Kenji Suzuki   Teruo Higashino
Andreas Ulrich   Toru Hasegawa (Eds.)

Testing of Software and Communicating Systems

20th IFIP TC 6/WG 6.1 International Conference, TestCom 2008
8th International Workshop, FATES 2008
Tokyo, Japan, June 10-13, 2008
Proceedings

Springer

Volume Editors

Kenji Suzuki
The University of Electro-Communications
Tokyo 182-8585, Japan
E-mail: suzuki@cs.uec.ac.jp

Teruo Higashino
Osaka University
Department of Information Networking
Osaka 565-0871, Japan
E-mail: higashino@ist.osaka-u.ac.jp

Andreas Ulrich
Siemens AG, Corporate Research & Technologies CT SE 1
81730 Munich, Germany
E-mail: andreas.ulrich@siemens.com

Toru Hasegawa
KDDI R&D Laboratories Inc.
Saitama 356-8502, Japan
E-mail: hasegawa@kddilabs.jp

Library of Congress Control Number: 2008927735

CR Subject Classification (1998): D.2.5, D.2-3, C.2, F.3, K.6

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-68514-6 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-68514-2 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12273713 06/3180 5 4 3 2 1 0

Preface

This volume contains the proceedings of TESTCOM/FATES 2008, a joint conference of two communities: TESTCOM was the 20th edition of the IFIP TC6/WG6.1 International Conference on Testing of Communicating Systems and FATES was the 8th edition of the International Workshop on Formal Approaches to Testing of Software. TESTCOM/FATES 2008 was held at the Campus Innovation Center in Tokyo, Japan during June 10–13, 2008.

Testing is one of the most important techniques for validating and checking the correctness of communication and software systems. Testing, however, is also a laborious and very cost-intensive task during the development process of such systems. TESTCOM is a series of international conferences addressing the problems of testing communicating systems, including communication protocols, services, distributed platforms, and middleware. FATES is a series of international workshops discussing the challenges of using rigorous and formal methods for testing software systems in general. TESTCOM/FATES aims at being a forum for researchers, developers, and testers to review, discuss, and learn about new approaches, concepts, theories, methodologies, tools, and experiences in the field of testing of communicating systems and software.

TESTCOM has a long history. Previously it was called the International Workshop on Protocol Test Systems (IWPTS) and changed its name to the International Workshop on Testing of Communicating System (IWTCS) later. The previous conferences were held in Vancouver, Canada (1988); Berlin, Germany (1989); McLean, USA (1990); Leidschendam, The Netherlands (1991); Montréal, Canada (1992); Pau, France (1993); Tokyo, Japan (1994); Evry, France (1995); Darmstadt, Germany (1996); Cheju Island, Korea (1997); Tomsk, Russia (1998); Budapest, Hungary (1999); Ottawa, Canada (2000); Berlin, Germany (2001); Sophia Antipolis, France (2002); Oxford, UK (2004); Montréal, Canada (2005); New York, USA (2006) and Tallinn, Estonia (2007).

FATES also has its history. The previous workshops were held in Aalborg, Denmark (2001); Brno, Czech Republic (2002); Montréal, Canada (2003); Linz, Austria (2004); Edinburgh, UK (2005); Seattle, USA (2006), and Tallinn, Estonia (2007). TESTCOM and FATES became a joint conference in 2007.

In addition, FORTE 2008, the 28th IFIP International Conference on Formal Methods for Networked and Distributed Systems, was also held at the same place and time, as it did last year. Thus, the co-location of TESTCOM/FATES and FORTE fostered the collaboration between different communities. The common spirit of both conferences was underpinned by joint opening and closing sessions, invited talks, as well as joint social events.

TESTCOM/FATES received a reasonable number of submissions this year. Initially 58 abstracts were submitted, from which 42 research full papers were elaborated. The Program Committee finally selected 18 papers for presentation at the conference. Together with the invited presentations by Yutaka Yasuda from KDDI Corporation, Japan, and Paul Baker from Motorola, UK, they formed the contents of the proceedings. In addition, the conference contained another invited presentation on behalf of FORTE by Wolfram Schulte from Microsoft Research, USA. Moreover, presentations on industrial best practices and work-in-progress presentations rounded off the conference. A tutorial day preceded the conference.

It took tremendous efforts to organize this event. We would like to thank all the contributors for the success of TESTCOM/FATES 2008. In particular we are grateful to the Steering Committee of IFIP TESTCOM, the Program Committee, and all reviewers for their support in selecting papers of high quality. Without these contributions, these proceedings would not exist. We thank the International Communications Foundation, Support Center for Advanced Telecommunications Technology Research, Foundation, Microsoft Research, and KDDI Corporation for their financial support and Springer for publishing the proceedings. Last but not least, we would also like to express our thanks to the members of the Local Arrangements team from KDDI R&D Laboratories Inc., The University of Electro-Communications, Osaka University, and the Nara Institute of Science and Technology for their continuous support of this conference.

March 2008

Kenji Suzuki
Teruo Higashino
Andreas Ulrich
Toru Hasegawa

Conference Organization

General Chairs

Kenji Suzuki (The University of Electro-Communications, Japan)
Teruo Higashino (Osaka University, Japan)

Program Chairs

Andreas Ulrich (Siemens AG, Corporate Technology, Germany)
Toru Hasegawa (KDDI R&D Laboratories Inc., Japan)

TESTCOM Steering Committee

John Derrick (Chairman, University of Sheffield, UK)
Ana R. Cavalli (INT, France)
Roland Groz (Grenoble Institute of Technology, France)
Alexandre Petrenko (CRIM, Canada)

Program Committee

Bernhard K. Aichernig (TU Graz, Austria)
Antonia Bertolino (ISTI-CNR, Italy)
Gregor v. Bochmann (University of Ottawa, Canada)
Richard Castanet (LABRI, France)
Shing Chi Cheung (Hong Kong University of Science and Technology, China)
Sarolta Dibuz (Ericsson, Hungary)
Rachida Dssouli (Concordia University, Canada)
Khaled El-Fakih (American University of Sharjah, UAE)
Marie-Claude Gaudel (University of Paris-Sud, France)
Jens Grabowski (University of Göttingen, Germany)
Rob Hierons (Brunel University, UK)
Dieter Hogrefe (University of Göttingen, Germany)
Antti Huima (Conformiq Software Ltd., Finland)
Thierry Jéron (IRISA Rennes, France)
Ferhat Khendek (Concordia University, Canada)
Myungchul Kim (ICU, Korea)
Yoshihi Kinoshita (AIST, Japan)
Hartmut König (BTU Cottbus, Germany)
Victor V. Kuliamin (ISP RAS, Russia)
David Lee (Ohio State University, USA)

Bruno Legeard (Leirios, France)
Giulio Maggiore (Telecom Italia Mobile, Italy)
José Carlos Maldonado (University of San Carlos, Brazil)
Brian Nielsen (University of Aalborg, Denmark)
Manuel Núñez (Universidad Complutense de Madrid, Spain)
Tomohiko Ogishi (KDDI R&D Laboratories Inc., Japan)
Ian Oliver (Nokia Research, Finland)
Doron Peled (University of Bar-Ilan, Israel)
Fumiaki Sato (Toho University, Japan)
Ina Schieferdecker (Fraunhofer FOKUS, Germany)
Jan Tretmans (Embedded Systems Institute, The Netherlands)
Hasan Ural (University of Ottawa, Canada)
Mark Utting (University of Waikato, New Zealand)
M. Ümit Uyar (City University of New York, USA)
Margus Veanes (Microsoft Research, USA)
César Viho (IRISA Rennes, France)
Carsten Weise (RWTH Aachen, Germany)
Colin Willcock (Nokia Siemens Network, Germany)
Burkhart Wolff (ETH Zurich, Switzerland)
Nina Yevtushenko (Tomsk State University, Russia)

Local Organization

Tomohiko Ogishi (Chair, KDDI R&D Laboratories Inc.)
Takaaki Umedu (Osaka University)

Additional Reviewers

Tatsuya Abe
César Andrés
Gabor Batori
Sergiy Boroday
Ferenc Bozoki
Patryk Chamuczynski
Vianney Darmaillacq
Thomas Deiß
Alexandra Desmoulin
Rita Dorofeeva
Levente Eros
Andreas Griesmayer
Maxim Gromov
Jiale Huo
Iksoon Hwang
Sungwon Kang
Zhifeng Lai
Wissam Mallouli
Mercedes G. Merayo
Laurent Mounier
Masahiro Nakano
Helmut Neukirchen
T.H. Ng
Svetlana Prokopenko
Ismael Rodriguez
Rudolf Schlatte
Soonuk Seol
Adenilso da Silva Simão
Simone R.S. Souza
Toshinori Takai
Erik Tschinkel
Yu Wang
Hiroshi Watanabe
Bachar Wehbi
Martin Weiglhofer
Chunyang Ye
Benjamin Zeiß

Sponsoring Institutions

International Communications Foundation, Tokyo, Japan
Support Center for Advanced Telecommunications Technology Research, Foundation, Tokyo, Japan
Microsoft Research, Redmond, USA
KDDI Corporation, Tokyo, Japan

Table of Contents

All-IP Based Ultra 3G Network/Service Development in a Competitive Market

Yutaka Yasuda

KDDI Corporation
3-10-10, Iidabashi, Chiyoda-ku, Tokyo, Japan

Abstract. FMC (Fixed Mobile Convergence) is a key concept of NGN (Next generation Network) and KDDI announced the Ultra 3G concept based on all-IP network configuration, 3 years ago. Now, the Ultra 3G concept has evolved toward FMBC (Fixed Mobile Broadcast Convergence). This talk introduces how KDDI has been swiftly and reliably developing all-IP based networks and FMBC services.

Keywords: All-IP Networks, FMBC, Modular Development.

1 Introduction

Recently, telecommunication carriers have considered that convergence of fixed and mobile communications, known as FMC (Fixed Mobile Communication) to be a key in creating new services and increasing the market. KDDI, a telecommunication carrier in Japan, has provided various kinds of FMC services such as "LISMO", a comprehensive music service that links "auc mobile phones with PCs (January 2006). In the future, KDDI will continue to meet customer demands by creating more convenient and attractive services and content along with the business strategy of "Fixed Mobile and Broadcast Convergence (FMBC)", targeting a fusion among fixed and mobile communications and broadcasting.

Within the competitive market in Japan, it is important to realize rapid and reliable developments of FMBC networks/services. After introducing the Ultra 3G concept, this invited talk describes how KDDI has been achieving rapid and reliable developments especially for FMBC services, before subsequently concluding with details of how KDDI provides reliability after developments.

2 The Ultra 3G Concept

Based on Ultra 3G, a next generation communication infrastructure concept, KDDI aims to build a fixed-mobile integrated network which provides integrated services, including third generation mobile phone systems such as "EV-DO Rev.A" (launched in December 2006), wireless LAN, new wireless systems such as mobile WiMAX and beyond 3G mobile network, and wired accesses such as ADSL and FTTH (Hikari-One). The concept is illustrated in Fig. 1. A service control system will be developed to provide seamless services with various forms of access means complementing each other. This system will be constructed in compliance with IP Multimedia Subsystem (IMS)/Multimedia Domain (MMD), which are standardized by 3GPP and 3GPP2, and expected to be globally deployed in the future.

This future infrastructure will enable customers to enjoy high-speed data services and high-quality multimedia services anytime and anywhere, within an optimum communication environment, without being aware of differences between fixed and mobile communications.

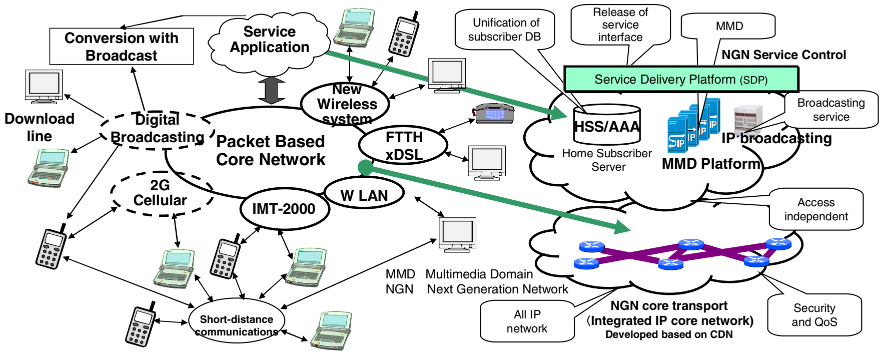


Fig. 1. Ultra 3 G Concept

3 Development Policies and Practices

There are two important issues relating to network and service developments, respectively. The first concerns the reliability of all-IP networks compared to the circuit based legacy network. Legacy SS7 protocol suites are designed to achieve its reliability with coordination between protocol layers, while conversely, IP technology on which VoIP (Voice over IP) is based is relatively independent from both upper and lower layers. This is clearly one of the advantages of IP technology, but, at the same time, needs some additional knowledge and operational skills to achieve the same level of reliability. Also, VoIP networks are more centralized compared to the legacy equivalents, meaning that one problem may have a wider impact on the entire network. The KDDI VoIP system has improved through experiences over the years.

The second issue is rapid service development. The competitive market demands the rapidity of FMBC network/service developments without any loss of reliability. The left of this section shows some recent KDDI practices to introduce modular development to software systems comprising services rapidly and reliably delivered.

3.1 Common Platform for Cellular Phone Software

KDDI cellular phone consists of chipsets provided by QUALCOMM and other peripheral devices. In order to achieve portability of application programs on them, KDDI has been engaged in creating a common platform, KCP (KDDI Common Platform), as shown in the left part of Fig. 2. But due to device specific interfaces, vendor dependent implementations still existed and as the result application program developments were time consuming and were error prone. In order to achieve more portability, KDDI extended it to the more modular platform, KCP+, as shown in the right part of Fig. 2. KCP+ is developed on MSM7500™ and BREW[1] which are the chipset and its application programming interface respectively. It makes all kinds of programs common to any cellular phones. These include not only application programs such as browsers, mailers but also middleware software and OS (Operating System) itself.

Providing KCP+ to vendors is expected to slash the development time and costs are expected to be reduced drastically. Besides vendors can also devote themselves to implementing their own functions related to UI (user interface) and/or design, etc. that differentiate their cellular phones to those of other vendors.

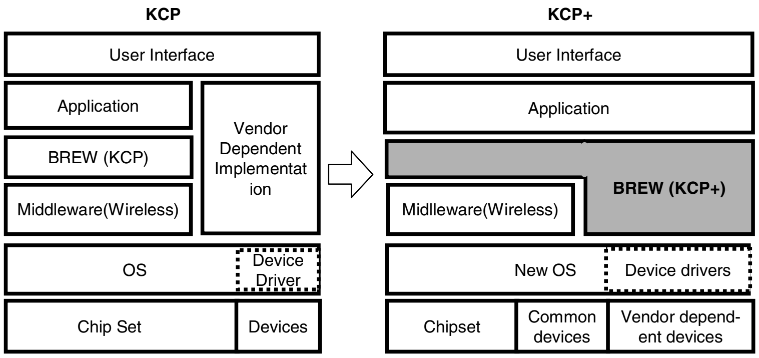


Fig. 2. Common Platform for Cellular Phone Software

3.2 SOA Based Development

BSSs (Business Support Systems) such as service, business and billing management systems used to be independently developed for individual services. Before the FMBC era, this style was not a problem; however, during the FMBC era, dependency among program modules of different BSSs has become problematic. Many new developments and updates which are performed due to rapidly changing business environments make them time consuming and error-prone, because a single change of an FMBC service has effects on several program modules of several BSSs. One possible solution is the introduction of an SOA (Service Oriented Architecture) based development.

As the initial step, KDDI has developed a BSS system that provides service management for Internet services over cellular phone networks. Before the system, known as ARIAL, was developed, service management systems were developed for individual services as shown in the left of Fig. 3. Conversely, functions common to all services such as “new order processes” and “call stop processes” are modeled as services of SOA-based development and are implemented in modular fashion using a BPM (Business Process Management) tool as shown in the right part of Fig.3. By introducing modular developments, a saving around 30% of the development time is possible while the throughput becomes 1.5 times as high as the previous BSS. Besides, failures that are localized to specific functions also improve reliability.

[1] BREW and MSM7500™ are trademarks of QUALCOMM.

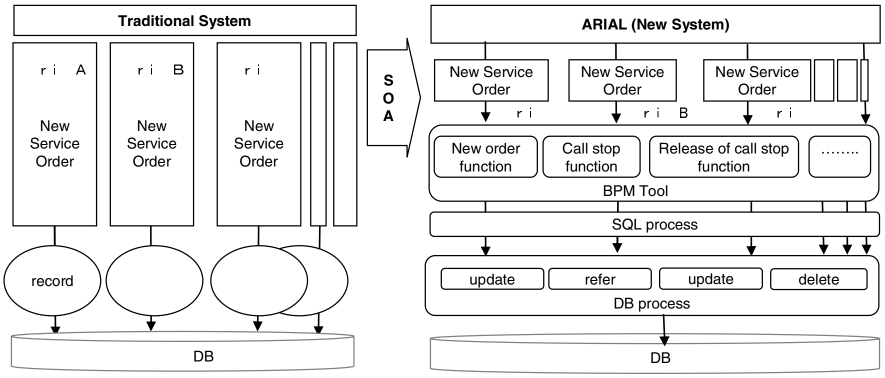


Fig. 3. SOA based Development

4 Conclusion

KDDI has been introducing FMBC core networks based on the Ultra 3G concept and FMBC services not only to survive the competitive environment, but also to enrich daily lives in Japan. This extended abstract shows the recent software development practices of KDDI. In addition KDDI is devoting itself to achieve flexible operations after developing FMBC network/service. The following are recent examples of techniques achieving flexible operations.

- Over-the-air software download techniques that distribute amendment of software bug and/or new software updates to cellular phones.
- Software-based device management techniques that collect and diagnose remote cellular phone logs.
- SLM (Service Level Management) techniques that measure the actively response times of servers and network equipments comprising services.
- Software radio techniques that enable programmability in wireless signal processing and reuse techniques that enable the software implementations of some wireless programmable devices to be ported to others at reasonable costs.

Models and Testing – A Recipe for Improved Effectiveness?

Paul Baker

Motorola Ltd.
Motorola's Corporate Software Excellence Team
Basingstoke, Hampshire, RG22 4PD, United Kingdom
Paul.Baker@motorola.com

Abstract. In an ongoing effort to reduce development costs in spite of increasing system complexity, Motorola has been a long-time adopter of Model-Driven Engineering practices. The foundation of this approach is the creation of rigorous models throughout the development process, thereby, enabling the introduction of automation into the development life cycle and supporting frameworks.

1 Introduction

In this talk we present Motorola's experiences over the past 15 years in using models [1][2] to improve testing effectiveness from defect prevention through to test configuration. We present the motivation(s) for using models and in particular their impact on product testing and quality. Specifically, we present and discuss cases and metrics where models have been used for:

- *Defect Prevention*. We present the approaches that have been employed for the early discovery of defects through the application of requirements verification techniques and the lessons learnt from doing so. In addition, we introduce recent work on trying to fix identified defects automatically through the application of inference [3][4][5];
- *Early testing of designs* through the use of co-simulation, as well as, experiences from previous usage of formal verification techniques[6];
- *Aiding test development and reuse*. In this case we discuss some of the issues encountered during general test development and elaborate why modeling concepts are useful in overcoming some of these obstacles [7];
- *Enabling test generation and automation.* It is well understood that models can lend themselves to test generation. We present some of the issues learnt and also their application within the context of existing modeling and test standards, such as UML, TTCN-3, MSC etc.;
- *Model-driven testability.* Here we introduce recent work on using models for the systematic understanding of testability concerns [8], which are crucial for cost effective test automation.

We also present lessons learnt from deploying model-related technologies and the different business strategies that have been used to effect their deployment, e.g. maturity models, aspects, and services. In doing so, we discuss the lessons learnt and directions/challenges moving forward with the perspective of a large testing technology portfolio.

