perform at each step an adequate partition of the input domain insofar as both are sound (no test is added) and complete (no test is lost) selection criteria.

2 Preliminaries

An *(algebraic) signature* $\Sigma = (S, F, V)$ consists in a set S of sorts, a set F of function names each one equipped with an arity in $S^* \times S$ and a S-indexed sets of variables V. In the sequel, a function f with the arity $(s_1 \dots s_n, s)$ will be noted $f : s_1 \times \dots \times s_n \to s$. Given a signature $\Sigma = (S, F, V)$, $T_\Sigma(V)$ and T_Σ are both S-sets of *terms with variables in* V and *ground terms*, respectively, freely generated from variables and functions in Σ and preserving arity of functions. Using a standard numbering of the tree nodes by natural number strings, we can refer to positions in a term. Thus, given a term t, a *position* of t is a string ω in $\mathbb{N}$ which represents the path from the root of t to the subterm whose head function occurs at this position. This subterm is noted $t_{|_\omega}$. Given a position $\omega \in \mathbb{N}^*$ in a term t, $t[t']_\omega$ is the term obtained from t by substituting the subterm $t_{|_\omega}$ by t'. A *substitution* is any mapping $\rho : V \to T_\Sigma(V)$ that preserves sorts. They are naturally extended to terms with variables. *Σ-equations* are formulae of the form $t = t'$ with $t, t' \in T_\Sigma(V)_s$ for $s \in S$. A *positive conditional Σ-formula* is then any sentence of the form $\alpha_1 \wedge \dots \wedge \alpha_n \Rightarrow \alpha_{n+1}$ where each α_i is a Σ-equation $(1 \le i \le n+1)$. $Sen(\Sigma)$ is the set of all positive conditional Σ-formulae. Given a formula $\varphi \in Sen(\Sigma)$, $Var(\varphi)$ is the set of all variables occurring in φ. A *(positive conditional) specification* $SP = (\Sigma, Ax)$ consists in a signature Σ and a set Ax of positive conditional formulae often called *axioms*.

A *Σ-algebra* $\mathcal{A}$ is a S-indexed set A equipped for each $f : s_1 \times \dots \times s_n \to s \in F$ with a mapping $f^\mathcal{A} : A_{s_1} \times \dots \times A_{s_n} \to A_s$. A Σ-morphism μ from a Σ-algebra $\mathcal{A}$ to a Σ-algebra $\mathcal{B}$ is a mapping $\mu : A \to B$ such that for all $s \in S$, $\mu(A_s) \subseteq B_s$

and for all $f : s_1 \times \ldots \times s_n \to s \in F$ and all $(a_1, \ldots, a_n) \in A_{s_1} \times \ldots \times A_{s_n}$ $\mu(f^{\mathcal{A}}(a_1, \ldots, a_n)) = f^{\mathcal{B}}(\mu(a_1), \ldots, \mu(a_n))$. $Alg(\Sigma)$ is the category objects of which are all Σ-algebras. The set of ground terms T_Σ can be extended into a Σ-algebra by providing each function name $f : s_1 \times \ldots \times s_n \to s \in F$ with an application $f^{T_\Sigma} : (t_1, \ldots, t_n) \mapsto f(t_1, \ldots, t_n)$. Given a Σ-algebra $\mathcal{A}$, we note $\_^A : T_\Sigma \to A$ the unique Σ-morphism that maps any $f(t_1, \ldots, t_n)$ to $f^{\mathcal{A}}(t_1^A, \ldots, t_n^A)$. A Σ-algebra $\mathcal{A}$ is said *reachable* if $\_^A$ is surjective. Given a Σ-algebra $\mathcal{A}$, a Σ-interpretation in A is any mapping $\iota : V \to A$. They are naturally extended to terms with variables. A Σ-algebra $\mathcal{A}$ *satisfies* a Σ-formula $\varphi : \wedge_{1 \leq i \leq n} t_i = t'_i \Rightarrow t = t'$, noted $\mathcal{A} \models \varphi$, if and only if for every Σ-interpretation ι in A, if $\iota(t_i) = \iota(t'_i)$ then $\iota(t) = \iota(t')$. Given $\Psi \subseteq Sen(\Sigma)$ and two Σ-algebras $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{A}$ is *Ψ-equivalent to* $\mathcal{B}$, noted $\mathcal{A} \equiv_\Psi \mathcal{B}$, if and only if we have: $\forall \varphi \in \Psi$, $\mathcal{A} \models \varphi \Longleftrightarrow \mathcal{B} \models \varphi$. Given a specification $SP = (\Sigma, Ax)$, a Σ-algebra $\mathcal{A}$ is a *SP-algebra* if for every $\varphi \in Ax$, $\mathcal{A} \models \varphi$. $Alg(SP)$ is full full subcategory of $Alg(\Sigma)$, objects of which are all SP-algebras. A Σ-formula φ is a *semantical consequence* of a specification $SP = (\Sigma, Ax)$, noted $SP \models \varphi$, if and only if for every SP-algebra $\mathcal{A}$, we have $\mathcal{A} \models \varphi$. $SP^\bullet$ is the set of all semantical consequences. A sound and complete calculus for positive conditional specifications (i.e. $SP \models \varphi \Longleftrightarrow SP \vdash \varphi$) is defined by the following inference rules:

$$\textbf{Trans.}\frac{SP \vdash \bigwedge_{1 \leq i \leq m} \alpha_i \Rightarrow t = t' \quad SP \vdash \bigwedge_{1 \leq i \leq m} \alpha_i \Rightarrow t' = t''}{SP \vdash \bigwedge_{1 \leq i \leq m} \alpha_i \Rightarrow t = t''} \qquad \textbf{Sym.}\frac{SP \vdash \bigwedge_{1 \leq i \leq m} \alpha_i \Rightarrow t = t'}{SP \vdash \bigwedge_{1 \leq i \leq m} \alpha_i \Rightarrow t' = t}$$

$$\textbf{Ref.}\frac{}{SP \vdash t = t} \qquad \textbf{Context}\frac{SP \vdash \bigwedge_{1 \leq i \leq m} \alpha_i \Rightarrow t_1 = t'_1 \ldots SP \vdash \bigwedge_{1 \leq i \leq m} \alpha_i \Rightarrow t_n = t'_n}{SP \vdash \bigwedge_{1 \leq i \leq m} \alpha_i \Rightarrow f(t_1, \ldots, t_n) = f(t'_1, \ldots, t_n)}$$

$$\textbf{Monotony}\frac{SP \vdash \bigwedge_{1 \leq i \leq m} \alpha_i \Rightarrow \alpha}{SP \vdash \bigwedge_{1 \leq i \leq m} \alpha_i \wedge \beta \Rightarrow \alpha} \qquad \textbf{Subst.}\frac{SP \vdash \bigwedge_{1 \leq i \leq m} \alpha_i \Rightarrow \alpha}{SP \vdash \bigwedge_{1 \leq i \leq m} \sigma(\alpha_i) \Rightarrow \sigma(\alpha)}$$

$$\textbf{Axiom}\frac{\varphi \in Ax}{(\Sigma, Ax) \vdash \varphi} \qquad \textbf{M.P}\frac{SP \vdash \bigwedge_{1 \leq i \leq m} \alpha_i \wedge u = v \Rightarrow \alpha \quad SP \vdash \bigwedge_{1 \leq i \leq m} \alpha_i \Rightarrow u = v}{SP \vdash \bigwedge_{1 \leq i \leq m} \alpha_i \Rightarrow \alpha}$$

3 Testing from Algebraic Specifications

3.1 A General Framework

The interpretation of test case submission as a success or failure is closely related to the notion of program correctness. More precisely, any test case submitted to a correct program should be analysed in a success case while ideally, an incorrect program should fail for at least a test case. In the context of testing from algebraic specifications, a first natural testing hypothesis is to suppose that programs are denoted by Σ-algebras. Hence, the test case interpretation can be defined from the satisfaction of some formulae. These formulae link input test data to expected results using operations, predicates or connectors of the specification. Thus, following our previous works [5, 14, 2], test cases are then denoted by formulae. As test case submission should yield a verdict, the formulae that represent test cases correspond to all formulae which can be interpreted by a computation of the program as "true" or "false". These "executable" formulae are also called *observable.* In practice, observable formulae are ground formulae only involving equalities on some given sorts (for example, all sorts provided with an equality predicate within the programming language).

Let $SP = (\Sigma, Ax)$ be a positive conditional specification and $Obs \subseteq Sen(\Sigma)$ any set of observable formulae. Let P be a program which is assimilated to a Σ-algebra of $Alg(\Sigma)$. It is sensible to assume that all values used by P are denoted by operation composition of Σ and that P has a functional behaviour with respect to the operations of Σ. Actually, our notion of correctness is based on this hypothesis. Indeed, under this minimal hypothesis, the program under test can be viewed as a simple reachable Σ-algebra which evaluates terms as the way the program computes the observable formulae. Then, test cases are observable formulae, which are successful for the program under test if and only if the Σ-algebra P satisfies them (i.e. executes them and interpret them as "true"):

Definition 1 (Test case and test set). *A* test case *is a formula of $SP^\bullet \cap Obs$. A* test set *T is a set of test cases. T is said to be* successful *for P if and only if $\forall \varphi \in T, P \models \varphi$.*

Correctness for dynamic testing is defined following an observational approach comparable to the ones used to define refinement of specifications: it is required that an algebra of the concrete specification is observationally equivalent to an algebra of the abstract specification. Here, by analogy, to be qualified as correct with respect to a specification, a program is required to be observationally equivalent to an algebra of the specification up to the observable formulae of Obs.

Definition 2 (Correctness). *P is* correct *for SP via Obs, denoted by $Correct_{Obs}(P, SP)$, if and only if there exists an algebra $\mathcal{A}$ in $Alg(SP)$ such that $\mathcal{A} \equiv_{Obs} P$.*

Note that our definition of test cases guarantees that any correct program is necessarily successful for the set of all test cases $SP^\bullet \cap Obs$. Indeed, $SP^\bullet \cap Obs$ is clearly the largest set of formulae which are both satisfied by all SP-algebras and executable by any program under test capable of interpreting formulae in Obs. This property is also called the unbiased property [5].