References

[1] Baker, P., Loh, S., Weil, F.: Model-Driven Engineering in a Large Industrial Context – Motorola Case Study. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 476–491. Springer, Heidelberg (2005)

[2] Weigert, T., Weil, F., Marth, K., Baker, P., Jervis, C., Dietz, P., Gui, Y., Van de Berg, A., Fleer, K., Nelson, D., Wells, M., Mastenbrook, B.: Experiences in Deploying Model-Driven Engineering. In: Gaudin, E., Najm, E., Reed, R. (eds.) SDL 2007. LNCS, vol. 4745, Springer, Heidelberg (2007)

[3] Baker, P., Bristow, P., Burton, S., King, D., Jervis, C., Mitchell, B., Thomson, R.: Detecting and Resolving Semantic Pathologies in UML Sequence Diagrams. In: Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Lisbon, September 2005, pp. 50–59 (2005)

[4] Mitchell, B.: Resolving Race Conditions in Asynchronous Partial Order Scenarios. IEEE Transactions on Software Engineering, TSE-0039-0205 31(9), 767–784 (2005)

[5] Baranov, S., Jervis, C., Kotlyarov, V., Letichevsky, A., Weigert, T.: UML for Real Design of Embedded Real-Time Systems, pp. 323–342 ISBN 978-1-4020-7501-8

[6] Baker, P., Jervis, C.: Early UML Model Testing using TTCN-3 and the UML Testing Profile. In: TAICPART: Testing, Academic, and Industrial Conference, Practice and Research, IEEE Computer Society Press, Los Alamitos ISBN 0-7695-2984-4

[7] Baker, P., Dai, Z.R., Grabowski, J., Haugen, O., Schieferdecker, I., Williams, C.E.: Model-Driven Testing Using the UML Testing Profile. Springer, Heidelberg (2007)

[8] Baker, P., Jervis, C.: Early Model Testing and Testability Analysis using UML. Software Testing, Verification and Reliability (STVR) (submitted)

Runtime Verification of C Programs

Klaus Havelund

Jet Propulsion Laboratory
California Institute of Technology
Pasadena CA 91109, USA
Klaus.Havelund@jpl.nasa.gov

Abstract. We present in this paper a framework, RMOR, for monitoring the execution of C programs against state machines, expressed in a textual (non-graphical) format in files separate from the program. The state machine language has been inspired by a graphical state machine language RCAT recently developed at the Jet Propulsion Laboratory, as an alternative to using Linear Temporal Logic (LTL) for requirements capture. Transitions between states are labeled with abstract event names and Boolean expressions over such. The abstract events are connected to code fragments using an aspect-oriented pointcut language similar to ASPECTJ's or ASPECTC's pointcut language. The system is implemented in the C analysis and transformation package CIL, and is programmed in OCAML, the implementation language of CIL. The work is closely related to the notion of stateful aspects within aspect-oriented programming, where pointcut languages are extended with temporal assertions over the execution trace.

1 Introduction

The field of program verification is concerned with the problem of determining whether a program conforms to a specification. The pure verification problem consists of proving that all possible executions of the program conform to the specification. This is in general undecidable. Runtime verification is a less ambitious, but more feasible approach, just attempting to prove conformance of a single execution wrt. a specification. The specification can in this context be seen as a formalized oracle that can be used during testing, or it can become part of a fault protection system that runs in tandem with the program during its deployment, while triggering error correction code when non-conformance to the specification is detected.

The paper presents the runtime verification framework, RMOR (Requirement Monitoring and Recovery, pronounced *"armor"*), for monitoring C programs against state machines, using an aspect-oriented pointcut language to perform program instrumentation and connect the abstract events occurring in state machines with code fragments. The work has been partly driven by the context of embedded systems for planetary rovers and unmanned deep-space spacecraft as developed at NASA's Jet Propulsion Laboratory (JPL), where the majority of such code is written in C. The work presented reflects the following four observations. First, state machines appear a natural notation for programmers to apply, in contrast to for example temporal logic, or even regular expressions. Regular expressions are likely the most attractive of the succinct notations, but seem to be best suited for specifying “small” properties, whereas state machines support “big” properties involving many states. Second, although graphical editors for state machines are convenient, many programmers find textual programming-like notations convenient. Third, program instrumentation should be automated, connecting events to program points. Aspect-oriented programming has offered powerful pointcut languages for expressing such instrumentation. Fourth, most runtime verfication environments to date have been developed for Java, and C has been somewhat ignored. This is unfortunate since a majority of embedded software is written in C.

The RMOR language has inspirations from several sources. The language supports a notion of state machines directly influenced by RCAT (Requirement CApture Tool), a graphical state machine language language and editor [24,25]. That graphical state machine language is inspired by Linear Temporal Logic (LTL) and allows for liveness properties to be stated as well as safety properties. This is achieved by introducing special *error states* and *liveness states*. RCAT was developed to support property specification for the SPIN model checker [17][1] and was together with RMOR products of the *Reliable Software Systems Development* (RSSD) project, funded by NASA. Beyond RCAT, another direct inspiration has been the STATL specification language [12], from where a distinction between *consuming* and *non-consuming* transitions was borrowed (a consuming transition leaves the source state, whereas a non-consuming leaves a “token” – does not consume the token – in the source state when the transition is taken). Finally aspect-oriented programming, and specifically ASPECTJ [18] has strongly inspired the pointcut language driving program instrumentation. More recently, ASPECTC [2] has emerged as an aspect-oriented framework for C. This will be discussed further in Section 7.

A considerable amount of research has been invested in program monitoring systems by different communities within the last 5-10 years. The runtime verification community is concerned with program correctness [10,19,13,26,11,8]. This includes our own work [15,16,4]. Most of these efforts investigate more or less powerful temporal logics, with an exception in [11], which suggests the use of graphical UML state charts. The aspect-oriented programming community is investigating what is referred to as stateful aspects, where the pointcut language is extended with dynamic trace predicates [9,29,7,28,1]. These pieces of work are often extensions of ASPECTJ [18]. TRACEMATHCES [1] for example is an extension of ASPECTJ with regular expressions. JASCO [28] is a state machine solution for Java. An exception is ARACHNE [9], which performs runtime weaving into binary code of C programs. ARACHNE supports a form of trace predicates describing sequences of function calls, a limited form of regular expressions. The SLIC language [3] of the SLAM project is a specification language for C much resembling an aspect-oriented programming language, but simplified to support static verification as well as monitoring. The language supports state variables as well as access to function arguments and return values, but state machines have to be encoded using `enum` types, and the event language is not as comprehensive as a general purpose pointcut language. The program analysis communitiy has also contributed to this field [20] and the model checking community, which uses timed automata for testing, including monitoring [27,6].

[1] RCAT automata are by the RCAT tool translated into Büchi automata. RMOR can specifically monitor against such Büchi automata, although this is not the main purpose of the tool.

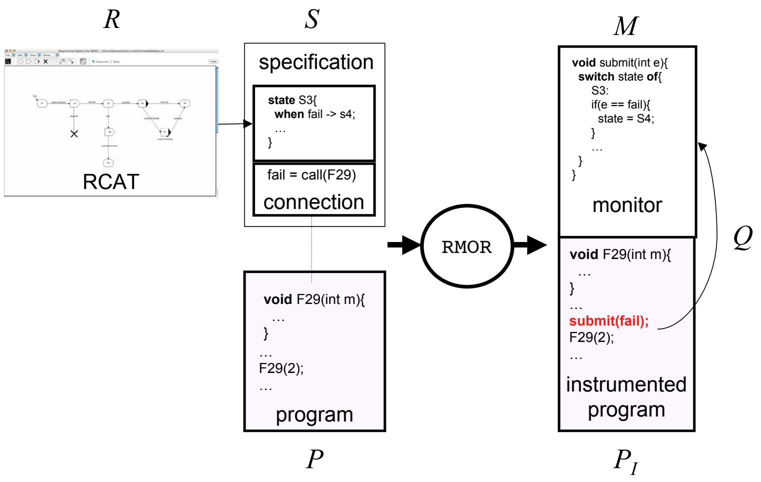
Our contributions to these efforts are: (i) to suggest a simple and natural textual programming notation for non-deterministic state machines integrated with an aspect-oriented pointcut language for program monitoring. This includes adapting the notions of error and live states from RCAT [24] for monitoring. With these concepts simple (finite trace) LTL properties can be stated naturally as state machines (the contribution of [24]) and monitored (our contribution). (ii) To implement such a system for C. Most embedded software is written in C. Most monitoring tools, however, have been focused on Java. The implementation uses CIL [21], which turns out to be very suited for developing source code instrumentation and runtime monitoring frameworks for C. (iii) To apply RMOR, resulting perhaps most importantly in feed-back from engineers wrt. usability.

The paper is organized as follows. Section 2 gives an overview of the RMOR architecture. Section 3 presents through examples the RMOR specification language. Section 4 summarizes the grammar of the specification language. Section 5 describes implementation details, including principles of the C code that is generated, as well as how the C code is instrumented. Section 6 presents case studies performed with RMOR. Finally Section 7 contains conclusions and outlines future work.

2 Overview of RMOR

The overall working of RMOR is illustrated in Figure 1. RMOR is a C program transformer, which inputs a pair consisting of a C program and a specification, and which outputs a C program that is *"armored"* by the specification. The specification is written in a textual format, that either can be programmed directly by a programmer, or it can be generated from a graphical state machine specification in the RCAT specification language. More specifically, RMOR takes as input a specification S in the RMOR specification language, and a C program P and produces a transformed program $Q = M + P_I$ which is the combination of a monitor M generated from the specification S, and an instrumented version P_I of P. P_I is P augmented with additional code that drives the monitor M. Executing the resulting program Q corresponds to executing the original program P, but with the monitor M constantly checking conformance to the specification. In case the specification is violated, an error message is printed on standard output, and in case specified, an error handling function is invoked.

The specification consists of two parts: the behavioral specification expressed as a set of state machines, or monitors as they are called, and an instrumentation specification. The state machines contain states and transitions between states that are triggered by the occurrence of events. Events are just abstract names. The instrumentation part specifies how these abstract event names connect to the code and is the basis for the automated program instrumentation. In the resulting instrumented code P_I, calls to the monitor M occur as calls of the `M_submit(int event)` function. Events are represented as integers. The calls of this function are automatically inserted by RMOR at locations defined by the instrumentation specification. The monitor M itself is a set of synthesized C functions that check conformance to the state machines and which are written into an `rmor.c` file that has to be compiled and linked together with the application. An `rmor.h` header file is also generated that containts the events and RMOR API prototypes



R
S
M
specification
state S3{ when fail -> s4; ... }
fail = call(F29)
connection
RCAT
void submit(int e){ switch state of{ S3: if(e == fail){ state = S4; } ... } }
monitor
RMOR
Q
void F29(int m){ ... } ... submit(fail); F29(2); ...
instrumented program
void F29(int m){ ... } ... F29(2); ...
program
P
P_I

Fig. 1. Overview of RMOR

(function signatures). The header file does not need to be included in the user program under normal circumstances, but can be as explained later in case the user program needs to explicitly refer to monitoring functions. The synthesized monitor uses a fixed amount of memory, hence it does not use dynamic memory allocation.

RMOR is implemented using CIL (C Intermediate Language), a C program analysis and transformation system [21]. CIL is programmed in OCAML, which consequently also is the programming language in which RMOR is implemented. CIL is a high-level representation along with a set of tools that permit easy analysis and source-to-source transformation of C programs. The CIL tool parses a C program and generates an abstract syntax tree annotated with type information. The generated tree represents a program in a clean subset of C. CIL is very robust and has been applied to for example the Linux kernel and GCC's C torture testsuite and processes not only ANSI-C programs but also those using Microsoft C or GNU C extensions. Consequently RMOR inherits the same characteristics. CIL provides a driver which behaves as either the gcc or Microsoft VC compiler and can invoke the preprocessor followed by the CIL application. The advantage of this script is that one can easily use RMOR with existing make files. The RMOR system extends CIL with approximately 2500 lines of code.

3 The RMOR State Machine Language

3.1 An Example C Program

In order to illustrate the specification language, consider the following toy application program about which properties will be formulated. The program, located in a file `main.c`, defines a collection of functions supporting uplink of data from a planetary rover to a space craft[2]:

[2] The example is fiction and does not represent an existing design.

```
char* header;
Connection open_connection(char* name) {...}
bool close_connection(Connection connection) {...}
void cancel_transmission(Connection connection) {...}
void write_buffer(Connection connection, int data) {...}
void commit_buffer(Connection connection) {...}
void acknowledge() {...}
void debug(char* str){...}

main(){
  Connection c1,c2;
  c1 = open_connection("connection1");
  c2 = open_connection("connection2");
  write_buffer(c1,100);
  commit_buffer(c1);
  close_connection(c1);
}
```

The program offers functions for opening and closing a connection between rover and space craft. While the connection is open data can be written to a data buffer, and finally committed, for which an acknowledgment is received and recorded with a call of the function `acknowledge`. A transmission can also be cancelled, not requiring further action. The program contains a global variable `header`, containing information about the current connection. The main program illustrates an example scenario.

3.2 Writing a Monitor

RMOR allows to specify properties about the execution order of *function calls* and *global variable accesses*. RMOR monitors safety properties, usually formulated as "*nothing bad happens*", as well as termination-bounded liveness properties "*something good eventually happens before program termination*". Safety properties are checked each time an event is submitted. Liveness properties are checked at the end of an execution when monitoring is terminated: at that point it is checked whether any outstanding events have not happened that were expected to happen according to the requirements represented by the monitors. In order to illustrate the RMOR notation a set of requirements will be modeled. Consider the following requirements R_1, R_2 and R_3 about the call-sequence of the functions in the above API. R_1*: "A connection is opened, accessed zero or more times, and subsequently either closed or canceled. An access is either a write operation or a commit operation";* R_2*: "The commit operation must be followed by an acknowledgement before any other operation can be performed, except a cancellation";* R_3*: "It is illegal to have more than one connection opened at any time".* These requirements can be formulated as several monitors, for example one for each requirement, or they can be grouped into one monitor as follows.

```
monitor UplinkRequirements {
  event OPEN = after call(main.c:open_connection);
```

```
    event WRITE = after call(main.c:write_buffer);
    event COMMIT = after call(main.c:commit_buffer);
    event ACK = after call(main.c:acknowledge);
    event CANCEL = after call(main.c:cancel_transmission);
    event CLOSE = after call(main.c:close_connection);

    initial state Closed {
       when OPEN -> Opened;
       when WRITE || COMMIT || ACK || CLOSE => error;
    }

    live state Opened {
      when COMMIT -> Committing;
      when CLOSE -> Closed;
      when ACK => error;
    }

    next state Committing {
      when ACK -> Opened;
    }

    super Active[Opened,Committing]{
      when CANCEL -> Closed;
      when OPEN => error;
    }
  }
```

The monitor introduces six events to be monitored and a state machine that any event sequence observed during program execution must conform to. Each event is defined by a predicate, denoting a set of statements in the program that satisfies it (a pointcut using aspect-oriented terminology), and a directive indicating whether the event should emitted before or after any statement satisfying the pointcut. As an example, the event OPEN is associated with the pointcut `call(main.c:open_connection)` which is matched by any *call* of the function `open_connection` defined in the file `main.c`. In the example program there are in fact two such calls. The `after` directive requires the event to be emitted to the monitor *after* each of these calls. It is in essence an instruction to RMOR to instrument the code by inserting a call to the monitor after these two calls. Similarly for the other events. Note that following aspect-oriented ideas, the program is oblivious to the fact that it is getting instrumented.

The state machine itself consists of three basic states: `Closed`, `Opened` and `Committing`. Each state is modeled as a named (the name of the state) block enclosed by curly-brackets `{ ... }` containing all its exiting transitions. The `Closed` state is the initial state, indicated with the *state modifier* keyword `initial`. In the `Closed` state, two transitions are defined. The first transition states that the event `OPEN` brings the monitor into the `Opened` state. Recall that an `OPEN` event occurs after any call of the function `main.c:open_connection`; The second transition states that if any of the events in the set $\{$WRITE, COMMIT, ACK, CLOSE$\}$ occurs, using the *or*-operator '||', it is it is regarded as an error – error is a special identifier denoting a built-in error state. The double arrow (=>) indicates a transition that leaves a token in the source state, in this case Closed, such that also future violations of this property is detected. Such a transition is called *non-consuming* since it does not consume the source token, as does the normal single arrow *consuming* transition (->). Recall that state machines are non-deterministic.

The Opened state is a *live* state as indicated by the modifier keyword live, meaning that this state must be left before program termination for this specification to be satisfied. This specifically means that either a COMMIT event or a CLOSE event must occur. An ACK event is not allowed to occur in this state. In the Committing state an ACK event must occur as the *next* observable event, indicated by the next state modifier keyword. This has as consequence that no other event can occur, except for a cancellation. The latter exception is a consequence of the super state named Active defined at the end of the monitor. This super state contains the two atomic states Opened and Committing and has two exiting transitions. This is a shorthand for these exiting transitions connected to each substate. The super state definition implies that when in any of the two sub-states it is regarded as an error if an OPEN event occurs, and a CANCEL event brings the monitor back to the initial Closed state.

3.3 Complex Pointcuts

Emissions of events to a monitor are inserted either before or after certain program locations (joinpoints) identified by pointcut expressions occurring after the '=' sign in event definitions. Pointcut expressions can, similar to AspectJ and AspectC [2], be used directly in event definitions, as we have seen above, or they can be defined and given names in explicit pointcut declarations, using Boolean combinators similar to those used on conditions. The following example illustrates this. Consider the additional requirement R_4*: "A write operation or an assignment to the* header *variable (collectively referred to as an update) should be followed by a commit operation before the connection is closed, unless the transmission is cancelled. This, however, only concerns* main updates *performed in the* main.c *file, ignoring updates made within any debugging function"*. In order to capture this requirement RMOR's poincut language is used to define the notion of a *main update*. The following monitor defines two poincuts, one used to define the other, and an event that is defined in terms of the latter pointcut.

```
monitor Symbols {
  pointcut Update = call(main.c:write*) || set(main.c:header);
  pointcut MainUpdate = Update &&
                        within(main.c) && !withincode(*debug*);

  event UPDATE = after MainUpdate;
}
```

The pointcut Update matches any program statement that is either: (i) is a call of a function defined in main.c and with a name matching the pattern write*, meaning having the name write as prefix, or (ii) is an update of the variable header declared in `main.c`. The file patterns (the part before and including the '`:`') are optional. Both file names and function/variable names can be indicated as patterns using "`*`" to represent any sequence of symbols. The pointcut `MainUpdate` refines the first pointcut to only concern those program statements occurring in the file `main.c` but not within any function with a name that contains the string `debug`. Finally, the event `UPDATE` is emitted after any main update. Note that this monitor contains no state machine and is purely introduced to define the pointcuts and the event. RMOR allows a monitor to import other monitors to access their pointcuts and events, and the next monitor imports the just presented one to access the `UPDATE` event, and also imports the original monitor to access further events.

```
monitor CommitUpdates {
  import Symbols;            // access UPDATE
  import UplinkRequirements; // access COMMIT, CANCEL and CLOSE

  state Start {
    when UPDATE => DoCommit;
  }

  live state DoCommit {
    when COMMIT -> Done;
    when CANCEL -> Done;
    when CLOSE -> error;
  }

  state Done{}
}
```

3.4 Error Handling

Our example program violates requirements R_2 (*a commit must be followed by an acknowledgment before anything else*), and R_3 (*no more than one open connection at a time*). Running the armored program produced by RMOR therefore causes two error messages to be printed on standard output. It is possible to provide a call-back handler function, which the monitor will call for each violation detected. This function must have the following name and type:

```
void handler(char *monitor, char *state, int kind) {
  ... user defined code ...
}
```

The first argument indicates the name (a string) of the monitor in which the error was encountered. The second argument indicates the name of the state it occurred in, and finally the third argument indicates the kind of error (a number between 0 and 2): (0) transition into an *error* state, (1) not leaving the *next* state before another event occurs, and (2) terminating in a *live* state. In order for errors to be handled by the handler function, the monitor must be declared with the `handled` modifier as follows:

```
handled monitor UplinkRequirements {
  ... as before ...
}
```

4 Elements of the RMOR Grammar

In this subsection elements of the grammar of RMOR are outlined, summarizing the concepts introduced in the example. A specification consists of a sequence of monitors[3]:

```
<specification> ::= <monitor>*
<monitor> ::=
      "handled"? "monitor" <monitor_name> "{" <declaration>* "}"
<declaration> ::=
      <import_decl> | <pointcut_decl> | <event_decl> |
      <state_decl> | <machine_decl>
```

An import declaration has the form:

```
<import-decl> ::= "import" <ident> ";"
```

Imports have the sole purpose of giving access to pointcuts and events from other monitors. Imports have no semantics at the state machine level. The grammar rules for pointcut declarations and pointcut expressions are as follows:

```
<pointcut_decl> ::= "pointcut" <ident> "=" <pointcut_expr> ";"
<pointcut_expr> ::=
    "call" "(" (<idpat1>":")?<idpat2> ")"
  | "set" "(" (<idpat1>":")?<idpat2> ")"
  | "within" "(" <idpat1> ")"
  | "withincode" "(" (<idpat1>":")?<idpat2> ")"
  | <ident>
  | <pointcut_expr> "&&" <pointcut_expr>
  | <pointcut_expr> "||" <pointcut_expr>
  | "!" <pointcut_expr>
  | "(" <pointcut_expr> ")"
<idpat1> ::= ("*" | letter|digit | "_" | "." | "-" | "/" )+
<idpat2> ::= ("*" | letter|digit | "_" )+
```

A poincut expression can specify a function call or a variable assignment, with `idpat1` indicating the name of the file in which the called function or updated variable is declared. The within pointcut matches statements occurring in files with names matching the argument, and withincode matches statements occurring within functions with names matching the argument. Beyond this, pointcuts can be referred to by name and conjoined with Boolean operators. An event declaration has one of two forms:

[3] The meta symbol `*` means zero or more occurrences, and `?` means zero or one occurrence.

```
<event_decl> ::=
    "event" <ident> "=" ("before "|" after") <pointcut_expr> ";"
  | "event" <ident> ("," <ident>) ";"
```

The event declarations shown this far are all of the first form. The second form is an abstract event declaration. It just introduces an event name that then can be used in state machines. However, no automated instrumentation is performed and it is the responsibility of the user to manually instrument the program to emit these events using the RMOR API. A state declaration can be of one of two forms:

```
<state_decl> ::=
    <state_modifier>* "state" <ident> "{" <transition>*  "}"
  | "super" <ident> "[" <ident> ("," <ident>)* "]" "{"
       <transition>*
    "}"
<state_modifier> ::= "initial" | "anytime" | "live" | "next"
<transition> ::= "when" <cond> ("->"|"=>") <ident> ";"
<cond> ::=
    "ANY" | <ident> | "!"<cond> | "(" <cond> ")"
  | <cond> "&&" <cond> | <cond> "||" <cond>
```

The first form is the basic state definition: a list of state modifiers, the keyword `state`, the name of the state, and a list of exiting transitions enclosed in a block. The second form is a super state definition, with the name of the super state and the list of sub-states in between `[ ... ]` brackets. These sub-states must be defined within the same monitor using the first form of state declaration. It is not possible to use another super state as a sub-state. The super state also has a list of exiting transitions. An `anytime` state always contains a token, even if an exiting transition is taken (state machines can be non-deterministic). The same effect can be obtained by defining all exiting transitions as non-consuming using the `=>` arrow. A condition is a Boolean expression over event identifiers and the `ANY` keyword, which in essence represents `true`, or “any” transition.

In an attempt to offer the possibility of grouping together state machines in one module it has been made possible to define several state machines inside a monitor. Such state machines cannot define any symbols or perform any imports:

```
<machine_decl> ::= "machine" <ident> "{" <state_decl>* "}"
```

RMOR offers in addition an API of functions with which the user application can interact with the monitors. These functions can for example be called from the handler. This includes functions for resetting and stopping monitors, submitting events, and printing monitor status for debugging purposes.

5 Implementation

OCAML [22] and its parser modules OCAMLLEX and OCAMLYACC were used to implement the parser for the RMOR specification language. The generated monitors in C utilize the SGLIB library [23], specifically double-linked lists for implementing sets. The program instrumentation module was, as already mentioned, implemented in OCAML on top of CIL [21].

5.1 Monitor Generation

The lexical scanning of RMOR specifications involves scanning of pointcut expressions, which is a well-known problem in aspect-oriented programming implementations, requiring the lexer to be state oriented, behaving differently in the *normal* and the *pointcut* state. OCAMLLEX allows for such state orientation, permitting us to apply a high-level parser generator for the task[4]. The program is parsed into an abstract syntax tree (AST), which is then processed for two purposes: translation of state machines to monitors, and instrumentation of the C code to emit events to the monitors (Section 5.2). The translator that produces state machines takes the AST as input and prints out the monitors in the file `rmor.c`. There are three constraints that specifically influence how RMOR is implemented: (i) monitors are allowed to be non-deterministic (a consequence for example of the => transition arrow, useful for monitoring), meaning that a state machine can be in more than one state at a given moment; (ii) dynamic memory allocation is not allowed since monitors should be able to monitor embedded flight code as part of a fault protection strategy, where only static memory allocation is allowed; (iii) a future extension of RMOR should allow for events to be parameterized with data values, and hence tokens in states should be able to carry values.

The first constraint requires each transition to produce a **set** of *next states*, computed from the set of *current states*. The second constraint requires that these different sets cannot be allocated dynamically on the fly as new sets are built. Instead, all states are allocated up front, and for each monitor is maintained three collections during next-state computation: a list of *free states*, a set of *current states*, and a set of *next states*. Each collection is modeled as a double-linked list. All states are initially stored in the free list. The monitor subsequently just moves states between these three sets when a new event arrives. At program termination it is checked that no tokens exist in live or next states. The motivation for representing sets as linked lists of records, and not as bitvectors, is the third constraint above, which requires data values to be part of state tokens in an extension of the tool. This will be further discussed in Section 7.

5.2 Instrumentation with CIL

The instrumentation module is implemented using CIL's object oriented visitor pattern framework. RMOR defines a class that subclasses a predefined visitor class, overriding a method for each kind of CIL construct that should be visited. CIL's visiting engine scans depth-first the structure of a CIL program and at each node executes the corresponding method in the user-defined visitor. The code below shows part of the visitor class defined for instrumentation. It overrides the method `vinst : instr -> instr list visitAction` that is applied to every basic instruction in the C program (essentially function calls and assignment statements, excluding composite statements, such

[4] The ASPECTJ parser is for example not constructed using a parser generator.

as loops). This function is expected to return a list of instructions, namely those that the visited instruction is replaced with. The body of the function computes a list of advices to be inserted (`advice_inserts`), that if not empty is split into those to be inserted before and after the instruction respectively.

```
class instrumentVisitor = object (self) inherit nopCilVisitor
  ...
  method vinst (i : instr) : instr list visitAction =
   ...
     let advice_inserts = match_instr ... i in
     if advice_inserts = [] then
       SkipChildren
     else
     begin
       let (before,after) = create_before_after advice_inserts in
       ChangeTo (before @ [i] @ after)
     end
end
```

The instrumentation consists of inserting calls of the function `M_submit(int event)` before or after joinpoints matching the pointcuts associated with events. The function `M_submit` stores the submitted event for later reference in the state machines, and subsequently calls the *next-state* function of each state machine.

6 Case Study

The Laboratory for Reliable Software at JPL has been developing a RAM File System (RAMFS) for use in future space missions. RAMFS will specifically be used as a backup file system on the next Mars Rover, MSL (Mars Science Laboratory), with launch date September-October 2009. MSL will be the biggest rover yet sent to Mars, and will be three times as heavy and twice the width of the Mars Exploration Rovers (MERs) that landed in 2004. RAMFS implements a thread-safe file system for flight systems in volatile memory (memory that requires a power supply to maintain the stored information). The main purpose of RAMFS is to provide a storage capability that can be used when the disk- or flash-file system is unavailable, e.g., when a spacecraft is in crippled mode, or in case there is not enough disk memory available. It is a project goal to apply various testing and verification technologies to establish confidence in the correctness of this file system [14]. Two different properties were formulated in RMOR and checked against the system. Both properties were satisfied, and malicious manual code modification caused them to be violated as expected.

Property 1: Matching Semaphore Accesses. The first property, called MatchSem, checks that semaphore operations are executed correctly: the semaphore must be reserved and released in strictly alternating order. The specification further states that once the semaphore has been reserved, it must eventually be released again. Reserving and releasing the semaphore is performed in the program respectively by calls of functions `osal_sem_take` and `osal_sem_give`. The monitor is defined as follows.

```
monitor MatchSem {
  event semtake = before call(osal_sem_take);
  event semgive = after call(osal_sem_give);

  state Start {
    when semtake -> HaveLock;
    when semgive -> error;
  }

  live state HaveLock {
    when semgive -> Start;
    when semtake -> error;
  }
}
```

Property 2: Protected Memory Updates. While the first property above states that the semaphore is used correctly, the second property states that memory accesses are correctly protected by the semaphore. That is, any access to memory must occur between a `semtake` and a `semgive`. Memory accesses come in two forms. The first are updates to the list of free memory through memory allocations with the function `ramfs_alloc_pages`, and memory freeing with the function `ramfs_free_pages`. The second are updates to the memory pages themselves through two functions `ramfs_update_entry` and `ramfs_update_header`. The monitor defines two pointcuts `free_list_update` and `page_update`, corresponding to these two kinds of calls.

```
monitor DataProtected {
  import MatchSem ;
  pointcut free_list_update =
      call(ramfs_alloc_pages) || call(ramfs_free_pages);
  pointcut page_update =
      call(ramfs_update_entry) || call(ramfs_update_header);
  event update = before free_list_update || page_update;

  state Unsafe {
    when semtake -> Safe;
    when update -> error;
  }

  state Safe {
    when semgive -> Unsafe;
  }
}
```

Observations. This case study demonstrated the ease with which a non-expert in RMOR was able to quickly learn the specification language and formulate properties.

Although not seen as a limitation during the exercise, the need for events to carry data values comes to the forefront in this example, specifically when it comes to the first property, that *the semaphore must be reserved and released in strictly alternating order*. The specification should ideally state that for a given semaphore *S*, its acquisition should be followed by a release of this same *S*.

A different case study was performed using RCAT from which RMOR monitors were generated as Büchi automata for an earlier version of RMOR. The case study was a rover controller for the Rocky 8 rover, a research vehicle that is used at JPL to develop, integrate, and demonstrate advanced autonomous robotic capabilities for future Mars missions. Since the specification language used was RCAT and since monitors were generated for an earlier version of RMOR, we shall not provide details about the example or the specifications. It suffices to say that the specification concepts used were similar to those of RMOR, and that the study supported the need for augmenting RMOR with the ability to express time constraints, and the ability to model conditions (predicates) on the state of the C program and use these as guards on transitions.

Concerning efficiency, the overhead naturally depends on the ratio with which monitored function calls and variable accesses are performed in the monitored application compared with the overall computation. Experiments showed that a single monitored call of a function with empty body results in an order of magnitude slow down of that call. Although monitored function calls usually constitute a small fraction of the overall computation, such overhead must be reduced using static analysis and algorithm optimizations.

7 Conclusions and Future Work

The following three aspects are important for acceptance of a technology such as RMOR: (i) *convenience* of the specification language; (ii) *expressiveness* of the specification language; (iii) *efficiency* of monitoring. A contribution of the paper is to illustrate the convenience of a state machine notation in combination with an aspect-oriented pointcut language. Concerning expressive power of the specification language, it currently only offers monitoring of propositional events. The notation should be extended with the ability to parameterize events with data values, corresponding to arguments in monitored functions, timers, and to generally enable C code to occur in the specification, for example allowing C code to be executed as a result of state machine transitions. In current work we are permitting this by directly extending ASPECTC [2] with state machines, utilizing ASPECTC's already existing pointcut language. This work is carried out using the SILVER extensible compiler framework [30]. Future work includes allowing user defined temporal logic operators as shorthands for state machines. Specifically, we plan to allow monitors to be parameterized with pointcuts. This will allow to define temporal operators/specification patterns within the language as is done in the EAGLE specification language [4], permitting very succinct specifications. We are furthermore exploring the possibility of adopting the more expressive rule-based logic RULER [5] as core logic, in which state machines form a special case. Efficiency can be obtained by application of static analysis to reduce code instrumentation.

Acknowledgements. Thanks goes to researchers at JPL. Margaret Smith designed the graphical RCAT language, which inspired the design of RMOR. Rajeev Joshi and Matthew Bennett participated in case studies using versions of RMOR. Gerard Holzmann supported the project by funding it as part of NASA's Reliable Software System Development project[5].

References

1. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittamplan, G., Tibble, J.: Adding Trace Matching with Free Variables to AspectJ. In: OOPSLA 2005, ACM Press, New York (2005)
2. AspectC, `http://research.msrg.utoronto.ca/ACC`
3. Ball, T., Rajamani, S.K.: SLIC: a Specification Language for Interface Checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research (2001)
4. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-Based Runtime Verification. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, Springer, Heidelberg (2004)
5. Barringer, H., Rydeheard, D., Havelund, K.: Rule Systems for Run-Time Monitoring: from Eagle to RuleR. In: Proc. of the 7th International Workshop on Runtime Verification (RV 2007), Vancouver, Canada. LNCS, vol. 4839, Springer, Heidelberg (2007)
6. Bensalem, S., Bozga, M., Krichen, M., Tripakis, S.: Testing Conformance of Real-Time Applications by Automatic Generation of Observers. In: Proc. of the 4th International Workshop on Runtime Verification (RV 2004). ENTCS, vol. 113, Elsevier, Amsterdam (2004)
7. Bockisch, C., Mezini, M., Ostermann, K.: Quantifying over Dynamic Properties of Program Execution. In: 2nd Dynamic Aspects Workshop (DAW 2005), Technical Report 05.01. Research Institute for Advanced Computer Science, pp. 71–75 (2005)
8. Chen, F., Roşu, G.: MOP: An Efficient and Generic Runtime Verification Framework. In: Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2007) (2007)
9. Douence, R., Fritz, T., Loriant, N., Menaud, J.-M., Sgura-Devillechaise, M., Südholt, M.: An Expressive Aspect Language for System Applications with Arachne. In: Proc. of the 4th international conference on Aspect-oriented software development, Chicago, USA, ACM Press, New York (2005)
10. Drusinsky, D.: Semantics and Runtime Monitoring of TLCharts: Statechart Automata with Temporal Logic Conditioned Transitions. In: Proc. of the 4th International Workshop on Runtime Verification (RV 2004), Barcelona, Spain. ENTCS, vol. 113, Elsevier, Amsterdam (2004)
11. Drusinsky, D.: Modeling and Verification using UML Statecharts, p. 400. Elsevier, Amsterdam (2006)
12. Eckmann, S., Vigna, G., Kemmerer, R.A.: STATL Definition. Reliable Software Group, Department of Computer Science, University of California, Santa Barbara, CA 93106 (2001)
13. Finkbeiner, B., Sipma, H.: Checking Finite Traces using Alternating Automata. In: Proc. of the 1st International Workshop on Runtime Verification (RV 2001). ENTCS, vol. 55(2), Elsevier, Amsterdam (2001)
14. Groce, A., Joshi, R.: Extending Model Checking with Dynamic Analysis. In: Logozzo, F., Peled, D., Zuck, L. (eds.) Proc. of Ninth International VMCAI conference (VMCAI 2008). LNCS, Springer, Heidelberg (2008)

[5] Part of the work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Part of the work was carried out at Kestrel Technology, under a contract with the National Aeronautics and Space Administration.

15. Havelund, K., Roşu, G.: An Overview of the Runtime Verification Tool Java PathExplorer. Formal Methods in System Design 24(2) (March 2004)
16. Havelund, K., Roşu, G.: Efficient Monitoring of Safety Properties. Software Tools for Technology Transfer 6(2), 158–173 (2004)
17. Holzmann, G.J.: The SPIN Model Checker, Primer and Reference Manual. Addison-Wesley, Reading (2004)
18. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
19. Kim, M., Kannan, S., Lee, I., Sokolsky, O.: Java-MaC: a Run-time Assurance Tool for Java. In: Proc. of the 1st International Workshop on Runtime Verification (RV 2001). ENTCS, vol. 55(2), Elsevier, Amsterdam (2001)
20. Martin, M., Livshits, B., Lam, M.S.: Finding Application Errors using PQL: a Program Query Language. In: Proc. of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, ACM Press, New York (2005)
21. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In: Proc. of Conference on Compilier Construction (2002)
22. OCAML, `http://caml.inria.fr/index.en.html`
23. SGLIB. A Simple Generic Library for C, `http://sglib.sourceforge.net`
24. Smith, M.: Requirements for the Demonstration Version of the Requirements Capture Tool (RCAT). JPL/RSS Technical Report, RSS Document Number: ESS-02-001 (2005)
25. Smith, M., Havelund, K.: Requirements Capture with RCAT. Jet Propulsion Laboratory, California Institute of Technology (submitted for publication, February 2008)
26. Stolz, V., Bodden, E.: Temporal Assertions using AspectJ. In: Proc. of the 5th International Workshop on Runtime Verification (RV 2005). ENTCS, vol. 144(4), Elsevier, Amsterdam (2005)
27. T-UPPAAL, `http://www.cs.aau.dk/~marius/tuppaal`
28. Vanderperren, W., Suvé, D., Augustina Cibrán, M., De Fraine, B.: Stateful Aspects in JAsCo. In: Gschwind, T., Aßmann, U., Nierstrasz, O. (eds.) SC 2005. LNCS, vol. 3628, Springer, Heidelberg (2005)
29. Walker, R., Viggers, K.: Implementing Protocols via Declarative Event Patterns. In: Taylor, R.N., Dwyer, M.B. (eds.) ACM Sigsoft 12th International Symposium on Foundations of Software Engineering (FSE-12), pp. 159–169. ACM Press, New York (2004)
30. Wyk, E.V., Bodin, D., Gao, J., Krishnan, L.: Silver: an Extensible Attribute Grammar System. In: Workshop on Language Descriptions, Tools, and Applications (2007)

Test Construction for Mathematical Functions[*]

Victor Kuliamin

Institute for System Programming
Russian Academy of Sciences
109004, B. Kommunistitcheskaya, 25, Moscow, Russia
kuliamin@ispras.ru

Abstract. The article deals with problems of testing implementations of mathematical functions working with floating-point numbers. It considers current standards' requirements to such implementations and demonstrates that those requirements are not sufficient for correct operation of modern systems using sophisticated mathematical modeling. Correct rounding requirement is suggested to guarantee preservation of all important properties of implemented functions and to support high level of interoperability between different mathematical libraries and modeling software using them. Test construction method is proposed for conformance test development for current standards supplemented with correct rounding requirement. The idea of the method is to use three different sources of test data: floating-point numbers satisfying specific patterns, boundaries of intervals of uniform function behavior, and points where correct rounding requires much higher precision than in average. Some practical results obtained by using the method proposed are also presented.

1 Introduction

In modern world computers help to visualize and understand behavior of very complex systems. Confirmation of this behavior with the help of real experiments is too expensive and often even impossible. To ensure correct results of such modeling we need to have adequate models and correctly working modeling systems. Constructing adequate models is very interesting problem, which, unfortunately, cannot be considered in this article in detail. The article concerns the second part – how to ensure correct operation of modeling systems. Such systems often use very sophisticated and peculiar algorithms, but in most cases they need for work basic mathematical functions implemented in software libraries or in hardware.

So, mathematical libraries are common components of most modeling software and correct operation of the latter cannot be achieved without correct implementation of basic functions by the former. In practice software quality is controlled and assured mostly with the help of testing, but testing of mathematical libraries usually is organized using simplistic ad hoc approaches and random

[*] This work is partially supported by RFBR grants 05-01-0099a, 06-07-89261a, and by contract 02.467.11.1006 with Russian Federal Science and Innovation Agency.

test data generation. Specifics of floating-point calculations makes construction of both correct and efficient implementations of functions, along with testing that they are actually correct, a nontrivial task. The main goal of this paper is to present a systematic method of test construction for mathematical functions implemented in software or hardware on the base of floating-point arithmetic stated in IEEE 754 standard [1].

The main ideas of the method proposed are to check correct rounding requirement and to combine three different sources of test data, which are targeted to catch common errors made by implementors of mathematical libraries.

- Floating-point (FP) numbers of specific structure. They include both boundary numbers like the greatest FP number (within certain precision), the smallest normalized number, etc., and numbers, which binary representations satisfy some specific patterns. In addition the closest floating-point numbers to the values of the reverse function in such points are calculated and also used as testing points for the direct function. Roughly speaking, a function is tested on points where its argument or its value are on the boundary or satisfy one of the chosen patterns.
- Boundaries of intervals, where the function under test behaves in uniform way. Several points on each of those intervals are also added to the test suite. Detailed rules for determining such intervals are presented below.
- Floating-point numbers, for which correct rounding of the function value requires much higher precision of calculations than average. These points are rather hard to seek and some methods for their search are presented in the section on test construction method.