3.2 Selection Criterion

The challenge of testing then consists in managing (infinite) test sets. In practice, experts apply some selection criteria on a reference test set in order to extract a test set of size sufficiently reasonable to be submitted to the program. The underlying idea is that all test sets satisfying a considered selection criterion reveal the same class of incorrect programs, intuitively the ones corresponding to the fault model captured by the criterion. For example, the criterion called "uniformity hypothesis" postulates that any chosen value is equivalent to another one. For example, if a test set is given by $\{\sigma(\varphi) \mid \sigma : V \to T_\Sigma\}$ where φ denotes a formula (e.g. an axiom of SP) built over the variable x, then the uniformity selection criterion consists in choosing one arbitrary substitution $\sigma_0 : V \to T_\Sigma$ in order to select one test: $\sigma_0(\varphi)$.

A classical way to select test data with a selection criterion C consists in spliting a given starting test set T into a family of test subsets $\{T_i\}_{i \in I_{C(T)}}$ such that $T = \cup_{i \in I_{C(T)}} T_i$ holds. A test set satisfying such a selection criterion simply contains at least one test case for each non empty subset T_i. Intuitively, all test cases in T_i are supposed equivalent to reveal incorrect programs with respect the fault model captured by T_i. In practice, T represents a property φ (an operation, an axiom or any formula chosen as a testing objective) to be partially covered by testing. The sets T_i then represent subproperties of φ. Hence, the selection criterion C is a coverage criterion according to the way C is splitting the initial test set T into the family $\{T_i\}_{i \in I_{C(T)}}$. This is the method that we will use in the paper to select test data, known under the term of *partition testing.*

Definition 3 (Selection criterion). *A* selection criterion *C is a mapping*[1] *$\mathcal{P}(SP^\bullet \cap Obs) \to \mathcal{P}(\mathcal{P}(SP^\bullet \cap Obs))$. For a test set T, we note $|C(T)| = \cup_{i \in I_{C(T)}} T_i$ where $C(T) = \{T_i\}_{i \in I_{C(T)}}$.*

T' satisfies *C* applied to *T, noted by $T' \sqsubset C(T)$ if and only if:*
$\forall i \in I_{C(T)}, T_i \neq \emptyset \Rightarrow T' \cap T_i \neq \emptyset$.

A selection criterion consists in a mapping that splits test sets into families of test sets. The selection criterion is satisfied as soon as the considered test set contains at least a test case within each (non empty) test set of the resulting family. To be pertinent, a selection criterion should ensure some properties between the starting test set and the resulting family of test sets:

Definition 4 (Properties). *Let C, C' be two selection criteria and T, T' two test sets.*

- *C is said* sound for *T if and only if $|C(T)| \subseteq T$*
- *C is said* complete for *T if and only if $|C(T)| = T$.*
- *C is* partitioning *T if and only if $\forall i, j \in I_{C(T)}, i \neq j \Rightarrow T_i \cap T_j = \emptyset$*
- *C is said* finer than *C', denoted by $C \leq C'$ if and only if*
  $\forall T, T' \subseteq SP^\bullet \cap Obs, T' \sqsubset C(T) \Rightarrow T' \sqsubset C'(T)$.

[1] For a given set X, $\mathcal{P}(X)$ denotes the set of all subsets of X.

- *A family of selection criteria* $\{C_k\}_{k \in \mathcal{C}}$ *is said* iterative *if and only if* $\forall k \in \mathcal{C}, \exists k' \in \mathcal{C}, C_{k'} \leq C_k$.

The properties of soundness and completeness are essential for an adequate selection criterion: soundness ensures that test cases will be selected within the starting test set (i.e. no test is added) while completeness ensures that we capture all test cases up to the notion of equivalent test cases (i.e. no test is lost). When C is partitioning T, this means that the different test sets T_i are not superposed, and then, that one should choose at least a different test case for each T_i to build a test set satisfying C for T. An iterative family of selection criteria allows the tester to extend and to precise the process of test selection up to get a test set of convenient size. In order to obtain such an iterative family of selection criteria, it suffices to nest selection criteria. More precisely, if a selection criterion C_k builds a test set family $\{T_1, \ldots, T_{n_k}\}$ for a given test set T, and if a generic selection criterion C applied to any test set of this family, say T_i for example, provides the family $\{T_i^1, \ldots T_i^{n_i}\}$, then we can consider the selection criterion $C_{k'}$ defined by $C_{k'}(T) = \{T_1, \ldots, T_{i-1}, T_i^1, \ldots T_i^{n_i}, T_{i+1}, \ldots, T_{n_k}\}$. Moreover, if the intermediate selection criterion C is sound and complete, then clearly $C_{k'}$ is finer than C_k. We can systematically apply the selection criterion C to any test set T_i occurring in $C_k(T)$ for arbitrary index k in $\mathcal{C}$ and test set T. Of course, in practice, to define selection criteria in a generic and concise way, they will be defined on test sets given in an intentional definition. Sections 4 and 5 will define such iterative families of selection criteria based on an unfolding procedure which makes a case analysis of each occurrence of operation according to specification axioms.

3.3 A Reference Test Set

$SP^\bullet$ contains all the formulas which can be deduced from SP. In particular, it contains tautologies, redundant formulas... Without any additional knowledge about the structure of the starting test set or of its elements, the only way to select a finite test set is to use random technics. On the contrary, for many test methods (anyway all test methods used in practice), some strategy schemata are proposed to guide test selection. Our selection method takes inspiration from classic methods that partition (more generally that split into) the input domain of each function. Hence, it is quite natural to restrict ourselves to ground equations of the form $f(t_1, \ldots, t_n) = t_{n+1}$ where $t_i \in T_\Sigma$ $(i = 1, \ldots, n+1)$ and $f \in F$. f is the function under test applied to the input data $t_1, \ldots, t_n$ and t_{n+1} is the expected result. The reference test set is then defined:

Definition 5 (Reference test set). *Let* $SP = (\Sigma, Ax)$ *be a specification where* $\Sigma = (S, F, V)$ *is a signature. Let us define the set* $T_0(SP)$ *as follows:*

$$T_0(SP) = \{f(u_1, \ldots, u_n) = v \mid f \in F, u_1, \ldots, u_n, v \in T_\Sigma, SP \vdash f(u_1, \ldots, u_n) = v\}$$

This set is maximal with respect to the set of observable formulae $Obs = \{u = v \mid u, v \in T_\Sigma\}$. Indeed, we have $T_0(SP) = SP^\bullet \cap Obs$ (see Section 3.1).

Definition 6 (Input domain of operations). *Let $SP = (\Sigma, Ax)$ be a specification. Let $f : s_1 \times \ldots \times s_n \to s$ be an operation of Σ. The* domain of f, *noted $T_0(SP)_{|_f}$, is the set defined by:*

$$T_0(SP)_{|_f} = \{f(u_1, \ldots, u_n) = v \mid f(u_1, \ldots, u_n) = v \in T_0(SP)\}$$

Elements in $T_0(SP)$ are too numerous (often infinite) to be manageable. A subset of $T_0(SP)$ with a manageable size can be selected using an unfolding procedure taking advantage of the structure of axioms. Indeed, selecting equations in $SP^\bullet$ from positive conditional specifications SP requires to apply modus-ponens in order to remove equations in formula premises. Succinctly, our method reports under the following form: splitting the input domain of f into many subdomains according to the structure of the axioms defining f, called *test sets for* f, and choosing any input in each non-empty sub-domain. For instance, for an axiom of the form $\wedge_{1 \leq i \leq m} \alpha_i \Rightarrow f(x_1, \ldots, x_n) = y$, we can consider the following test set: $\{f(\sigma(t_1) \ldots, \sigma(t_n)) = \sigma(t) \mid \sigma : V \to T_\Sigma, \forall \varepsilon \in \{\alpha_i\}_{i \in 1..m},\ SP \models \sigma(\varepsilon)\}$.

Definition 7 (Test set for operations). *Let $SP = (\Sigma, Ax)$ be a specification where $\Sigma = (S, F, V)$ is a signature. Let $\mathcal{C}$ be a set of Σ-equations called* set of Σ-constraints. *Let $f : s_1 \times \ldots \times s_n \to s$ be an operation of Σ and $t_1, ..., t_n, t$ be terms of sorts $s_1, \ldots s_n, s$.*

A test set for $f(t_1, ..., t_n) = t$ with respect to $\mathcal{C}$, *noted $T_{\mathcal{C}, f(t_1, ..., t_n) = t}$, is the set of ground equations defined by:*

$$T_{\mathcal{C}, f(t_1, ..., t_n) = t} = \{\sigma(f(t_1, ..., t_n)) = \sigma(t) \mid \sigma : V \to T_\Sigma, \forall \varepsilon \in \mathcal{C}\ SP \models \sigma(\varepsilon)\}$$

4 The Selection Criteria Based on Axiom Unfolding in LOFT

In this section, we formalize the problem of test selection from algebraic specifications such as implemented in the test selection tool LOFT [5, 19].

4.1 The Unfolding Procedure in LOFT

The unfolding procedure as implemented in the test selection tool LOFT assumes that any conditional positive specification SP is actually presented under the form of a conditional rewrite system $\mathcal{R}$, that is any axiom is a conditional rewrite rule of the form $\wedge_{1 \leq i \leq m} \alpha_i \Rightarrow g(v_1, \ldots, v_n) \to v$ or can be directly transformed into a conditional rewrite rule [2]. Moreover, to ensure both soundeness and completeness of the unfolding procedure with respect to the

[2] In this last case, this can be achieved, for instance, by imposing that signatures $\Sigma = (S, F, V)$ are with constructors in $C \subseteq F$, and the equation $g(u_1, \ldots, u_n) = v$ satisfies $g \in F \setminus C$, $u_i \in T_\Omega(V)$ with $\Omega = (S, C, V)$, and all non-constructor operations in v are lesser than g according to a well-founded order $\succ$ on F. In this case, $g(u_1, \ldots, u_n) = v$ can be oriented from the left to the right (i.e. $g(u_1, \ldots, u_n) \to v$).

reference test set $T_0(SP)$, $\mathcal{R}$ is assumed to be both confluent and equipped with a well-founded ordering satisfying for all $\wedge_{1 \leq i \leq m}\ t_i = t'_i \Rightarrow t \rightarrow t'$ in $\mathcal{R}$ and all substitutions σ:

- $\sigma(t) > \sigma(t')$ and
- $\sigma(t) > \sigma(t_i)$ and $\sigma(t) > \sigma(t'_i)$ for all $1 \leq i \leq m$.