The contribution of this article is the systematic description of the approach proposed, much more clear and concise then it was done in the paper [2] describing the starting phase of the underlying work. In addition this paper describes methods for searching FP numbers, for which correct rounding of the function value is hard, including the new one, *the integer secants method.* This research was started during the OLVER project [3] on formalization of LSB [4] standard requirements and conformance test development.

Before presenting the test construction method itself it is useful to recall some details of FP arithmetic, which are necessary for understanding details of the method. To make tests practically useful we also should clearly understand what precise requirements should be checked. So, the next section presents review of existing standards concerning mathematical functions working with FP numbers and analysis of these standards' requirements.

2 Standards' Requirements

To be able to check implementation of a function we should know how it should behave. Practically significant requirements on the behavior of functions on FP numbers can be found in several standards.

– Standards IEEE 754 [1] (also known as IEC 60559 [5]) and IEEE 854 [6] define representation of FP numbers, rounding modes, also describe basic arithmetic operations, comparisons, type conversions, square root function, and floating-point remainder.
– Standards ISO C [7] and POSIX [8] impose additional requirements on about 40 functions of real and complex variable implemented in standard C library.
– Standard ISO/IEC 10697 [9,10,11] gives more elaborated and precise set of requirements for elementary functions.

2.1 Floating-Point Numbers

Standards IEEE 754 and IEEE 854 define FP numbers based on various radices. Further exposition concerns only binary numbers, because other radices are used in practice rarely. Nevertheless, all the techniques presented can be extended to FP numbers with different radix, if it is necessary.

Representation of binary FP numbers is defined by two main parameters – n, the number of bits in the representation, and $k < n$, the number of bits used to represent an exponent. The interpretation of different bits is presented below.

– The first bit represents the sign of a number.
– The next k bits – from the 2-nd to the $k+1-$th – represent *the exponent* of a number.
– All the rest bits – from $k+2-$th to $n-$th – represent *the mantissa* or *the significand* of a number.

A number X with the sign bit S, the exponent E, and the mantissa M is expressed in the following way.

1. If $E > 0$ and $E < 2^k - 1$ then X is called *normalized* and is calculated with the formula $X = (-1)^S 2^{(E-2^{k-1}+1)}(1 + M/2^{n-k-1})$. Actual exponent is shifted to make possible representation of both large and small numbers. The last part of the formula is simply 1 followed by point and mantissa bits as the binary representation of X without exponent.
2. If $E = 0$ then X is called *denormalized* and is computed according to another formula $X = (-1)^S 2^{(-2^{k-1}+2)}(M/2^{n-k-1})$. Here mantissa bits follow 0 and the point. Note also, that this gives two zero values $+0$ and -0.
3. Exponent $2^k - 1$ is used to represent special values – positive and negative infinities (using zero mantissa) and *not-a-number* NaN (using any nonzero mantissa). Infinities represent results of operations that actually give mathematically infinite result or too big result to be represented as a floating-point number. NaN represents results of operations that cannot be considered consistently as finite or infinite, e.g. $0/0 =$ NaN.

IEEE 754 standard defines the following FP number formats: single precision ($n = 32$ and $k = 8$), double precision ($n = 64$ and $k = 11$), and extended double precision ($128 \geq n \geq 79$ and $k \geq 15$ (Intel processors use $n = 79$ and $k = 15$). In the next version of the standard quadruple precision numbers ($n = 128$ and $k = 15$) will be added.

2.2 IEEE 754 Requirements

Along with the representation of FP numbers IEEE 754 defines requirements to basic arithmetic operations on them (addition, subtraction, multiplication, and division), comparisons, conversions between different formats, square root function, and calculation of FP remainder [12]. Since results of these operations applied to FP numbers are often not exact FP numbers, it defines rules of rounding such results. Four rounding modes are defined: to the nearest FP number, up (to the least FP number greater than the result), down (to the greatest FP number less than the result), and to 0 (up for negative results and down for positive ones). If the result is exactly in the middle between two neighbor FP numbers, its rounding to nearest get the one having 0 as the last bit of its mantissa.

To make imprecise results more visible IEEE 754 defines a set of FP exception flags that should be raised in specific circumstances.

– Invalid flag should be raised if the result is NaN, while arguments of the operation performed are not NaNs.
– Divide-by-zero flag should be raised if the result is exactly positive or negative infinity, while arguments of the operation are finite.
– Overflow flag should be raised if the results' absolute value is greater than maximum FP number.
– Underflow flag should be raised if the result is not 0, while its absolute value is less than minimum positive normalized FP number.
– Inexact flag should be raised if the precise result is not an FP number, but its absolute value is inside the interval between minimum and maximum positive FP numbers.

2.3 Requirements of ISO C and POSIX

ISO C [7] and POSIX [8] standards provide description of mathematical functions of standard C library, including most important elementary functions (square and cubic roots, power, exponential and logarithm with bases $e, 2$ and 10, most commonly used trigonometric, hyperbolic functions and their reverse functions) of real or complex variables. Also some special functions are described – error function, complementary error function, gamma function, and logarithmic gamma function.

ISO C standard defines a set of points where the specified functions have exact well-known values, e.g. $\log 1 = 0, \cos 0 = 1, \sinh 0 = 0$. It also specifies situations where invalid and divide-by-zero flags should be raised, the first one – if a function is calculated outside of its domain, the second one – if the value of a function is precisely positive or negative infinity. These requirements are specified as normative for real functions and only as informative for complex functions.

POSIX slightly extends the set of described functions; it requires implementing Bessel functions of the first and the second kind of orders $0, 1$, and of an arbitrary integer order given as the second parameter. It also extends ISO C by specifying situations when overflow and underflow flags should be raised for functions in real variables. Additional POSIX requirement is that real functions having asymptotic $f(x) \sim x$ near 0 should return x for each denormalized argument value x. Note, that this would be in contradiction with IEEE 754 rounding requirements if they were applied to such functions.

Both standards do not say anything on precision of function calculation in general situation.

2.4 Requirements of ISO 10697

The only standard specifying some calculation precision for rich set of mathematical functions is ISO 10697 [9,10,11], standard on language independent arithmetic. It describes the following requirements to implementations of elementary functions in real and complex variables.

– Preservation of sign and preservation of monotonicity of ideal mathematical function where no frequent oscillation occurs. Frequent oscillation occurs where difference between two neighbor FP numbers is comparable with length of intervals of monotonicity or sign preservation. Trigonometric functions are the only elementary functions that oscillate frequently on some intervals. So, the standard defines *the big angle* – the least positive value, for which the last unit of mantissa or *ulp,* unit on the last place, becomes greater than $\pi/1000$. This value is about $2.8 \cdot 10^{13}$ for double precision. For arguments greater than the big angle preservation of sign and monotonicity does not required from implementations of trigonometric functions.
– The standard requires that rounding errors should not be greater than $0.5 - 2.0$ ulp, depending on the function implemented. Again, this in not required from implementations of trigonometric functions on arguments greater than the big angle. Note that precision 0.5 ulp is equivalent to the correct rounding to the nearest FP number.
– ISO 10697 requires to preserve evenness or oddity of implemented functions, and for this reason it does not support directed rounding modes – up and down. Only symmetric modes – to nearest and to zero – are considered as correct.
– The standards also specifies well-known exact values for all functions, extending ISO C requirements. In addition it requires from an implementation to preserve asymptotic of the implemented function in 0. FP numbers are distributed with different densities along the real axis, and their density increases while approaching 0 – the double precision number closest to 0 has value 2^{-1074}, while the one closest to 1 differs from it by 2^{-53}. For that reason, for example, an implementation of exp should return exactly 1 in some neighborhood of 0, and an implementation of sin should return x also in some neighborhood of 0.
– The last set of requirements imposed by ISO 10697 is concerned with natural inequalities between some functions, e.g. $\cosh(x) \geq \sinh(x)$, which should be preserved by their implementations.

So, ISO 10697 provides the most detailed set of requirements including requirements on calculation precision. Unfortunately, it has not yet recognized by practitioners and no widely-used library has declared correspondence with this standard. Maybe this situation will improve in future.

3 Analysis of Requirements

Analysis of existing standards shows that they are not fully consistent with each other and are usually restricted to some specific set of functions. Trying to construct some systematic description of general requirements based on significant properties of mathematical functions concerned with their computation one can get the following list.

– Exact values and asymptotic near them.
– Preservation of sign and monotonicity.
– Preservation of inequalities with other functions.
– Symmetries – evenness, oddity, periodicity, or more complex properties like $\Gamma(x+1) = x\Gamma(x)$.
– NaN results outside of function's domain, infinity results in function's poles, correct overflow and underflow detection, raising correct exception flags (extension of IEEE 754 and POSIX requirements).
– Preservation of bounds of function range, e.g. $-\pi/2 \leq \arctan(x) \leq \pi/2$, $-1 \leq \tanh(x) \leq 1$.
– Correct rounding according to natural extension of IEEE 754 rules and raising inexact flag on imprecise results.

In this list correct rounding requirement is of particular significance. It has the following important advantages.

– If we provide correct rounding, we immediately have almost all other properties in this list [13]. But if we want to preserve these properties without correct rounding, it requires much harder work, peculiar errors become possible, and thorough testing of such an implementation becomes very nontrivial and much harder task.
– Correct rounding provides results closest to the precise ones. If we have no correct rounding, it is necessary to specify how the results may differ from the precise ones, which is very rarely done in practice. It is supposed usually that correct rounding for sine function on large arguments is too expensive, but none of sine implementations used in practice explicitly declares its error bounds on various intervals. Most users usually don't analyze the results obtained from standard mathematical libraries, and are not competent enough to see the boundaries between areas where their results are relevant and the ones where they become irrelevant due to (not stated explicitly) calculating errors in standard functions. Correct rounding moves most of the problems of error analysis to the algorithms used by the applications, standard libraries become as precise as it is possible.
– Correct rounding requirement also implies almost perfect compatibility of different mathematical libraries and precise repeatability of calculation results of modeling applications on different platforms, which means very good

portability of such applications. This goal is also rather hard to achieve without such a requirement – one needs to standardize specific algorithms as it was made by Sun in mathematical library of Java 2. Also, strict precision specification is much more flexible requirement than standardization of algorithms.

High effort required to develop a function implementation and its resulting ineffectiveness are always mentioned as drawbacks of correct rounding requirement. However, good algorithms and techniques that help to resolve these issues are already known for a long time (e.g. see [14,15] for correct argument reduction for trigonometric functions). Work of Arenaire group [16] in INRIA on **crlibm** [17,18] library demonstrates that inefficiency problems can be resolved in almost all cases. So, now these drawbacks of correct rounding can be considered as not really relevant.

More serious issue is contradiction between correct rounding requirement and some other useful properties of mathematical functions. In each case of such a contradiction we should decide how to resolve it.

– Oddity and some other symmetries using minus sign or taking reciprocal values, like $\tan(\pi/2 - x) = 1/\tan(x)$, can be broken by directed rounding modes (up and down), while symmetric modes (to nearest and to 0) preserve them. In this case it is natural to prefer correct directed rounding if it is chosen, because usually such modes are used to get correct boundaries on exact results.
– Correct rounding can sometimes contradict with natural boundaries of function range, if these boundaries are not representable as precise FP numbers. For example, $-\pi/2 \leq \arctan(x) \leq \pi/2$ is an important property. It occurs that single precision FP number closest to $\pi/2$ is greater than it, so if we round arctangent values on large arguments to the nearest FP number, we get $\arctan(x) > \pi/2$, that can radically change the results of modeling of some complex systems. In this case we prefer to give priority to the bounds preservation requirement and do not round values of arctangent (with either rounding mode) to FP numbers out of its range.

So, further we consider test construction to check correct rounding requirement with 4 rounding modes specified by IEEE 754 with the single exception – when correct rounding breaks natural boundaries of a function range, we preserve these boundaries. The reader will see that even for implementations that do not satisfy these requirements such tests can also be very useful.

3.1 Table Maker Dilemma

An important issue related with correct rounding requirement is so called *table maker dilemma* [19,20]. It occurs when we need much higher precision of calculations to get correctly rounded value of a function. An example is the value of natural logarithm of a double precision FP number $1.613955\text{DC}802\text{F}8_{16} \cdot 2^{-35}$ (mantissa is represented in hexadecimals) equal to $-17.\text{F}02\text{F}9\text{BAF}6035\ 7\text{F}^{14}9\ldots_{16}$. Here

F^{14} means 14 digits F, giving with neighbor digits 60 consecutive units staying after a zero just after the double precision mantissa. This value is very close to the mean of two neighbor FP numbers, and to be able to round it correctly to the nearest FP number we need calculations with relative error bound about 2^{-113} while 0.5 ulp precision is only 2^{-53}.

Simple statistical model [19] presuming that mantissa's bits of function values are distributed uniformly and independently implies that 2^{n-k-1} FP numbers with one exponent can give $2^{n-k-1} \cdot 2^{-m+1}$ values with m consecutive equal bits – *bad cases,* in which table maker dilemma occurs. So, if $m \leq n-k$ (24 for single and 53 for double precision), then there may exist points where correct rounding of a function requires precision about 2^{m+n-k}.

Experiments (see below methods for bad cases search) shows that this is true in common case, on the intervals where a function has no singularities or simple asymptotic like $\cos x \sim 1$ or $\sin x \sim x$. But extraordinary bad cases also exists for most functions – m can be greater than 60 for double precision.

4 Test Construction Method

Test construction method proposed checks difference between correctly rounded value of a function and the value returned by its implementation in a set of test points. We prefer to have a rules of test point selection based only on the properties of the function under test and structure of FP numbers, and do not consider specific algorithms of implementations. This black box approach appears to be rather effective in revealing errors in practice, and at the same time it does not require detailed analysis of numerous and growing set of possible implementation algorithms and various errors that can be made in them.

Test points are chosen by the following rules.

1. **FP numbers of special structure:**
   First, natural boundary values in the set of FP numbers are taken as test points: $0, -0, \infty, -\infty$, NaN, the least and the greatest positive and negative denormalized and normalized numbers. This boundary values usually uncover errors in non-mature implementations. For example, the procedure recommended by Intel to calculated exponential function on Intel processors older than Pentium II gives NaN in $\pm\infty$, instead of 0 and $+\infty$ (see [21])

   Second, numbers with mantissa satisfying some specific patterns are chosen. Errors in an algorithm or an implementation often lead to incorrect calculations on some patterns. The notorious Pentium division bug [22] can be detected only on divisors having units as mantissa bits from 5-th to 10-th. In out practice an algorithm of square root calculation encoded in hardware made errors on about 10% of double precision numbers, square roots for which have mantissa of a form ****FFFFFFFFF, where * means an arbitrary hexadecimal digit. Pattern use for testing FP calculations is already described, e.g. in [23].

   So, several dozens different patterns are chosen, including all zeroes, all units, 0FFFF0000AAAA, and some others with alternating groups of digits

0, F, 5 (0101), and A (1010). On each exponent where the function under test is defined and is not degenerate (that is its values are in the range of representable numbers and do not all equal to one constant value or to function's argument) we take points with mantissas satisfying these patterns.

Third, two previous rules are used to get points where reverse function is calculated and pairs of closest FP numbers to its values are taken as test arguments for direct function. So, a function is tested in points, which satisfy some patterns, and in points where its value is closest to the same patterns.

2. **Boundaries of intervals of specific function behavior:**
   All singularities of the function under test, bounds of intervals of its non-overflow behavior, of constant sign, of monotonicity or simple asymptotic determine some partitioning of FP numbers. Boundaries of these intervals and several points on each of them are chosen as test points. They also usually reveal various errors.

   In case of frequent oscillation (e.g. for trigonometric and Bessel functions) this approach faces with an enormous number of intervals to be covered. To resolve this problem we choose only intervals where extreme values of the function are closest to its actual extreme values. For example, for trigonometric functions one can found the set of FP numbers closest to integer multiples of $\pi/2$ with the help of continued fractions [15]. This method gives about 2000 points for double precision numbers. On quarter-periods containing these points trigonometric functions approach to $0, \pm1$, or $\pm\infty$ much closer than on ordinary quatre-period. So, these quatre-periods are taken as characteristic intervals for those functions.
3. **FP numbers, for which calculation of correctly rounded function value requires higher precision:**
   Bad cases, which require more than M additional bits for correct rounding (the "badness"), are taken as test points. M is chosen equal to $n - k - 10$, which gives about 1000 test points on each exponent where the function under test is not degenerate. In addition some points with M near $(n - k)/2$ are chosen too, because some errors can be uncovered in not-very-bad cases ([24] gives an example of such an error in an implementation of square root). This rule adds test points helping to reveal calculation errors and inaccuracies of various nature.

Implementation of the method is rather straightforward. Test points are gathered into simple text files, each test point is accompanied with correctly rounded value of the function under test for each rounding mode (only two different values are required at most). Correctly rounded values are calculated with the help of multiprecision implementations of the same functions (e.g. in Maple or MPFR library [25]), taking higher precision to guarantee correct results. A test program reads test data, calls the function under test, and compares its result with the correct one. In case of discrepancy the difference in ulps is counted and reported. In addition the test program checks exception flags raising according to IEEE 754 rules extended to the function under test. So, test execution is completely automated.

The only not so easy task is to compute bad cases for a function. Methods that solve this problem are presented in the next section.

4.1 Bad Cases Computation

The following techniques can be used for bad case computation.

- Exhaustive search. In practice it can be applied only for single precision numbers (they have less than 2^{32} values). Almost 2^{64} double precision numbers and the need to calculate the function under test with high precision for each number make this technique unfeasible for higher precisions even with use of the most powerful modern computers.
- *Dyadic method* [24,26]. This method gets argument bad cases on the base of the function values in dyadic numbers in some points. It can be used to compute bad cases for algebraic functions – for square root [24,26] and for cubic root (no references).

  For example, if square root of FP number X is near the mean of two FP number, then $\sqrt{2^m N} \approx M + 1/2$, where we can consider M, N as integers between 2^{n-k-1} and 2^{n-k}, $X = 2^m N$, m is $n-k-1$ or $n-k-2$. By squaring this almost-equality we get integer equality $2^{m+2}N = (2M+1)^2 - j$, where j is a small integer number. So $(2M+1)^2 = j \bmod 2^{m+2}$ and $2M+1$ can be found as a square root of j modulo 2^{m+2}. The last task can be solved by consecutive computing square roots of j modulo $8, 16, 32, \ldots 2^{m+2}$ with the help of Henzel lifting (they give an infinite sequence tending to the dyadic square root of j).

  The same considerations can be applied to directed rounding bad cases [24] and to cubic root.
- *Reduced search* [19,27]. This method searches bad cases as FP-numbers grid nodes closest to the function under test graph. The function is approximated with high precision by linear polynomials on a set of intervals, and then its graph is substituted by straight line segments corresponding to those polynomials. For each segment the grid nodes closest to it are determined using 3-distance theorem. This theorem says that for a given sequence $x_0, x_1, ...$ of points on a circle, where x_0 is an arbitrary point and x_{i+1} is obtained from x_i by rotation on some angle α, not depending on i, there are always at most three different distances between neighbor points (see details in [27]). Intersections of a straight line segment with vertical lines of FP-numbers grid make up such a sequence being regarded modulo the distance between horizontal lines. 3-distance theorem thus help to perform only several simple operations for each FP point to learn whether it gives the value close to the segment or not. All suspicious points are stored and on the second phase the function is calculated in them with high precision to get really bad cases.
- *Lattice reduction*[28]. The idea is the same – to look for FP-numbers grid nodes closest to the function under test graph. In difference with the previous approach this one uses high precision approximations of the function under test by polynomials of small degree (but not linear). Then it searches

the closest approximation of this polynomial values in FP points by polynomials with integer coefficients with the help of *lattice reduction* – search of the shortest vector in a lattice of the same-degree polynomials with integer coefficients. Integer roots of found integer polynomial correspond to points of FP grid near the graph of the source polynomial (see details in [28]).