$\mathcal{R}$ is then reductive [13]. It is well-known that such rewrite systems ensure that both join conditional rewriting is decidable [13] and: $u \overset{*}{\leftrightarrow}_{\mathcal{R}} v \Longleftrightarrow SP \vdash u = v$.

The unfolding procedure developed here, has in input:

- a conditional positive specification $SP = (\Sigma, Ax)$ presented under the form of a reductive and confluent rewrite system, and
- a set Γ of couples (Σ-constraint set, test pattern).

The first set $\Gamma_0 = \{(\{f(x_1, \ldots, x_n) = y\}, f(x_1, \ldots, x_n) = y)\}$ where $x_i, y \in V$ $(1 \leq i \leq n)$.

The unfolding procedure is expressed by the two following inference rules [3]:

Reduce

$$\frac{\Gamma \cup \{(\mathcal{C} \cup \{r = s\}, f(t_1, ..., t_n) = t)\}}{\Gamma \cup (\sigma(\{(\mathcal{C}\}), \sigma(f(t_1, ..., t_n) = t))} \sigma \text{ most general unifier of } r \text{ and } s$$

Unfolding

$$\frac{\Gamma \cup \{(\mathcal{C} \cup \{r = s\}, f(t_1, ..., t_n) = t)\}}{\Gamma \cup \displaystyle\bigcup_{(c,\sigma) \in Tr(u_{|\omega}, r=s)} \{(\sigma(\mathcal{C}) \cup c, \sigma(f(t_1, ..., t_n) = t)\}} \ \omega \text{ a position in } u \text{ and } u \in \{r, s\}$$

where $Tr(u_{|\omega}, r = s)$ for r and s not unifiable, is the set of couples defined by:

$$\left\{ (\{\sigma(r[v]_\omega) = \sigma(s), \sigma(\alpha_1), \ldots, \sigma(\alpha_m)\}, \sigma) \,\middle|\, \begin{array}{l} \sigma \text{ mgu of } u_{|\omega} \text{ and } g(v_1, \ldots, v_n), \\ \displaystyle\bigwedge_{1 \leq i \leq m} \alpha_i \Rightarrow g(v_1, \ldots, v_n) \rightarrow v \in Ax \end{array} \right\}$$

As the definition of $Tr(u_{|\omega}, r = s)$ is based on the subterm relation and unification, this set is computable if the specification SP has a finite set of axioms. Hence, given an equation $r = s$ we have the selection criterion C_ε that maps any $T_{\mathcal{C}, f(t_1, \ldots, t_n) = t}$ to $\{T_{\mathcal{C} \setminus \{r=s\} \cup c, \sigma(f(t_1, \ldots, t_n) = t)}\}_{(c,\sigma) \in Tr(u_{|\omega}, r=s)}$ if $r = s \in \mathcal{C}$, $T_{\mathcal{C}, f(t_1, \ldots, t_n) = t}$ otherwise.

We write $\Gamma \vdash_U \Gamma'$ to indicate that Γ can be transformed to Γ' by applying one of the above inference rules.

[3] The unfolding procedure as defined in this section is actually a slight improvement of the original one given in [19]. Indeed, in [19], specifications are supposed to satisfy all the requirements succinctly given in the previous footnote 2.

Hence, an unfolding procedure is a program that accepts in input a positive conditional specification $SP = (\Sigma, Ax)$ and uses the above inference rules to generate a (finite or infinite) sequence

$$\Gamma_0 \vdash_U \Gamma_1 \vdash_U \Gamma_2 \vdash_U \Gamma_3 \vdash_U \dots$$

where $\Gamma_0 = \{(\{f(x_1,\dots,x_n) = y\}, f(x_1,\dots,x_n) = y)\}$ with x_i, y $(1 \leq i \leq n)$ are variables of Σ.

This unfolding procedure has been implemented in the test selection tool LOFT [5, 19] which has been developed in PROLOG. This enables it to benefit from a powerful resolution procedure of constraints. To illustrate this tool on an example, let us specify the *insert* operation with respect to the *List* constructors. This gives rise to the following specification written in the specification language CASL:

spec INSERT =
  NAT
then
  type *List* ::= [] | __::__(*Nat*; *List*)
  op *insert* : *Nat* × *List* → *List*
  ∀ *x*, *y*: *Nat*; *L*: *List*
  • $insert(x, []) = x :: []$ %(insert_empty)%
  • $x \leq y \Rightarrow insert(x, y :: L) = x :: y :: L$ %(insert_leq)%
  • $\neg\, x \leq y \Rightarrow insert(x, y :: L) = y :: insert(x, L)$ %(insert_g)%
end

It is obvious to transform this specification into an equivalent rewrite system by orienting each conclusion of axioms form the left to the right. With the recursive path ordering $>^{rpo}$ resulting from the precedence ordering: $insert > \_ :: \_ > []$, this rewrite system is reductive. Therefore, let us use LOFT to split the domain of *insert* operation by unfolding its axioms. Hence, we obtain three selection constraints corresponding to the three axioms of *insert*. The following LOFT command expresses that the ≤ predicate must not be unfolded while the *insert* operation must be unfolded once.

```
 ??- unfold_std([#(’__≤__:Nat,Nat->boolean’,0),#(’insert:nat,list->list’,1)],
insert(X,L1) = L2).

FINAL BINDING:
L1:list = empty
L2:list = __::__(X:nat,empty)
SOLUTION #1,     CPUTIME = 0

FINAL BINDING:
L1:nlist = __::__(_v0:nat,_v1:list)
L2:nlist = __::__(X:nat,__::__(_v0,_v1))
REMAINING CONSTRAINTS = { __≤__(X,_v0) = true }
SOLUTION #2,     CPUTIME = 0
```

```
FINAL BINDING:
L1:nlist = __::__(_v0:nat,_v1:nlist)
L2:nlist = __::__(_v0,_v2:nlist)
REMAINING CONSTRAINTS = { __≤__(X:nat,_v0) = false, insert(X,_v1) = _v2 }
SOLUTION #3,     CPUTIME = 0

GLOBAL TIME ELAPSED = 0 NUMBER OF SOLUTIONS = 3 yes
```

Note that LOFT uses a purely equational logic. Consequently, predicates are translated as boolean operations. Now, if we unfold twice the *insert* operation, only the subdomain #3 is split, because only the third constraint contains an equation with an occurrence of *insert*. We do not give the LOFT outputs here, but only the test cases produced to cover the last subdomain.

```
SOLVED CONSTRAINTS: BINDING:
L1:nlist = cons(_v0:nat,cons(_v1:nat,_v2:nlist)) L2:nlist =
cons(_v0,cons(_v1,_v3:nlist)) CONSTRAINTS = {
__≤__(X:nat,_v0) = false, __≤__(X,_v1) = false,
insert(X,_v2) = _v3 }

FINAL BINDING:
X:nat = 6
L1:nlist = __::__(1,__::__(0,__::__(2,__::__(0,__::__(9,empty)))))
L2:nlist = __::__(1,__::__(0,__::__(2,__::__(0,__::__(6,__::__(9,empty))))))

SOLUTION #5,     CPUTIME = 9
```

4.2 Soundness and Completeness

Test sets for operations are naturally extended to set of constraint sets as follows: Let Γ be a set of couples (Σ-constraint sets, test pattern)

$$T_\Gamma = \bigcup_{(\mathcal{C},f(t_1,\ldots,t_n)=t)\in\Gamma} T_{\mathcal{C},f(t_1,\ldots,t_n)=t}$$

The completeness result needs to assume that for any Γ resulting of the unfolding procedure, any $(\mathcal{C}, f(t_1,...,t_n = t)) \in \Gamma$, any $\varepsilon \in \mathcal{C}$ and any $\varphi \in Ax$, $Var(\varepsilon) \cap Var(\varphi) = \emptyset$. This can be easily obtained at each iteration of the unfolding procedure by renaming variables by fresh ones.

Therefore, for any specification SP presented under the form of a reductive and confluent rewrite system $\mathcal{R}$, both soundness and completeness of the unfolding procedure hold. Indeed, we have:

Theorem 8. *If $\Gamma \vdash_U \Gamma'$ then $T_\Gamma = T_{\Gamma'}$.*

(The proof is given in [1])

From Theorem 8, we have the expected result as a corollary:

Corollary 9 (Soundness and completeness). *If* $\Gamma_0 \vdash_U \Gamma_1 \vdash_U \Gamma_2 \vdash_U \ldots$ *then for all* $i < \omega$, $T_{\Gamma_i} = T_0(SP)_{|f}$.

By Theorem 8, the LOFT selection procedure is then sound and complete. Moreover, this selection procedure is iterative (see definition 3). However, the selection procedure provided by LOFT is not partitioning because specification axioms are not necessarily disjoint. More precisely, if two rewriting rules (obtained from two different axioms) can be applied simultaneously (at the same position of the same term of the same constraint), then the two resulting subdomains then have some common tests.

Recently, the LOFT selection procedure has been re-used in the GATeL tool [20, 21] that allows to produce test cases from LUSTRE specifications. LUSTRE is a synchronous language widely used in industry to build reactive systems. Hence, the GATeL tool unfolds LUSTRE equations to obtain test subdomains also defined by constraints. Therefore, these constraints are solved such that a test case is randomly built for each subdomain (if not empty).