– *Integer secants method.* This is a new method proposed by the author. The idea is again to look for FP-numbers grid nodes closest to the function under test graph. But instead of search-based approaches more direct calculation is provided. this direct calculation is possible in a neighborhood of a point where the tangent $ax + b$ to the function graph has integer first coefficient a (or a is a reciprocal of an integer). Straight lines-secants parallel to this tangent on the distances of integer multiples of FP grid step cover all grid nodes. So, the closest grid nodes are near graph's intersections with these secants (see Fig. 1).

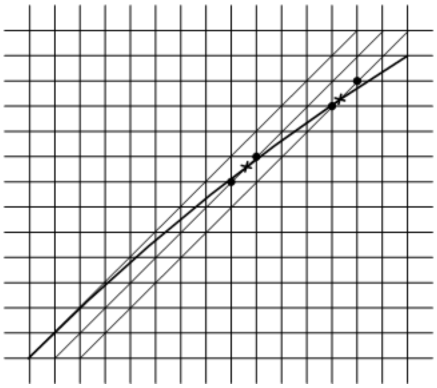


Fig. 1. Integer secants method

To compute such intersections consider the distance between the function graph and its tangent as a new function $F(x) = f(x) - ax - b$. This function has Taylor series in the point of contact starting from the square (or cubic) term, since it is the difference between Taylor series of the initial function f in the point of contact and the polynomial $ax + b$ of the contacting tangent. This series can be reversed in two steps – by computing a series that is a square (or cubic) root of this series, and then by reversing the root series (it is possible since root series starts from linear term). Having this reverse function $H(y)$ we can directly compute abscissas of intersection points – they are mere values of this function on integer multiples of grid step, since the distance between the graph and the tangent in these points is exact multiple of the distance between horizontal lines of FP grid.

This method helps to compute bad cases near integer tangents rather quickly, but with linearly growing distance from the contact point the "badness" (number of additional bits of function value to calculate for correct precision) of points decreases linearly while the number of points themselves increases exponentially. For example, for sine function near 0 and double precision numbers on exponents from -26 to -12 about 10^9 bad points can be found, but the number of points with "badness" greater than 40 is about 70000. The empiric rule for sine is that the number of intersections on each next exponent is multiplied by 8, but the number of points with certain "badness" remains almost the same.

The presented approaches gives a way to automate test point calculation. The only drawback of existing methods is that they require to write many different programs for one function under test to compute bad cases in different intervals.

5 Applications

The test construction method presented above along with test data generation methods have been applied to make test suites for **sqrt, exp, sin,** and **atan** functions in single and double precision. The tests have been executed on various platforms including Windows XP (MS Visual Studio 2005 libraries) and Linux (**glibc** library of different versions).

Surprisingly big number of various errors has been found, most of them are related with incorrect results on the boundaries of intervals where values of the function are representable as FP numbers or with big angles for sine. FP exception flags raising errors are also often uncovered in the situations of the first kind. Bad cases usually reveal small calculations errors and argument reduction errors for trigonometric functions.

Square root implementations are more mature and have only one-bit calculation errors. Arctangent implementations are different – the one of Mircosoft Visual Studio runtime libraries has only one-bit errors (for rounding to nearest only every fourth bad case reveal such an error, while for other rounding modes – every second one), while glibc implementations of arctangent have more serious errors especially for rounding up, down and to zero.

Versions of **glibc** library are partitioned into two groups – in the first group calculation of functions like square root and exponential have mostly one-bit errors in all rounding modes, but sine became highly erroneous on large arguments; in the second group calculations are precise for rounding to nearest, but can have big errors in other rounding modes, breaking even the basic properties like $\exp x >= 0$ and $-1 \leq \sin x \leq 1$. Examples of the first group are **glibc** 2.1.3, 2.3.2, 2.7 in RedHat Fedore Core distributions – sine implementation in them demonstrates big argument reduction errors, small bad cases reveal only 1-bit calculation errors in about 5% cases in double precision. Examples of the second group are **glibc** 2.3.4 in RHEL 4.0 or **glibc** 2.3.5 in SUSE 10.0 – sine implementation is almost absolutely correct in rounding to nearest mode (only flag setting errors detected in it), but is highly erroneous in other rounding modes (sometimes sine results exceeds 10^{15}!) Even single precision sine sometimes is greater than 1. Bad cases reveal small calculation errors in double precision in about 15% cases. Implementation of sine in Visual Studio.NET 2005 shows argument reduction errors and confusingly incorrect calculation of sine for negative numbers – almost all results for negative arguments are incorrect. Bad cases reveal small calculation errors in about 38% cases.

Systematic exposition of testing results for 17 different implementations of exponential function can be found in [2]. This work revealed about 10 different error kinds, some of them being specific for a single platform, but 3-4 common for most platforms tested. Usually exponential is implemented without significant errors, but bad cases still found small errors in about 12% cases. On **glibc** implementations of the second group exponential function can be highly erroneous for rounding to zero or to infinities. Several points found where its results are negative or much greater than 1 for negative arguments.

The main result is that tests based on structure of FP numbers and intervals of uniform behavior of the function under test are very good for finding various errors, while bad cases for correct rounding help to assess calculation errors in whole and general distribution of inaccuracies.

6 Conclusion

The approach presented in the paper helps to construct systematic test suites for floating-point based implementations of various mathematical functions in one real variable. Error-revealing power of such test suites is rather high – many errors were found in mature and widely used libraries. Although test suites obtained check correct rounding requirement, they also give important information about implementations that do not obey this restriction. For example, the implementations demonstrating only one-bit errors (difference with the correct result only in the last bit of mantissa) can surely be classified as more mature and correct than others.

The search for bad cases requires a lot of machine time, and now the author has explored only neighborhood of zero for all elementary functions in one variable mentioned in POSIX, except for trigonometric and hyperbolic arccosine, and neighborhood of infinity for some of these functions (exponential, arctangent, hyperbolic tangent). Some test data were taken from tests of **crlibm** library [18], which comprise a part of worst cases found by methods described in [19,20,27,28]. Only last year a method has been proposed to compute bad cases for trigonometric functions on large arguments [29]. So, to obtain the full data on bad cases a lot of work is still required.

The experiments conducted demonstrated that most of the test data can be excluded from a test suite without any loss of its error detection power. In particular, for exponential function the test suite constructed using the method described and consisting of about 3.7 million test cases, and the reduced test suite of about 10000 test cases detect all the same errors. The reduction was made by simple removing the test points that do not reveal specific errors or do not add something new to inaccuracy distribution on available 17 implementations of exponential. Now the author tries to formulate a set of rules that can help to reduce test suites without losses in errors revealed.

The interesting problem is to extend the method proposed for functions in two or more variables (and so, in complex variables). One idea is rather straightforward – it is necessary to use not intervals, but areas of uniform behavior of the function under test. But extension of rules concerning FP numbers of special structure and bad cases seem to be much more peculiar, since their straightforward generalizations gives zillions of test cases without any hope to get all the data in a reasonable time. So, some reduction rules should be introduced here from the very beginning to obtain observable test suites.

The tests developed with the presented approach can also facilitate and simplify construction of correct mathematical libraries giving more adequate and precise means for evaluation of their correctness.

References

1. IEEE 754-1985. IEEE Standard for Binary Floating-Point Arithmetic. IEEE, NY (1985)
2. Kuliamin, V.: Standardization and Testing of Implementations of Mathematical Functions in Floating Point Numbers. Programming and Computer Software 33(3), 154–173 (2007)
3. `http://linuxtesting.org`
4. `http://linux-foundation.org`
5. IEC 60559. Binary Floating-Point Arithmetic for Microprocessor Systems. Geneve, ISO (1989)
6. IEEE 854-1987. IEEE Standard for Radix-Independent Floating-Point Arithmetic. IEEE, NY (1987)
7. ISO/IEC 9899. Programming Languages - C. Geneve: ISO (1999)
8. IEEE 1003.1-2004. Information Technology - Portable Operating System Interface (POSIX). IEEE, NY (2004)
9. ISO/IEC 10967-1. Information Technology - Language Independent Arithmetic - Part 1: Integer and Floating Point Arithmetic. Geneve, ISO (1994)
10. ISO/IEC 10967-2. Information Technology - Language Independent Arithmetic - Part 2: Elementary Numerical Functions. Geneve, ISO (2002)
11. ISO/IEC 10967-3. Information Technology - Language Independent Arithmetic - Part 3: Complex Integer and Floating Arithmetic and Complex Elementary Numerical Functions. Draft. Geneve, ISO (2002)
12. Goldberg, D.: What Every Computer Scientist Should Know about Floating-Point Arithmetic. ACM Computing Surveys 23(1), 5–48 (1991)
13. Defour, D., Hanrot, G., Lefevre, V., Muller, J.-M., Revol, N., Zimmermann, P.: Proposal for a standardization of mathematical function implementation in floating-point arithmetic. Numerical Algorithms 37(1-4), 367–375 (2004)
14. Ng, K.C.: Arguments Reduction for Huge Arguments: Good to the Last Bit (1992), `http://www.validlab.com/arg.pdf`
15. Kahan, W.: Minimizing q*m − n, Unpublished (1983), `http://cs.berkeley.edu/~wkahan/testpi/nearpi.c`
16. `http://www.inria.fr/recherche/equipes/arenaire.en.html`
17. de Dinechin, F., Ershov, A., Gast, N.: Towards the post-ultimate libm. In: Proc. of 17th Symposium on Computer Arithmetic, IEEE Computer Society Press, Los Alamitos (2005)

18. `http://lipforge.ens-lyon.fr/www/crlibm/`
19. Lefèvre, V., Muller, J.-M., Tisserand, A.: The Table Maker's Dilemma. INRIA Research Report 98-12 (1998)
20. Lefèvre, V., Muller, J.-M.: Worst Cases for Correct Rounding of the Elementary Functions in Double Precision. In: Proc. of 15th IEEE Symposium on Computer Arithmetic, Vail, Colorado, USA, June (2001)
21. `http://sourceware.org/cgi-bin/cvsweb.cgi/libc/sysdeps/i386/fpu/e_expl.c?cvsroot=glibc`
22. Edelman, A.: The Mathematics of the Pentium Division Bug. SIAM Review 39(1), 54–67 (1997)
23. Ziv, A., Aharoni, M., Asaf, S.: Solving Range Constraints for Binary Floating-Point Instructions. In: Proc. of 16-th IEEE Symposium on Computer Arithmetic (ARITH-16 2003), pp. 158–163 (2003)
24. Parks, M.: Number-Theoretic Test Generation for Directed Rounding. IEEE Trans. on Computers 49(7), 651–658 (2000)
25. `http://www.mpfr.org`
26. Kahan, W.: A Test for Correctly Rounded SQRT. Computer Science Dept, Berkeley (1994), `http://www.cs.berkeley.edu/~wkahan/SQRTest.ps`
27. Lefèvre, V.: An algorithm that computes a lower bound on the distance between a segment and $\mathbb{Z}^2$. In: Developments in Reliable Computing, pp. 203–212. Kluwer, Dordrecht, Netherlands (1999)
28. Stehlé, D., Lefèvre, V., Zimmermann, P.: Worst cases and lattice reduction. In: Proc. of the 16th Symposium on Computer Arithmetic (ARITH'16), pp. 142–147. IEEE Computer Society Press, Los Alamitos (2003)
29. Hanrot, G., Lefèvre, V., Stehlé, D., Zimmermann, P.: Worst Cases of a Periodic Function for Large Arguments. INRIA Research Report 6106 (2007)

Model-Based Testing Service on the Web

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

[1] Tampere University of Technology
Department of Software Systems
P.O. Box 553
FI-33101 Tampere, Finland
{antti.m.jaaskelainen,firstname.lastname}@tut.fi
[2] Nokia Devices
P.O. Box 68
FI-33721 Tampere, Finland

Abstract. Model-based testing (MBT) seems to be technically superior to conventional test automation. However, MBT features some difficulties that can hamper its deployment in industrial contexts. We are developing a domain-specific MBT solution for graphical user interface (GUI) testing of Symbian S60 smartphone applications. We believe that such a tailor-made solution can be easier to deploy than ones that are more generic. In this paper, we present a service concept and an associated web interface that hide the inherent complexity of the test generation algorithms and large test models. The interface enables an easy-to-use MBT service based on the well-known keyword concept. With this solution, a better separation of concerns can be obtained between the test modeling tasks that often require special expertise, and test execution that can be performed by testers. We believe that this can significantly speed up the industrial transfer of model-based testing technologies, at least in this context.

1 Introduction

A widespread problem in software development organizations is how to cut down on the money, time, and effort spent on testing without compromising the quality. A frequent solution is to automate the execution of predefined test cases using test automation tools. Unfortunately, especially in graphical user interface (GUI) testing, test automation often does not find the bugs that it should and the tools provide a return on the investment only in regression type of testing. One of the main reasons for this is that the predefined test cases are linear and static in nature – they do not include the necessary variation to cover defected areas of the code, and they (almost) never change. Moreover, since GUI is often very volatile, it takes time to update the test suites to test the new version of the system under test (SUT). Hence, costly but flexible manual testing is still often chosen as the primary method to ensure the quality, at least in the context of mass consumer products, where GUIs are extremely important.

Model-based testing (MBT) practices [1] that generate tests automatically can introduce more variance to the tests, or even generate an infinite number of different tests. Moreover, maintenance of the testware should become easier when only the models have to be maintained and new updated tests can be generated automatically. Furthermore, developing the test models may reveal more bugs than the actual test execution based on those models. Since model development can be started long before the SUT is mature enough for automatic test execution, detection of bugs early in the product lifecycle is supported.

Concerning industrial deployment of MBT, it has been reported, for instance, that several Microsoft product groups use an MBT tool (called Spec Explorer) on a daily basis [2]. However, it seems that large-scale industrial adoption of the methodology is yet to be seen. If MBT is technologically superior, why has it not overcome conventional ways of automating tests? Based on some earlier studies [3,4] as well as our initial experience, it seems that there are some non-technological obstacles to large-scale deployment. These include the lack of easy-to-use tools and necessary skills. Moreover, since the roles of the testing personnel are affected by this paradigm change, the test organization needs to be adapted as well [5].

In this paper, we tackle the first of these issues, i.e. matching the skills of the testers with easy-to-use tools. We think that one problem with the first generation MBT tools was that they were too general in trying to address too many testing contexts at the same time. We believe that the possibilities of success in MBT deployment will improve with a more *domain-specific solution* that is adapted to a specific context. In our case, the context is the GUI testing of Symbian smartphone applications. There have been cumulatively over 150 million Symbian smartphones shipped [6]. We concentrate on the devices with the S60 GUI framework [7], which is the most commonly found application platform in the current phone models. In addition to device manufacturers, there are a large number of third party software developers making applications on top of Symbian S60. Compared to a more generic approach, based on UML and profiles, for instance [8], our tools should effect a higher level of usability and automation in this particular context.

The background of our approach has been introduced previously in [5,9,10,11]. In this paper, based on earlier work [12,13], the MBT service interface is presented in detail. Our approach is based on a simple web GUI that can be used for providing a model-based testing service. The interface supports setting up MBT sessions. In a session, the server sends a sequence of *keywords* to the client, which executes them on the SUT. For each received keyword, the client returns to the server a Boolean return value: either the execution of the keyword succeeded or not. This *on-line approach* enables the server to generate tests based on the responses of the client, in a way somewhat similar to the Spec Explorer tool [2].

Our scheme should facilitate industrial deployment by minimizing the tasks of the testers. In addition to the service interface, this paper presents an overview of the associated open source tools. The remainder of the paper is structured as follows: In Section 2, we present the background of this paper, i.e., domain-specific MBT for S60 GUI testing. Sections 3 and 4 describe the modeling formalism and the associated tool set. In Section 5, the service concept is introduced in detail including the interfaces that we have defined. Finally, Section 6 concludes the paper with a final discussion including ideas for future work.

2 Domain-Specific MBT

Research on model-based testing (MBT) has been conducted widely in both industry and academia. From the practical perspective, the fundamental difference between MBT and non-MBT automation is that, in the latter case, the tests are scripted in some programming or scripting language. In the former case, on the other hand, the tests are generated based on a formal model of the SUT. The model describes the system from the perspective of testing at a high level of abstraction. However, the definition of a "model" varies greatly, depending on the approach [1]. In our approach, a model is a parallel composition of Labeled State Transition Systems (LSTSs). This formalism enables us to generate tests that introduce variation in the tested *behavior*, for instance, by executing different actions in many different orders allowed by the SUT. In some other MBT approaches, the goal might be to generate all possible data values for some type of parameters. Thus, there are many different types of MBT solutions that do not necessarily have much in common. The algorithms for generating tests from the models may be significantly different, depending on the formalism and the testing context.

However, a common goal in many MBT schemes is to execute high volumes of different tests. Once the MBT regime has been set up and running, the generation of *new* tests based on the models is as easy as running the same old tests again and again. Obviously, old tests can still be repeated for debugging purposes if necessary.

In spite of these benefits, the industrial adoption of this technology has been slow. Robinson [3] states that the most common problems in deployment are the managerial difficulties, the making of easy-to-use tools, and the reorganization of the work with the tools. Hartman [4] reports problems with the complexity of the provided solution and counter-intuitive modeling. Our early experiences support these findings. Moreover, it must be acknowledged that modeling needs a special kind of expertise that may not be available in a testing organization. However, such expertise might be available as a service, especially when operating in a specialized domain such as testing smartphone applications.

We think that a problem with the first generation MBT tools was that they were too general. These tools tried too much to address many testing contexts at the same time, for instance by generating tests based on UML models that could describe almost any type of SUT. We believe that the chances of success in MBT deployment will improve with more domain-specific solutions that are adapted to specific contexts. In our case, the context is the GUI testing of Symbian S60 [6,7] smartphone applications. Symbian is the most widely spread operating system for smartphones and S60 is a GUI platform built on the top of it. There are a large number of third party software developers making applications on top of Symbian S60. One driving force in any automation solution for this product family setting is the ability to reuse as many tests as possible when a new product of the family is created. Thus, we have built our test model library to support the reuse of test models.

In addition, in terms of industrial adoption, MBT needs to be adapted to the existing testing processes that are shifting towards more agile practices [14] from the traditional ones based on the V-model [15] and its variations. In agile contexts, on the one hand, developers are already relying on test automation to support refactoring and generally understand its benefits as compared to manual testing. On the other hand, it seems especially important to provide easy-to-use tools and services that do not place an additional burden, such as that of test modeling, on the project personnel. We have identified a minimum of three modes [11] to be supported in agile processes: *smoke testing* should be performed in each continuous integration cycle; user stories can be tested in a *use-case testing* mode; and there should be a *bug hunting* mode, whose only purpose is to support finding defects efficiently in long test runs.

Concerning domain-specific issues, the Symbian S60 domain entails the following problems, among others, from the testing point of view:

– How to make sure the application under test works with pre-installed applications such as calendar, email, and camera?
– How to test the interactions between the different applications running on the phone? How to make sure that the phone does not crash if a user installs a third-party application? What happens if, for instance, some application attempts to delete an MP3 file that is being played by another application?
– How to test that your software works with different keyboards and screen resolutions?

The domain concepts of Symbian S60 testing can be described using *keywords* and *action words* [16,17]. Action words describe the tasks of the user, such as opening the camera application, dialing a specified number, or inserting the number of the recipient to a message. Keywords, on the other hand, correspond to physical interaction with the device such as the key presses and observations. Each action word needs to be implemented by at least one sequence of keywords. For example, starting a camera application can be performed using a short-cut key or a menu, for instance, and verifying that a given string is found from the screen. The verification enables checking that the state of the model and state of the SUT match each other during the test run.

Keywords and similar concepts are commonly used in GUI testing tools. We believe that using these concepts in conjunction with MBT can help to deploy the approach in industrial settings. Since testers are already familiar with the keyword concept we just need to hide the inherent complexity of the solution and provide as simple a user interface as possible. The existing test execution tools that already implement keywords should be adaptable to receive a sequence of keywords from a server. The role of the server is to encapsulate the test model library and the associated test generation heuristics. Based on a single keyword execution on the SUT, the client tool returns to the server a Boolean value based on success or failure of the execution. The server then selects the next keyword to be sent to the client based on this return value.

3 Modeling Formalism

In this section, the fundamentals of our modeling formalism are presented for the interested reader. As already mentioned, we use Labeled State Transition Systems (LSTSs) as our modeling formalism. This is an extension of the Labeled Transition System (LTS) format with labels added to states as well as to transitions. The formal definition is presented below. It should be noted that while each transition is associated with exactly one action, any number of attributes may be in effect in a state.

Definition 1 (LSTS). *A* labeled state transition system*, abbreviated LSTS, is defined as a sextuple* $(S, \Sigma, \Delta, \hat{s}, \Pi, val)$ *where* S *is the set of* states*,* Σ *is the set of* actions *(transition labels),* $\Delta \subseteq S \times \Sigma \times S$ *is the set of* transitions*,* $\hat{s} \in S$ *is the* initial state*,* Π *is the set of* attributes *(state labels) and* $val : S \longrightarrow 2^{\Pi}$ *is the* attribute evaluation function*, whose value* $val(s)$ *is the set of attributes in effect in state* s*.*

In our approach, the models are divided into four categories according to their uses: *action machines*, *refinement machines*, *launch machines* and *initialization machines*. Action machines are used to model the SUTs on the action word level. Thus, they are the main focus of the modeling work. Keyword implementations for action words are defined in refinement machines. Together, these machines form most of the model architecture; the remaining two types are focused on supportive tasks. Launch machines define keyword sequences required to start up an action machine, such as switching to a specific application. Initialization machines, on the other hand, define sequences for setting the SUT into the initial state assumed by action machines and are executed before the actual test run. They can also be used to return the SUT back to a known state after the test. Both of these functions have simple default actions. Hence, explicitly defined launch and initialization machines are rarely needed.

Concerning the keywords, many of them require one or more parameters to define their function. Sometimes these are fixed to the GUI, such as a parameter that defines which key to press, but sometimes they represent real-world data: a date or a phone number, for example. Embedding such information directly into the models is problematic, because they would be limited to a fixed set of data values and possibly tied to a specific test configuration. Another problem with the use of data is that storing it in state machines requires duplicate states for each possible value of data, which quickly results in a state space explosion [18]. To solve these problems, we have developed two methods of varying the data in models: localization data and data statements.

The basic function of *localization data* is to hold the text strings of the GUI in different languages, so that the models need not be tied to any specific language variant of the SUT. The data is incorporated into the model by placing a special identifier in a keyword. When the keyword is executed, the identifier is replaced with the corresponding element from the localization tables. More complicated use of data can be accomplished by placing *data statements* (Python [19] code) in actions. These statements may be used in any actions, not just keywords. Data provided by external *data tables* can be used in these data statements.

In order to be used in a test run, the models must be combined in *parallel composition*. The models involved in this process are action machines, refinement machines, launch machines (both explicitly defined and automatically generated), and a special model called the *task switcher*. The latter is generated to manage some of the synchronizations between the models. In the composition, the models are examined and rules generated for them according to the domain-specific semantics to determine what actions can be executed in a given state. As usual, the composition can be used to create one large test model that combines all the various components, or it can be performed on the fly during the test run. We have found the latter method to be preferable, since combining a large number of models can easily result in a serious state explosion problem. The definition of the parallel composition, extended from [20] for LSTSs, is the following:

Definition 2 (Parallel composition $\|_R$). $\|_R$ $(L_1,\ldots,L_n)$ *is the* parallel composition *of LSTSs* $L_1,\ldots,L_n$, $L_i = (S_i,\Sigma_i,\Delta_i,\hat{s}_i,\Pi_i,val_i)$*, according to* rules R; $\forall i,j; 1 \leq i < j \leq n : \Pi_i \cap \Pi_j = \emptyset$. *Let* Σ_R *be a set of resulting actions and* $\surd$ *a "pass" symbol such that* $\forall i; 1 \leq i \leq n : \surd \notin \Sigma_i$. *The rule set* $R \subseteq (\Sigma_1 \cup \{\surd\}) \times \cdots \times (\Sigma_n \cup \{\surd\}) \times \Sigma_R$. *Now* $\|_R$ $(L_1,\ldots,L_n) = (S,\Sigma,\Delta,\hat{s},\Pi,val)$*, 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 $(1 \leq i \leq n)$ *either*
  - $(s_i,a_i,s_i') \in \Delta_i$ *or*
  - $a_i = \surd$ *and* $s_i = s_i'$
- $\hat{s} = (\hat{s}_1,\ldots,\hat{s}_n)$
- $\Pi = \Pi_1 \cup \cdots \cup \Pi_n$
- $val((s_1,\ldots,s_n)) = \{\pi \in \Pi \mid \exists i; 1 \leq i \leq n : \pi \in val_i(s_i)\}$.

The composition is based on a rule set which explicitly defines the synchronizations between the actions. An action of the composed LSTS can be executed only if the corresponding actions can be executed in each component LSTS, or if the component LSTS is indifferent to the execution of the action. In some, extreme cases an action may require the cooperation of all the component LSTSs, or a single component LSTS may execute an action alone. In practice, however, most actions in our models are executed singly or synchronized between two components, though larger synchronizations also exist.

An important concept in the models is the division of states into *running* and *sleeping states*. In more detail, running states contain the actual functionality of the models, whereas sleeping states are used to synchronize the models with each other. The domain-specific semantics ensure that exactly one model is in a running state at any time, as is the case with Symbian applications. As testing begins, the running model is always the task switcher. Running and sleeping states are defined implicitly according to the transitions in the models.

4 Overview of the Tools

In this section, we provide an overview of the toolset supporting our approach. The toolset is currently under construction. The tool architecture is illustrated in Figure 1. The toolset can be divided into four parts plus a database. The first is the model design part, which is used for creating the component models and data tables. The second is the test control part, where tests are launched and observed. The third is the test generation part that is responsible for assembling the tests and controlling their execution. The fourth is the keyword execution part, whose task is to communicate with the SUT through its GUI.

Concerning the model design part of the toolset, the tools are used to create the test models and prepare them for execution. There are two primary design tools: Model Designer [13] and Recorder [21]. The latter is an event capturing tool designed to create

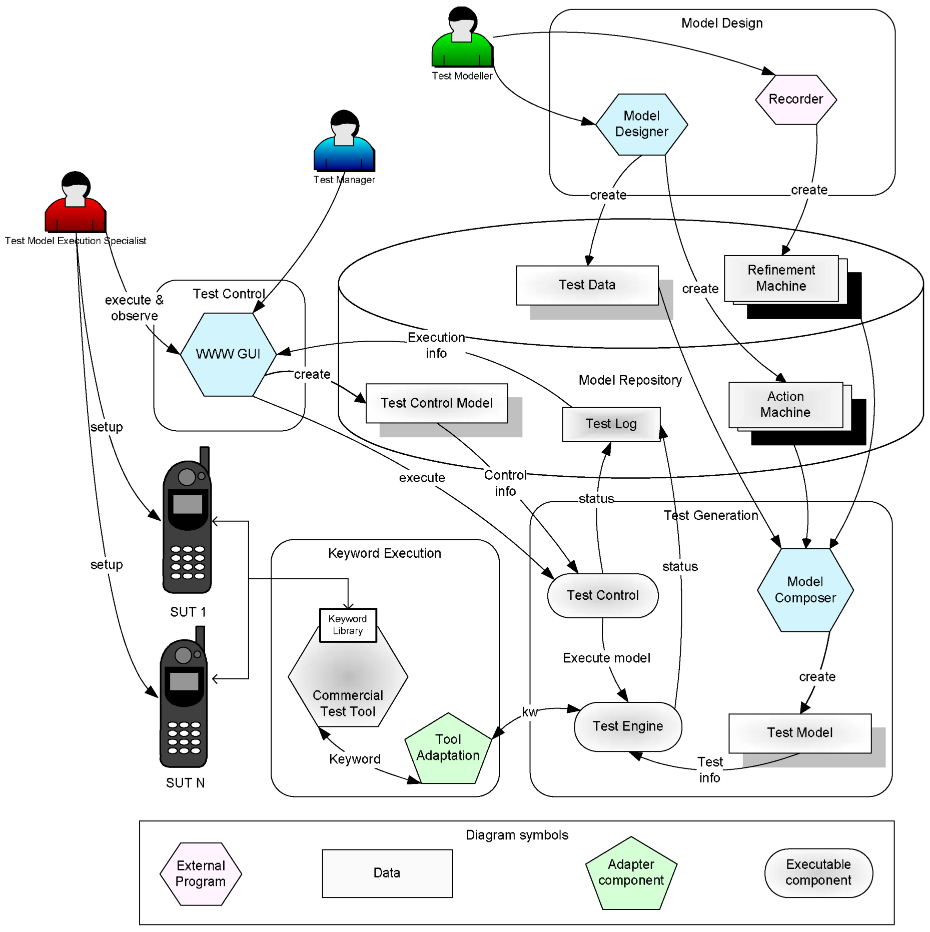


Fig. 1. Test tool architecture

keyword sequences out of GUI actions; these sequences can then be formed into refinement machines. Model Designer, on the other hand, is the main tool for creating action machines and data tables. It is also responsible for assembling the models into a working set ready for testing; even refinement machines created with Recorder pass through Model Designer. The elements of this working set are placed into the model repository.

After the models with their associated information have been prepared with the design tools, the focus moves to the test control part. This part contains a web GUI which is used to launch the test sessions. Once a test session has been set up, the Test Control tool in the test generation part of the toolset takes over. First, it checks the *coverage requirement* (a formal test objective) that it received and determines what model components are required for the test run. These are given to Model Composer, which combines them into a single model on the fly. The model is managed by Test Engine, which determines what to do next, based on the parameters it receives from Test Control. Both

Test Control and Test Engine report the progress of the test run into a test log, which may be used for observing, debugging, or repeating the test.

As keywords are executed in the model, Test Engine relays them to the keyword execution part. The purpose of this part is to handle their execution in the SUT. The SUT responds with the success status (true or false) of the keyword, which is then relayed back to Test Engine. The first link in the communication between Test Engine and the SUT is handled by a specific adapter tool, which translates the keywords into a form understood by the receiver and manages the gradual execution of some more complex keywords. The next part in the chain is the test tool which directly interacts with the SUT. The nature of this tool depends on the SUTs in question and is not provided alongside the toolset. The users of the toolset must provide their own test tool and use the simple interface offered by the adapter. In our case, we have used commercial components, namely Mercury Functional Testing for Wireless (MFTW) and Mercury QuickTest Professional (QTP) [22].

We have designed the architecture to support the plugging-in of different test generation heuristics. Currently, we have implemented three heuristics which allow us to experiment with the tools: a purely random heuristics that can be used in bug hunting mode, and two heuristics based on game-theory [11] to be used in the use case testing mode: a single thread and a two thread version. The difference between the two is that the latter continues to search an optimal path to a state fulfilling the coverage requirement, while the other thread waits for a return value from the client executing a keyword.

It is anticipated that in deploying our approach the testing personnel should consist of the following roles (see Figure 1): test manager, test modeler, and test model execution specialist. The test manager defines the entry and exit criteria for the test model execution, and defines which metrics are gathered. The test manager should also focus on communicating the testing technology aspects. This includes explaining how model-based testing compares to conventional testing methods and advocating reasons for and against using it for management and testing personnel. In these respects, model-based testing is similar to any new process initiation.

The main goal of the test modeler is to update and maintain the test model library using the Model Designer and Recorder tools based on product specifications if such exist. The test modeler can also be responsible for designing the execution of the model and setting up the environment accordingly.

The test model execution specialist orders the test sessions from the web GUI according to the chosen test strategy. He/she also observes the test execution to ensure that the models are used according the agreed principles and test data. Another focus of this role is in reporting the results and faults onward. The purpose is to document the test model usage and testware in a way that enables its reuse.

5 Providing a Symbian S60 Test Service

In this section, the service scheme is presented in detail. The following subsections describe the interfaces provided by our server.

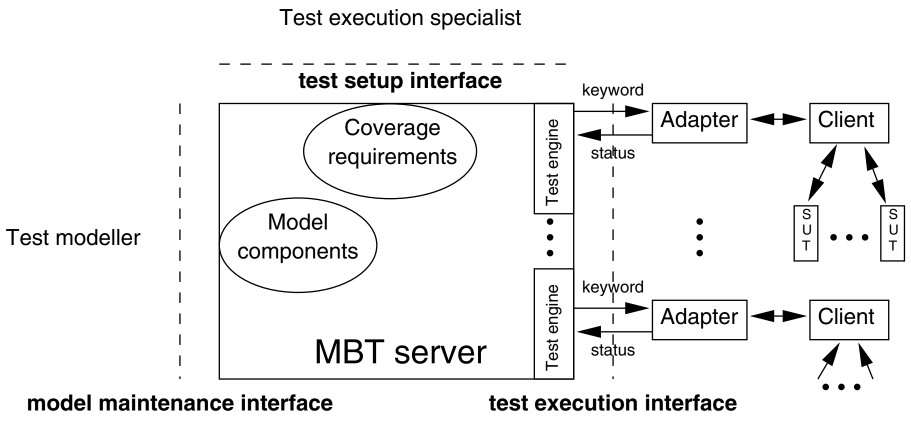


Fig. 2. MBT testing server, adapters, clients, and SUTs

5.1 Server and Clients

The architecture of the toolset described earlier enables a client-server scheme where the keyword execution and test generation parts are separated. To facilitate the deployment of model-based GUI testing in the context of Symbian S60 applications, we have set up a prototype version of the server that implements the test generation part. It provides testers an easy interface to the MBT tools.

The server is accessed through three interfaces. First, there is an interface through which test modelers update the test model components on the server. Second, there is a web interface through which test execution specialists can set up tests. Finally, there is an interface for sending keywords to adapters which execute the corresponding events on actual devices. Figure 2 illustrates the scheme.

Although the MBT server could be installed as a local application in the client machine, there are some practical reasons for dedicating a separate PC for that purpose. The most important reason is that some of our test generation algorithms, i.e. the ones based on game heuristics, can produce better results given more processor time and memory. Fortunately, computing power is very cheap nowadays but it still pays off to have a dedicated machine. Moreover, the server provides a shared platform for test modelers to update the model library and test execution specialists to set up tests. Furthermore, all the users of the server do not need to know the details of the SUT, for instance the physical form or other design issues that may be confidential at the time of testing. For the purposes of test modeling, it should be enough to know what previously tested member of the product family this new member resembles the most and what the differences are concerning the modeled behavior.

5.2 Test Setup Interface

There are a number of parameters that need to be given in order to start a test run. The most important ones are:

1. SUT types: which phone models will be used in the test run? This affects the automatic selection of test model components.
2. Test model: which applications will be used in the test run? Based on this choice, the test model components are selected and composed together to form a single test model that will be used in the test run.
3. Test mode: the test can be executed in smoke test, bug hunt, and use-case testing mode. In each mode, a coverage criterion should also be given. The criterion defines when the test run can be stopped, but it can also be used to guide the test generation as in the case of use-case testing mode.
4. Number of clients: how many clients can be used to execute the test? Using more than one client can often improve the time in which the test is finished. For example, a complicated coverage criterion can often be divided into smaller criteria that can be fulfilled in concurrent test executions.
5. The test generation algorithm, connection parameters, and logging system.

To support different types of testing in the various phases of the testing process, the server supports the three testing modes mentioned above. In the smoke testing mode the server generates tests in a breadth-first search fashion until the coverage criterion has been fulfilled; for instance, 30 minutes have passed or 1000 keywords have been executed. In the use case mode, the tester inputs a use case (in the form of a sequence of action words) to the server, which then generates tests to cover that use case using the game heuristics. As already discussed, the main motivation for this mode is compatibility with the existing testing processes: the tests are usually based on requirements and the test results can be reported based on the coverage of the requirements. In the bug-hunting mode, in addition to purely random generation, the server could generate a much longer sequence of keywords that tries to interleave the behavior of the different applications as much as possible in order to detect hard-to-find bugs related to mutual exclusion, memory leaks, etc.

When the test setup is ready, the corresponding test model is automatically built from components of the model library. After that, the given coverage criterion could be split so that there is a chunk for every client to cover. Finally, one *test engine* process per every client could be launched to listen to a TCP/IP connection. A test engine will serve a client until its part of the coverage criterion has been covered or it is interrupted. Now the MBT server is ready for the real test run, during which the clients and the server communicate through the test execution interface.

5.3 Test Execution Interface

To start a test run, the test execution specialist starts the devices to be used as targets in the tests as well as the clients and adapters. The adapters are configured so that they connect to the test engines waiting on the server. Test execution on the client starts immediately when its adapter has been connected to the test engine.

During the execution, a test engine repeats a loop where it first sends a keyword to an adapter. The adapter, with the help of the test execution tool it is controlling, converts the keyword into an input event or an observation on the SUT. As already discussed, there are different keywords for pushing a button on the phone keypad and verifying that a given string is found on the screen, for instance. After that, the adapter returns the status of the keyword execution, i.e. a Boolean value denoting success or failure, to the test engine. In a normal case, when the status of the keyword execution is allowed by the test model, the server loops and sends a new keyword to the adapter.

Otherwise, unexpected behavior of the SUT is detected, maybe due to a bug in the SUT, and the server starts a shutdown or recovering sequence. It informs the adapter that it has found an anomaly. The adapter may then save screenshots, a memory dump or other information useful for debugging. It also sends an acknowledgement of having finished operations to the server. Finally, the test engine may either close the connection, or try to recover from the error by sending some keywords again, for instance to reboot the SUT.

Regardless of the mode, during a test session a log of executed keywords is recorded for debugging purposes. When a failure is noticed, the log can be used for repeating the same sequence of keywords in order to reproduce the failure.

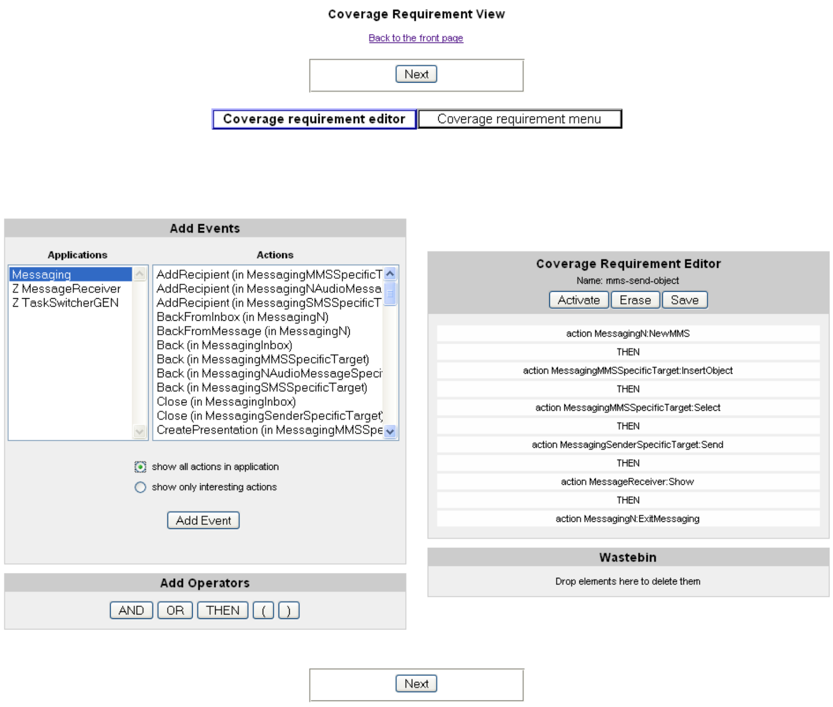
GUI testing can sometimes be slow, even with the most sophisticated tools. In order to cope with this, we should extend our solution to support the concurrent testing of several target phones using one server. Testing a new Symbian S60 application could be done so that one client is used for testing the application in isolation from other applications, while other clients are testing some application interactions.

5.4 Using the Web GUI

The testers interact with the server using a web interface. The interface has been implemented in AJAX [23] and it consists of several different views. In the following, we will introduce the basic usage of the interface step by step.