5 Our Selection Criteria Based on Axiom Unfolding

In the last section, positive conditional specifications were equipped with strong conditions (presenting specifications by reductive and confluent rewrite systems) in order to obtain both soundness and completeness of the unfolding procedure. Here, the only required constraint, for the same results, is that specifications are positive conditional and that's all.

5.1 Unfolding Procedure

As in the previous section, the unfolding procedure developed here, has the following inputs:

- a conditional positive specification $SP = (\Sigma, Ax)$ (without any other constraints), and
- a set Γ of couples (Σ-constraint sets, test pattern).

The first set $\Gamma_0 = \{(\{f(x_1, \ldots, x_n) = y\}, f(x_1, \ldots, x_n) = y)\}$ where $x_i, y \in V$ $(1 \leq i \leq n)$.

The unfolding procedure is expressed by the two following inference rules:

Reduce

$$\frac{\Gamma \cup \{(\mathcal{C} \cup \{r = s\}, f(t_1, ..., t_n))\}}{\Gamma \cup \{(\sigma(\mathcal{C}), \sigma(f(t_1, ..., t_n)))\}} \sigma \text{ mgu of } r \text{ and } s$$

Unfolding

$$\frac{\Gamma \cup \{(\mathcal{C} \cup \{\varepsilon\}, f(t_1, ..., t_n) = t\}}{\Gamma \cup \bigcup_{(c,\sigma) \in Tr(\varepsilon)} \{(\sigma(\mathcal{C}) \cup c, \sigma(f(t_1, ..., t_n) = t))\}}$$

where $Tr(\varepsilon)$ for $\varepsilon = (r = s)$ (or symmetrically $s = r$) with r and s not unifiable, is the set of Σ-constraint sets defined by:

$$\left\{ \begin{array}{c} \left\{ (\{\sigma(r[v]_\omega) = \sigma(s), \sigma(\alpha_1), \ldots, \sigma(\alpha_m)\}, \sigma) \;\middle|\; \begin{array}{c} \sigma \text{ mgu of } r_{|\omega} \text{ and } g(v_1, \ldots, v_n), \\ (\bigwedge\limits_{1 \leq i \leq m} \alpha_i \Rightarrow g(v_1, \ldots, v_n) = v \in Ax \\ or \\ \bigwedge\limits_{1 \leq i \leq m} \alpha_i \Rightarrow v = g(v_1, \ldots, v_n) \in Ax) \end{array} \right\} \\ \cup \\ \left\{ (\{\sigma(r) = \sigma(s[v]_\omega), \sigma(\alpha_1), \ldots, \sigma(\alpha_m)\}, \sigma) \;\middle|\; \begin{array}{c} \sigma \text{ mgu of } s_{|\omega} \text{ and } g(v_1, \ldots, v_n), \\ (\bigwedge\limits_{1 \leq i \leq m} \alpha_i \Rightarrow g(v_1, \ldots, v_n) = v \in Ax \\ or \\ \bigwedge\limits_{1 \leq i \leq m} \alpha_i \Rightarrow v = g(v_1, \ldots, v_n) \in Ax) \end{array} \right\} \end{array} \right\}$$

As the definition of $Tr(r = s)$ is based on the subterm relation and unification, this set is computable if the specification SP has a finite set of axioms. Hence, given an equation ε we have the selection criterion C_ε that maps any $T_{\mathcal{C},f(t_1,\ldots,t_n)=t}$ to $\{T_{\mathcal{C}\setminus\{\varepsilon\}\cup c}, \sigma(f(t_1,...,t_n) = t)\}_{(c,\sigma)\in Tr(\varepsilon)}$ if $\varepsilon \in \mathcal{C}$, $T_{\mathcal{C},f(t_1,\ldots,t_n)=t}$ otherwise.

We can observe that the above unfolding procedure is strongly combinatory. This is the result of a complete unfolding on all subterms of both terms t and r. This ensures the completeness of the procedure with respect to the test set $T_0(SP)$ (see the next section). As we saw in section 4, this combinatory can be less when dealing with very low-level specifications (i.e. executable ones) [5, 19]. The interest here is that the unfolding procedure can be applied to any positive conditional specification with a finite set of axioms. No other requirement is imposed to ensure both completeness and soundness of the unfolding process. Hence, this procedure enables us to start functional testing at a more abstract level of specifications than executable ones.

5.2 Soundness and Completeness

We recall that test sets for operations are naturally extended to set of constraint sets as follows: Let Γ be a set of couples (Σ-constraint sets, test pattern)

$$T_\Gamma = \bigcup_{(\mathcal{C},f(t_1,\ldots,t_n)=t)\in\Gamma} T_{\mathcal{C},f(t_1,\ldots,t_n)=t}$$

As previously, the completeness result needs to assume that for any Γ resulting of the unfolding procedure, any $\mathcal{C} \in \Gamma$, any $\varepsilon \in \mathcal{C}$ and any $\varphi \in Ax$, $Var(\varepsilon) \cap Var(\varphi) = \emptyset$.