When the tester wants to start a test session, he or she first logs into the system. After that, the system offers two alternatives: either to start a session by repeating a log from some previous session or simply from scratch. In the latter case, a model configuration must next be selected. Such a configuration can consist of models of certain applications whose interactions should be tested, for instance. Next, a view called the coverage requirement editor is opened (see Figure 3). In this view, the tester can construct a new coverage requirement from actions of the model components included in this configuration. Since the number of different actions can be large, there is a possibility to limit the shown actions to those marked "interesting" by the test modelers. The coverage requirement is composed of actions and operators *THEN*, *AND*, and *OR*, as well as parentheses. As an example, consider a requirement for sending a multimedia message (MMS) from one SUT to another with an attachment:

```
action Messaging1-Main:NewMMS THEN
action Messaging1-MMS:InsertObject THEN
action Messaging1-MMS:Select THEN
action Messaging1-Sender:Send THEN
action Messaging2-Receiver:Show THEN (
  action Messaging1-Main:ExitMessaging AND
  action Messaging2-Main:ExitMessaging
)
```


Coverage Requirement View
Back to the front page
Next
Coverage requirement editor
Coverage requirement menu
Add Events
Applications
Actions
Messaging
Z MessageReceiver
Z TaskSwitcherGEN
AddRecipient (in MessagingMMSSpecificT
AddRecipient (in MessagingNAudioMessa
AddRecipient (in MessagingSMSSpecificT
BackFromInbox (in MessagingN)
BackFromMessage (in MessagingN)
Back (in MessagingInbox)
Back (in MessagingMMSSpecificTarget)
Back (in MessagingNAudioMessageSpeci
Back (in MessagingSMSSpecificTarget)
Close (in MessagingInbox)
Close (in MessagingSenderSpecificTarget)
CreatePresentation (in MessagingMMSSpe
show all actions in application
show only interesting actions
Add Event
Add Operators
AND
OR
THEN
(
)
Coverage Requirement Editor
Activate
Erase
Save
Wastebin
Drop elements here to delete them
Next

Name: mms-send-object

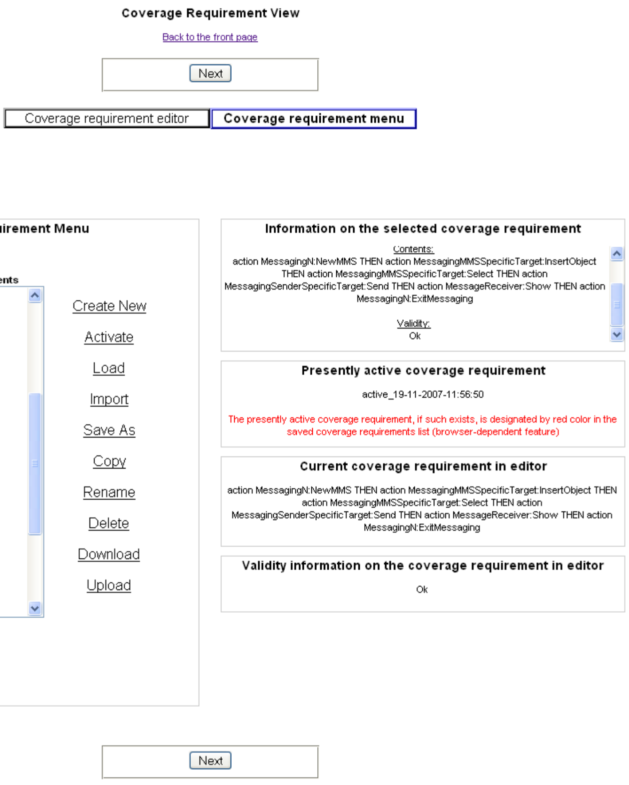
action MessagingN:NewMMS
THEN
action MessagingMMSSpecificTarget:InsertObject
THEN
action MessagingMMSSpecificTarget:Select
THEN
action MessagingSenderSpecificTarget:Send
THEN
action MessageReceiver:Show
THEN
action MessagingN:ExitMessaging

Fig. 3. Coverage requirement editor

In the example, Messaging1 is the SUT that should send the MMS and Messaging2 the one that should receive it. Once the message has been composed, sent, received and opened, both SUTs should return to the main menu in a non-specified order. The right hand side of Figure 3 shows the corresponding coverage requirement in the case of one SUT. In the one phone configuration, the sender and the receiver are the same device, while in the two phone configuration they are different. Replacing operator AND with OR would simply mean that either one of the phones should return to the main menu. If the requirement under construction if not well-formed, the requirement turns red and an error message is displayed. The coverage language is presented in more detail in [11].

Since constructing long coverage requirements can take some effort and time, there is a view where they can be saved and loaded (see Figure 4). Moreover, there is an option to upload and download coverage requirements if the tester wants to use another editor.

In the next view, the tester can set the parameters for the test session. First of all, there are different heuristics corresponding to the different testing modes. Moreover, there are some other parameters to be selected based on the heuristics used. For instance, using the game heuristics in the requirement coverage mode requires the depth of the search tree. There are naturally default values available, but based on the model complexity, better results, i.e. reaching the coverage requirement faster, can be achieved by carefully


Coverage Requirement View
Back to the front page
Next
Coverage requirement editor
Coverage requirement menu
Coverage Requirement Menu
Saved coverage requirements
Create New
Activate
Load
Import
Save As
Copy
Rename
Delete
Download
Upload
Next

active_19-09-2007-09:07:06
active_19-09-2007-09:21:35
active_19-09-2007-09:37:30
active_19-09-2007-11:26:49
active_19-09-2007-15:26:26
active_19-09-2007-15:59:54
active_19-11-2007-11:15:28
active_19-11-2007-11:56:50
active_22-10-2007-10:22:06
active_22-10-2007-11:04:32
active_22-10-2007-14:23:56
active_23-10-2007-11:33:25
active_25-09-2007-11:01:52
active_25-09-2007-11:03:50
active_25-09-2007-11:08:18
active_25-09-2007-11:19:19
active_25-09-2007-11:44:51
active_25-09-2007-15:35:12
active_25-09-2007-15:40:23
active_25-09-2007-15:53:51
active_28-08-2007-14:04:04
active_31-10-2007-12:12:48
active_31-10-2007-15:23:15

Information on the selected coverage requirement

Contents:
action MessagingN:NewMMS THEN action MessagingMMSSpecificTarget:InsertObject THEN action MessagingMMSSpecificTarget:Select THEN action MessagingSenderSpecificTarget:Send THEN action MessageReceiver:Show THEN action MessagingN:ExitMessaging

Validity:
Ok

Presently active coverage requirement

active_19-11-2007-11:56:50

The presently active coverage requirement, if such exists, is designated by red color in the saved coverage requirements list (browser-dependent feature)

Current coverage requirement in editor

action MessagingN:NewMMS THEN action MessagingMMSSpecificTarget:InsertObject THEN action MessagingMMSSpecificTarget:Select THEN action MessagingSenderSpecificTarget:Send THEN action MessageReceiver:Show THEN action MessagingN:ExitMessaging

Validity information on the coverage requirement in editor

Ok

Fig. 4. Coverage requirement menu

selecting the parameters. In addition to these, the tester can specify the seed for the random number generator.

Another important selection to be made in this view is the data and localization tables to be used in the test runs. For this purpose, the tester is presented with a list of predefined files in the server.

Finally, the tester can choose to start the test run in the next view. There is also a selection on how detailed a log is displayed during the test run. In any case, the tester can always choose to view all the logged information. The log is automatically saved so that the test run can be repeated for debugging purposes, for instance. When the test execution specialist presses the “Start” button, the server starts waiting for a connection from a client where the SUTs have been connected using Bluetooth or a USB connection. An example test setup with two targets is shown in Figure 5. On the right hand side the test log in the web GUI is shown. The client machine on the left hand side has two targets connected using a Bluetooth connection.

After the test session is finished, the web interface turns either green or red, based on success or failure. In the latter case, the tester may want to download the log for reporting or debugging. In the former case, the tester can report that the requirement in

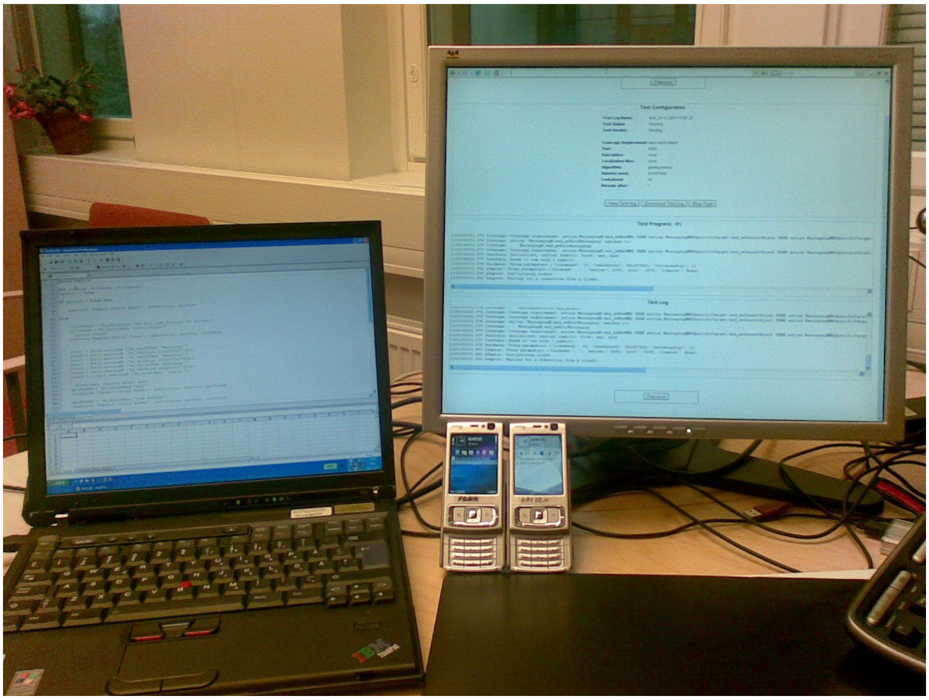


Fig. 5. Test setup with two SUTs

question has now been tested. The interested reader can view a video of the test session described in the above example at `http://www.cs.tut.fi/~teams`

6 Discussion

In this paper we have described a model-based GUI testing service for Symbian S60 smartphone applications. The approach is based on a test server that is currently in the prototype stage. We are implementing the tools we have described and are releasing new versions under the MIT Open Source Licence. A download request can be made through the URL mentioned above.

In our solution, the server encapsulates the domain-specific test models and the associated test generation heuristics. The testers, or test execution specialists, order tests from the server, and the test adapter clients connect to the phone targets under test. The main benefit of this approach compared to more generic approaches is that it should be easier to deploy in industrial environments; in practice, the tasks of the tester are minimized to specifying the coverage requirement as well as some parameters for heuristics, etc. We are developing the web interface to be as usable as possible and plan to conduct usability surveys in the future.

How then could the service model be used? The organization of testing services affects what kind of testing process could be used. This demands a flexible approach for ease of coordination [24]. In industrial practice, it would be important to get reliable service based on the current testing needs. This is in line with the current trends of the software industry [25]. At best, there would be several providers for the service to fulfill the needs of different end-users. Beside technical competence, communication skills are emphasized in order to provide transparency to the details of the solution.

Case studies on using the service concept are on the way. We have already used the web GUI internally for several months. In these experiments, the SUT has been the S60 Messaging application, including features such as short message service (SMS) and multimedia messages (MMS). The former supports sending only textual messages, while the latter supports attaching photos, video and audio clips. So far we have performed testing with configurations of one to two phones. Based on the positive results of this internal use, we are working towards transferring this technology to our industrial partners. One of the partners has already successfully tried out our test server in actual test runs without the web GUI. We anticipate that the web GUI will help us in conducting wider studies in the future.

Acknowledgements

This paper reports the ongoing results of research funded by the Finnish Funding Agency for Technology and Innovation (TEKES), Nokia, Conformiq Software, F-Secure, and Plenware, as well as the Academy of Finland (grant number 121012). For details, see `http://practise.cs.tut.fi/project.php?project=tema`

References

1. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann, San Francisco (2007)
2. Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., Veanes, M.: Testing concurrent object-oriented systems with Spec Explorer. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 542–547. Springer, Heidelberg (2005)
3. Robinson, H.: Obstacles and opportunities for model-based testing in an industrial software environment. In: Proceedings of the 1st European Conference on Model-Driven Software Engineering, Nuremberg, Germany, pp. 118–127 (2003)
4. Hartman, A.: AGEDIS project final report. Cited (March 2008), `http://www.agedis.de/documents/FinalPublicReport28D1.629.PDF`
5. Katara, M., Kervinen, A., Maunumaa, M., Pääkkönen, T., Satama, M.: Towards deploying model-based testing with a domain-specific modeling approach. In: Proceedings of TAIC PART – Testing: Academic & Industrial Conference, Windsor, UK, pp. 81–89. IEEE Computer Society, Los Alamitos (2006)
6. Symbian Cited (March 2008), `http://www.symbian.com/`
7. S60. Cited (March 2008), `http://www.s60.com`
8. OMG: UML testing profile, v 1.0. Cited (March 2008), `http://www.omg.org/technology/documents/formal/test_profile.htm`
9. Kervinen, A., Maunumaa, M., Pääkkönen, T., Katara, M.: Model-based testing through a GUI. In: Grieskamp, W., Weise, C. (eds.) FATES 2005. LNCS, vol. 3997, pp. 16–31. Springer, Heidelberg (2006)

10. Kervinen, A., Maunumaa, M., Katara, M.: Controlling testing using three-tier model architecture. In: Proceedings of the Second Workshop on Model Based Testing (MBT 2006), Vienna, Austria. Electronic Notes in Theoretical Computer Science, vol. 164(4), pp. 53–66. Elsevier, Amsterdam (2006)
11. Katara, M., Kervinen, A.: Making model-based testing more agile: a use case driven approach. In: Bin, E., Ziv, A., Ur, S. (eds.) HVC 2006. LNCS, vol. 4383, pp. 219–234. Springer, Heidelberg (2007)
12. Katara, M., Kervinen, A., Maunumaa, M., Pääkkönen, T., Jääskeläinen, A.: Can I have some model-based GUI tests please? Providing a model-based testing service through a web interface. In: Proceedings of the second annual Conference of the Association for Software Testing (CAST 2007), Bellevue, WA, USA (2007)
13. Jääskeläinen, A.: A domain-specific tool for creation and management of test models. Master's thesis, Tampere University of Technology (2008)
14. Boehm, B., Turner, R.: Balancing Agility and Discipline: A Guide for the Perplexed. Addison-Wesley, Reading (2004)
15. Rook, P.: Controlling software projects. Softw. Eng. J. 1, 7–16 (1986)
16. Buwalda, H.: Action figures. STQE Magazine, 42–47 (March/April 2003)
17. Fewster, M., Graham, D.: Software Test Automation: Effective use of test execution tools. Addison-Wesley, Reading (1999)
18. Valmari, A.: The state explosion problem. In: Lectures on Petri Nets I: Basic Models, London, UK, pp. 429–528. Springer, Heidelberg (1996)
19. Python: Python Programming Language homepage. Cited (March 2008), `http://python.org/`
20. Karsisto, K.: A new parallel composition operator for verification tools. Doctoral dissertation, Tampere University of Technology (number 420 in publications) (2003)
21. Satama, M.: Event capturing tool for model-based GUI test automation. Master's thesis, Tampere University of Technology (2006), Cited March 2008, `http://practise.cs.tut.fi/project.php?project=tema&page=publications`
22. HP: Mercury Functional Testing homepage. Cited (March 2008), `http://www.mercury.com/us/products/quality-center/functional-testing/`
23. Zakas, N.C., McPeak, J., Fawcett, J.: Professional Ajax, 2nd edn. Wiley, Chichester (2007)
24. Taipale, O., Smolander, K.: Improving software testing by observing practice. In: ISESE 2006. Proceedings of the 2006 ACM/IEEE international symposium on empirical software engineering, pp. 262–271. ACM Press, New York (2006)
25. Microsoft: Microsoft unveils vision and road map to simplify SOA, bridge software plus services, and take composite applications mainstream (2007-11-28). Cited March 2008, `http://www.microsoft.com/presspass/press/2007/oct07/10-30OsloPR.mspx`

Using Disparity to Enhance Test Generation for Hybrid Systems

Thao Dang and Tarik Nahhal

VERIMAG, 2 avenue de Vignate
38610 Gières, France
{Thao.Dang,Tarik.Nahhal}@imag.fr

Abstract. This paper deals with the problem of test generation for hybrid systems, which are systems with mixed discrete and continuous dynamics. In our previous work [9], we developed a coverage guided test generation algorithm, inspired by a probabilistic motion planning technique. The algorithm is guided via a process of sampling goal states, which indicate the directions to steer the system towards. In this paper, we pursue this work further and propose a method to enhance coverage quality by introducing a new notion of disparity. This notion is used to predict the situations where the goal states can not be 'directly' reached. We then develop an adaptive sampling method which permits improving coverage quality. This method was implemented and successfully applied to a number of case studies in analog and mixed-signal circuits, a domain where hybrid systems can be used as an appropriate high level model.

1 Introduction

Hybrid systems, that is, systems exhibiting both continuous and discrete dynamics, have proven to be a useful mathematical model for various physical phenomena and engineering systems. Due to the safety critical features of many such applications, much effort has been devoted to the development of automatic analysis methods and tools for hybrid systems, based on formal verification. Although these methods and tools have been successfully applied to a number of interesting case studies, their applicability is still limited to systems of small size due to the complexity of formal verification. It soon became clear that for systems of industrial size, one needs more 'light-weight' methods. Testing is another validation approach, which can be used for much larger systems and is a standard tool in industry, although it can only reveal an error but does not permit proving its absence. Although testing has been well studied in the context of finite state machines (e.g. [13] and references therein) and, more recently, of real-time systems, it has not been much investigated for continuous and hybrid systems. Therefore, a question of great interest is to bridge the gap between the verification and testing approaches, by defining a formal framework for testing of hybrid systems and developing methods and tools that help automate the testing process.

A number of special characteristics of hybrid systems make their testing particularly challenging, in particular the infiniteness of their state space and input space. In general, in order to test an open system, one first needs to feed an input signal to the system and then check whether the behavior induced by this input signal is as expected. When there is an infinite number of possible input signals, it is important to choose the ones that lead to interesting scenarios (with respect to the property/functionality to test). Concerning existing work in test generation for hybrid systems, the paper [12] proposed a framework for generating test cases by simulating hybrid models specified using the language Charon. The probabilistic test generation approach based on the motion planning technique **RRT** [15], proposed in [5,11], is similar to the algorithm **gRRT** we proposed in [9]. Our method **gRRT** differs from these in the use of a coverage measure to guide the test generation. Although for various examples, the guiding tool significantly improves the coverage quality, it still suffers from a problem that we call the *controllability issue*. Roughly speaking, the guiding tool, while trying to increase the coverage, might make the system follow the directions which are unreachable. Indeed, any **RRT**-based search method which does not take into account the system's dynamics might suffer from this problem.

In this paper, we propose a method to tackle this controllability issue by introducing a new notion of disparity describing the difference between two distributions of point sets. The information of the disparity between the goal and the visited states is used to steer the exploration towards the area where the dynamics effectively allows to improve the coverage. A combination of this disparity guided method and the coverage guided algorithm **gRRT** results in a new adaptive algorithm, which we call **agRRT**. Experimental results show the coverage efficiency of the new algorithm. In terms of applications, besides traditional applications of hybrid systems (e.g. control systems), we have explored a new domain which is analog and mixed signal circuits. Indeed, hybrid systems provide a mathematical model appropriate for the modeling and analysis of these circuits. The choice of this application domain is motivated by the need in automatic tools to facilitate the design of these circuits which, for various reasons, is still lagging behind the digital circuit design.

The paper is organized as follows. We recall the hybrid systems testing framework in Section 2 and the test generation algorithm **gRRT** in Section 3, where we also discuss the controllability issue. In Section 4 we introduce the disparity notion and how to use it to enhance the coverage quality of tests. The last section is devoted to the experimental results obtained on some benchmarks of analog and mixed signal circuits.

2 Testing Problem

As a model for hybrid systems, we use hybrid automata. In most classic versions of hybrid automata, continuous dynamics are defined using ordinary differential equations. However, with a view to applications to circuits where continuous dynamics are often described by differential algebraic equations, we adapt the model to capture this particularity.