Theorem 9. *If $\Gamma \vdash_U \Gamma'$ then $T_\Gamma = T_{\Gamma'}$.*

(The proof is given in [1])

6 Conclusion

In this article, we have been interested in test set selection methods. We have focused on selection criteria for partition testing strategies. It consists in dividing the input domain of each operation into subdomains and then in selecting test cases from each of these subdomains. Some relevant properties (soundness, completeness, partition, iterative family) on these selection criteria have been presented. The unfolding selection criterion consists in covering the input domain of an operation using case analysis on specification axioms. The unfolding procedure can be iterated in order to split input domains of operations into finer subdomains. We have then extended an unfolding procedure previously developed in [5, 19] that could only be performed on executable specifications. Our unfolding procedure can be applied to any positive conditional specification. We have shown that both unfolding procedures are sound (no test is added) and complete (no test is lost) with respect to the starting reference test data set.

We still have ongoing researches concerning the definition of selection criteria for a larger class of specification including structuration primitives and this work takes inspiration from [17, 18]. Our goal is to be able to propose a framework of functional testing including selection criteria which would be devoted to specification coverage and usable at all steps of the software life cycle, and particularly, at the requirement step.

References

1. M. Aiguier, A. Arnould, C. Boin, P. Le Gall, and B. Marre. Testing from algebraic specifications: Test data set selection by unfolding axioms. Rapport LaMI 110, Université d'Évry-Val d'Essonne, 2005. ftp://ftp.lami.univ-evry.fr/pub/publications/reports.
2. A. Arnould and P. Le Gall. Test de conformité: une approche algébrique. *Technique et Science Informatiques, Test de logiciel, vol. 21, n° 9*, pages 1219–1242, 2002.
3. A. Arnould, P. Le Gall, and B. Marre. Dynamic testing from bounded data type specifications. In *Dependable Computing - EDCC-2*, volume 1150 of *LNCS*, pages 285–302, Taormina, Italy, Octobre 1996. Springer.
4. G. Bernot. Testing against formal specifications: a theoretical view. In *TAPSOFT'91, International Joint Conference on the Theory and Practice of Software Development*, volume 494 of *LNCS*, pages 99–119, Brighton UK, 1991. Springer.
5. G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, 1991.
6. Gilles Bernot, Laurent Bouaziz, and Pascale Le Gall. A theory of probabilistic functional testing. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 216–226. ACM Press, 1997.
7. M. Bidoit, R. Hennicker, and M. Wirsing. Behavioural and abstractor specifications. *Science of Computer Programming*, 25(2-3):149–186, 1995.
8. Achim D. Brucker and Burkhart Wolff. Symbolic test case generation for primitive recursive functions. In *Formal Approaches to Testing of Software*. 2004. to appear.
9. Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *International Conference on Functional Programming*, pages 268–279, 2000.

10. J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *FME'93: Industial-Strenth Formal Methods, First International Symposium of Formal Methods Europe*, volume 670 of *LNCS*, pages 268–284, Odense, Denmark, April 1993. Springer Verlag.
11. M. Doche and V. Wiels. Extended institutions for testing. In *AMAST'2000*, number 1816 in Lecture Notes in Computer Science, pages 514–528, 2000.
12. M.C. Gaudel. Testing can be formal, too. In *TAPSOFT'95, International Joint Conference, Theory And Practice of Software Development*, volume 915 of *LNCS*, pages 82–96, Aarhus, Denmark, 1995. Springer Verlag.
13. J.-P. Jouannaud and B. Waldmann. Reductive conditional term rewriting systems. In M. Wirsing, editor, *3rd IFIP Conference on Formal Description of Programming Concepts*. Elsevier Science Publishers, 1985.
14. P. Le Gall and A. Arnould. Formal specification and test: correctness and oracle. In *11th WADT joint with the 9th general COMPASS workshop*, volume 1130 of *LNCS*, pages 342–358. Springer, 1996. Oslo, Norway, Sep. 1995, Selected papers.
15. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.
16. B. Legeard, F. Peureux, and M. Utting. Controlling test case explosion in test generation from b formal models. *Softw. Test., Verif. Reliab.*, 14(2):81–103, 2004.
17. P. Machado. Testing from structured algebraic specifications. In *AMAST2000*, volume 1816 of *LNCS*, pages 529–544, 2000.
18. P. Machado and D. Sannella. Unit testing for casl architectural specifications. In *Mathematical Foundations of Computer Science*, number 2420 in LNCS, pages 506–518. Springer-Verlag, 2002.
19. B. Marre. Toward an automatic test data set selection using algebraic specifications and logic programming. In K. Furukawa, editor, *Eight International Conference on Logic Programming (ICLP'91)*, pages 25–28. MIT Press, 1991.
20. B. Marre and A. Arnould. Test sequences generation from LUSTRE descriptions: GATEL. In *ASE-00: The 15th IEEE Conference on Automated Software Engineering*, pages 229–237, Grenoble, September 2000. IEEE CS Press.
21. B. Marre and B. Blanc. Test selection strategies for lustre descriptions in gatel. In *MBT 2004 joint to ETAPS'2004*, volume 111 of *ENTCS*, pages 93–111, 2004.

Author Index