A *hybrid automaton* is a tuple $\mathcal{A} = (\mathcal{X}, Q, E, F, \mathcal{I}, \mathcal{G}, \mathcal{R})$ where $\mathcal{X} \subseteq \mathbb{R}^n$ is the continuous state space; Q is a finite set of locations; E is a set of discrete transitions; $F = \{F_q \mid q \in Q\}$ such that for each $q \in Q$, $F_q = (U_q, W_q, f_q)$ defines a differential algebraic equation of the form[1]: $f_q(x(t), \dot{x}(t), u(t), w) = 0$ where the input signal $u : \mathbb{R}_+ \to U_q \subset \mathbb{R}^p$ is piecewise continuous and $w \in W_q \subset \mathbb{R}^m$ is the parameter vector; $\mathcal{I} = \{\mathcal{I}_q \subseteq \mathbb{R}^n \mid q \in Q\}$ is a set of staying conditions; $\mathcal{G} = \{\mathcal{G}_e \mid e \in E\}$ is a set of guards such that for each discrete transition $e = (q, q') \in E$, $\mathcal{G}_e \subseteq \mathcal{I}_q$; $\mathcal{R} = \{\mathcal{R}_e \mid e \in E\}$ is a set of reset maps. For each $e = (q, q') \in E$, $\mathcal{R}_e : \mathcal{G}_q \to 2^{\mathcal{I}_{q'}}$ defines how x may change when $\mathcal{A}$ switches from q to q'. The hybrid state space is $\mathcal{S} = Q \times \mathcal{X}$. A state (q, x) of $\mathcal{A}$ can change by *continuous evolution* and by *discrete evolution*. In location q, the continuous evolution of x is governed by the differential algebraic equation $f_q(x(t), \dot{x}(t), u(t), w) = 0$. Let $\phi(t, x, u(\cdot))$ be the solution of this equation with the initial condition x and under the input $u(\cdot)$. A *continuous transition* $(q, x) \overset{u(\cdot),h}{\to} (q, x')$ where $h > 0$ means that $x' = \phi(h, x, u(\cdot))$ and for all $t \in [0, h] : \phi(t, x, u(\cdot)) \in \mathcal{I}_q$. We say that $u(\cdot)$ is *admissible* starting at (q, x) for h time. For a state (q, x), if there exists a transition $e = (q, q') \in E$ and that $x \in \mathcal{G}_e$, then the transition e is enabled, the system can switch to q' and the continuous variables become $x' \in \mathcal{R}_e(x)$. This is denoted by $(q, x) \overset{e}{\to} (q', x')$, and we say that the discrete transition e is admissible at (q, x). The hybrid automata we consider are assumed to be *non-Zeno*. Note that this model is *non-deterministic* (both in continuous and discrete dynamics). To define our testing framework, we need the notions of *inputs* and *observations*.

If an input is *controllable by the tester*[2], it is called a *control input*; otherwise, it is called a *disturbance input*. A control action can be *continuous* or *discrete*. We assume that all the continuous inputs of the system are controllable. Since we want to implement the tester as a computer program, we are interested in continuous input functions that are piecewise-constant. Hence, a *continuous control action*, such as $(\bar{u}_q, h)$ specifies that the system continues with the dynamics F_q under the input $u(t) = \bar{u}_q$ for exactly h time.

For a state (q, x), a sequence of input actions $\omega = \iota_0, \iota_1, \ldots, \iota_k$ is admissible at (q, x) if: (1) ι_0 is admissible at (q, x), and (2) for each $i = 1, \ldots, k$, if (q_i, x_i) be the state such that $(q_{i-1}, x_{i-1}) \overset{\iota_{i-1}}{\to} (q_i, x_i)$, then ι_i is admissible at (q_i, x_i). The sequence $(q, x), (q_1, x_1) \ldots, (q_k, x_k)$ is called the *trace* starting at (q, x) under ω and is denoted by $\tau((q, x), \omega)$.

Since the tester cannot manipulate uncontrollable actions, we need the notion of admissible control action sequences. However, due to space limitation, we do not include its formal definition, which can be found in [3]. Intuitively, if we apply a control action sequence to the automaton, some disturbance actions can

[1] We assume the existence and uniqueness of solutions of these equations.

[2] By 'controllable' here we mean that the tester can manipulate this input, and it should not be confused with the term 'controllable' in control theory.

occur between the control actions. A control action sequence that does not cause the automaton to be blocked is called admissble. The set of traces starting at (q, x) after an admissible control action sequence ω is denoted by $Tr((q, x), \omega)$. We denote by $S_{\mathcal{C}}(\mathcal{A})$ the *set of all admissible control action sequences* starting at an initial state (q_{init}, x_{init}).

We assume that the location of the hybrid automaton $\mathcal{A}$ is observable by the tester. We also assume a set $V_o(\mathcal{A})$ of observable continuous variables of $\mathcal{A}$. The projection of a continuous state x of $\mathcal{A}$ on $V_o(\mathcal{A})$, denoted by $\pi(x, V_o(\mathcal{A}))$, is called an *observation.* The projection can be then defined for a trace. Let ω be an admissible control action sequence starting at an initial state (q_{init}, x_{init}) of $\mathcal{A}$. The set of *observation sequences* associated with ω is $S_{\mathcal{O}}(\mathcal{A}, \omega) = \{\pi(\tau, V_o(\mathcal{A})) \mid \tau \in Tr((q_{init}, x_{init}), \omega)\}$.

In our framework, the specification is modeled by a hybrid automaton $\mathcal{A}$ and the system under test SUT (e.g. an implementation) by another hybrid automaton $\mathcal{A}_s$ such that $V_o(\mathcal{A}) \subseteq V_o(\mathcal{A}_s)$ and $S_{\mathcal{C}}(\mathcal{A}) \subseteq S_{\mathcal{C}}(\mathcal{A}_s)$ (that is, a control sequence which is admissible for $\mathcal{A}$ is also admissible for $\mathcal{A}_s$). Note that we do not assume that we know the model $\mathcal{A}_s$. The goal of testing is to make statements about the relation between the traces of the SUT and those of the specification. The tester performs experiments on $\mathcal{A}_s$ in order to study the relation between $\mathcal{A}$ and $\mathcal{A}_s$. It emits an admissible control sequence to the SUT and measures the resulting observation sequence in order to produce a verdict ('pass', or ' fail', or 'inconclusive'). The observations are measured at the end of each continuous control action and after each discrete (disturbance or control) action.

Definition 1 (Conformance). *The system under test $\mathcal{A}_s$ is conform to the specification $\mathcal{A}$, denoted by $\mathcal{A} \approx \mathcal{A}_s$, iff $\forall \omega \in S_{\mathcal{C}}(\mathcal{A}) :\ \pi(S_{\mathcal{O}}(\mathcal{A}_s, \omega), V_o(\mathcal{A})) \subseteq S_{\mathcal{O}}(\mathcal{A}, \omega)$.*

A *test case* is represented by a (finite) tree where each node is associated with an observation and each edge with a control action. A hybrid automaton might have an infinite number of infinite traces; however, the tester can only perform a finite number of test cases in finite time. Therefore, we need to select a finite portion of the input space of $\mathcal{A}$ and test the conformance of $\mathcal{A}_s$ with respect to this portion. The selection is done using a *coverage criterion* that we formally define in the following.

Star Discrepancy Coverage. We are interested in defining a coverage measure that describes how 'well' the visited states represent the reachable set. This measure is defined using the *star discrepancy*, which is an important notion in equidistribution theory as well as in quasi-Monte Carlo techniques [1].

We first define the coverage for each location. Since a hybrid system can only evolve within the staying sets of the locations, we are interested in the coverage over these sets. For simplicity we assume that all the staying sets are boxes. If a staying set $\mathcal{I}_q$ is not a box, we can take the smallest oriented box that encloses $\mathcal{I}_q$, and apply the star discrepancy definition to the oriented box after an appropriate coordination change. Let P be a set of k points inside $\mathcal{B} = [l_1, L_1] \times \ldots \times [l_n, L_n]$, which is the staying set of the location q. Let Γ

be the set of all sub-boxes J of the form $J = \prod_{i=1}^{n}[l_i, \beta_i]$ with $\beta_i \in [l_i, L_i]$ (see Figure 1 for an illustration). The local discrepancy of the point set P with respect to the subbox J is $D(P, J) = |\frac{A(P, J)}{k} - \frac{\lambda(J)}{\lambda(\mathcal{B})}|$ where $A(P, J)$ is the number of points of P inside J, and $\lambda(J)$ is the volume of J. The star discrepancy of P with respect to the box $\mathcal{B}$ is defined as: $D^*(P, \mathcal{B}) = sup_{J \in \Gamma} D(P, J)$. Note that $0 < D^*(P, \mathcal{B}) \leq 1$.

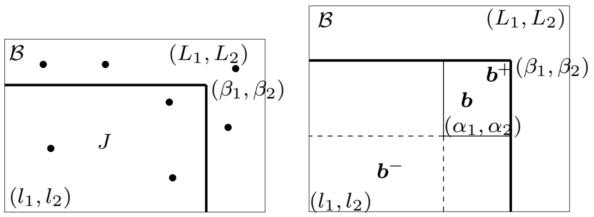


Fig. 1. Illustration of the star discrepancy notion

Intuitively, the star discrepancy is a measure for the irregularity of a set of points. A large value $D^*(P, \mathcal{B})$ means that the points in P are not well equidistributed over $\mathcal{B}$. When the region is a hyper-cube, the star discrepancy measures how badly the point set estimates the volume of the cube.

Let $\mathcal{P} = \{(q, P_q) \mid q \in Q \ \wedge \ P_q \subset \mathcal{I}_q\}$ be the set of hybrid states. We define the coverage for each location $q \in Q$ as $Cov_q(\mathcal{P}) = 1 - D^*(P_q, \mathcal{I}_q)$.

Definition 2 (Star discrepancy coverage). *The coverage of* $\mathcal{P}$ *is defined as:* $Cov(\mathcal{P}) = \frac{1}{||Q||} \sum_{q \in Q} Cov_q(\mathcal{P})$ *where* $||Q||$ *is the number of locations in* Q.

3 Coverage Guided Test Generation

In this section we recall the **gRRT** algorithm [9], which is a combination of the Rapidly-exploring Random Tree (**RRT**) algorithm, a successful robot motion planning technique (see for example [15]), and a guiding tool used to achieve a good test coverage. Essentially, the algorithm constructs a tree $\mathcal{T}$ as follows (see Algorithm 1). First, from the set of initial states, we sample a finite number of initial states, each corresponds to a root of the tree. Similarly, we can consider a finite number of parameter values and associate them with each initial state. Along a path from each root, the parameter vector remains constant.

In each iteration, a goal state s_{goal} is sampled. A neighbor $s_{near} = (q_{near}, x_{near})$ of s_{goal} is then determined. This neighbor is used as the starting state for the current iteration. To define a neighbor of a state, we define a hybrid distance from $s_1 = (q_1, x_1)$ to $s_2 = (q_2, x_2)$ as an average length of all the potential traces from s_1 to s_2 (see [9] for more detail). In ContinuousStep, we want to find an input $\bar{u}_{q_{near}}$ to take the system from s_{near} towards s_{goal} as closely as possible after h time, which results in a new state s_{new}. To find $\bar{u}_{q_{near}}$, when the set U

is not finite it can be sampled, or one can solve a local optimal control problem. Then, from s_{new}, we compute its successors by all possible discrete transitions. Note that the simulator can also detect the uncontrollable discrete transitions that become enabled during a continuous step, and in this case the current continuous step is stopped (which is equivalent to using a variable time step). The algorithm terminates after some maximal number of iterations. To extract a test case from the tree, we project the states at the nodes on the observable variables of $\mathcal{A}$.

In the classic **RRT** algorithms, which work in a continuous setting, only x_{goal} needs to be sampled, and a commonly used sampling distribution of x_{goal} is uniform over the state space. In addition, the point x_{near} is defined as a nearest neighbor of x_{goal} in some usual distance, such as the Euclidian distance. In our **gRRT** algorithm, the goal state sampling is not uniform and the function GuidedSampling plays the role of guiding the exploration by sampling of goal states according to the current coverage of the visited states.

Algorithm 1. Test generation algorithm **gRRT**

```
$\mathcal{T}.init(s_{init})$, $j = 1$                          ▷ $s_{init}$: initial state
repeat
    $s_{goal}$ = GuidedSampling($\mathcal{S}$)                  ▷ $\mathcal{S}$: hybrid state space
    $s_{near}$ = Neighbor($\mathcal{T}, s_{goal}$)
    $(s_{new}, \bar{u}_{q_{near}})$ = ContinuousStep($s_{near}, h$)
    DiscreteSteps($\mathcal{T}, s_{new}$), $j{+}{+}$
until $j \geq J_{max}$
```

Coverage guided sampling. To evaluate the coverage of a set of states, we estimate a lower and upper bound of the star discrepancy (exact computation is well-known to be a hard problem). These bounds as well as the information obtained from their estimation are used to decide which parts of the state space have been 'well explored' and which parts need to be explored more. Let us briefly describe this estimation method (see [9]). Let $\mathcal{B} = [l_1, L_1] \times \ldots \times [l_n, L_n]$. We define a box partition of $\mathcal{B}$ as a set of boxes $\Pi = \{\boldsymbol{b}^1, \ldots, \boldsymbol{b}^m\}$ such that $\cup_{i=1}^m \boldsymbol{b}^i = \mathcal{B}$ and the interiors of the boxes $\boldsymbol{b}^i$ do not intersect. Each such box is called an *elementary box.* Given a box $\boldsymbol{b} = [\alpha_1, \beta_1] \times \ldots \times [\alpha_n, \beta_n] \in \Pi$, we define $\boldsymbol{b}^+ = [l_1, \beta_1] \times \ldots \times [l_n, \beta_n]$ and $\boldsymbol{b}^- = [l_1, \alpha_1] \times \ldots \times [l_n, \alpha_n]$ (see Figure 1). For any finite box partition Π of $\mathcal{B}$, an upper bound $B(P, \Pi)$ and a lower bound $C(P, \Pi)$ of the star discrepancy $D^*(P, \mathcal{B})$ can be written as: $B(P, \Pi) = \max_{\boldsymbol{b} \in \Pi} \max\{\frac{A(P, \boldsymbol{b}^+)}{k} - \frac{\lambda(\boldsymbol{b}^-)}{\lambda(\mathcal{B})}, \frac{\lambda(\boldsymbol{b}^+)}{\lambda(\mathcal{B})} - \frac{A(P, \boldsymbol{b}^-)}{k}\}$ and $C(P, \Pi) = \max_{\boldsymbol{b} \in \Pi} \max\{|\frac{A(P, \boldsymbol{b}^-)}{k} - \frac{\lambda(\boldsymbol{b}^-)}{\lambda(\mathcal{B})}|, |\frac{A(P, \boldsymbol{b}^+)}{k} - \frac{\lambda(\boldsymbol{b}^+)}{\lambda(\mathcal{B})}|\}$.

To sample a goal state, we first sample a discrete location and then a continuous state. Let $\mathcal{P} = \{(q, P_q) \mid q \in Q \wedge P_q \subset \mathcal{I}_q\}$ be the current set of visited states. The discrete location sampling distribution depends on the current coverage of each location: $Pr[q_{goal} = q] = \dfrac{D^*(P_q, \mathcal{I}_q)}{\sum_{q' \in Q} D^*(P_{q'}, \mathcal{I}_{q'})}$. To sample x_{goal}, we first sample an elementary box $\boldsymbol{b}_{goal}$ from the set Π, then we sample a point x_{goal} in $\boldsymbol{b}_{goal}$ uniformly. The elementary box sampling distribution is biased in order to improve the coverage. We favor the selection of an elementary box such that a new point x added in this box results in a reduction of the lower and upper bounds(see [9]).

To demonstrate the performance of **gRRT**, we use two illustrative examples. For brevity, we call the classical **RRT** algorithm using uniform sampling and the Euclidian metric **hRRT**. The reason we choose these examples is that they differ in the reachability property. In the first example, the system is 'controllable' in the sense that the whole state space is reachable from the initial states (by using appropriate inputs), but in the second example the reachable set is only a small part of the state space.

Example 1. This is a two-dimensional continuous system where the state space $\mathcal{X}$ is a box $\mathcal{B} = [-3, 3] \times [-3, 3]$. The continuous dynamics is $f(x, t) = u(t)$ where the input set is $U = \{u \in \mathbb{R}^2 \mid ||u|| \leq 0.2\}$.

We use 100 input values resulting from a discretization of the set U. The initial state is $(-2.9, -2.9)$. The time step is 0.002. Figure 2 shows the result obtained using **gRRT** and the evolution of the coverage of the states generated by **gRRT** (solid curve) and by **hRRT** (dashed curve). The figure indicates that **gRRT** achieved a better coverage quality, especially in convergence rate.

Example 2. This example is a linear system with a stable focus at the origin. Its dynamics is as follows: $\dot{x} = -x - 1.9y + u_1$ and $\dot{y} = 1.9x - y + u_2$. We let the dynamics be slightly perturbed by an additive input u. The state space is the box $\mathcal{B} = [-3, 3] \times [-3, 3]$. The input set $U = \{u \in \mathbb{R}^2 \mid ||u|| \leq 0.2\}$. Figure 3 shows the results obtained after 50000 iterations. We can see that again the guided sampling method achieved a better coverage result.

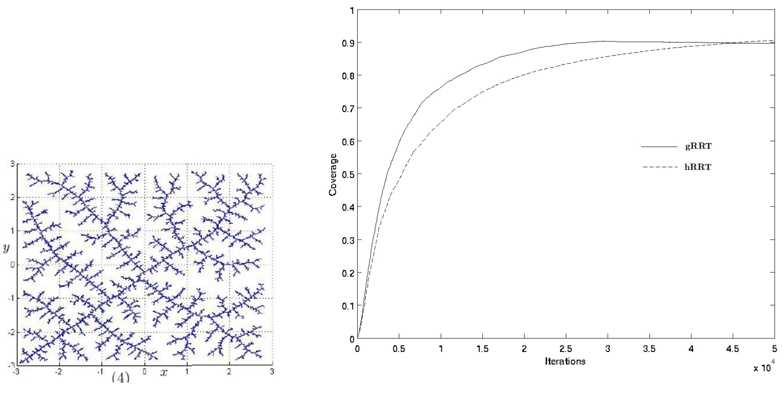


Fig. 2. Left: The **gRRT** exploration result. Right: Test coverage evolution.

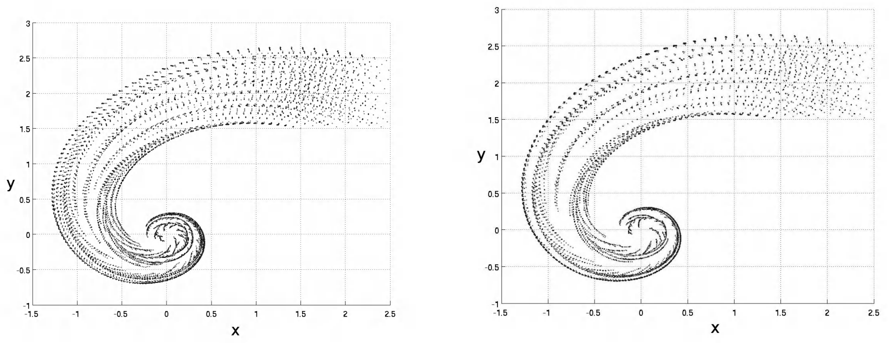


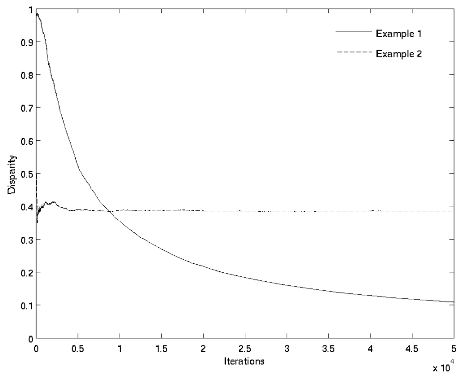
Fig. 3. Results obtained using the guided sampling method (left) and using the uniform sampling method (right)

Controllability Issue. From different experiments with Example 2, we observed that the coverage performance of **gRRT** is not satisfying when the reachable space is only a small part of the whole state space. This can be explained as follows. There are boxes, such as those near the bottom right vertex of the bounding box, which have a high potential of reducing the bounds of the star discrepancy. Thus, the sampler frequently selects these boxes. However, these boxes are not reachable from the initial states, and all attempts to reach them do not expand the tree beyond the boundary of the reachable set. This results in a large number of points concentrated near this part of the boundary, while other parts of the reachable set are not well explored.

It is important to emphasize that this problem is not specific to **gRRT**. The **RRT** algorithm using the uniform sampling method and, more generally, any algorithm that does not take into account the differential contraints of the system, may suffer from this phenomenon. This phenomenon can however be captured by the evolution of the disparity between the set of goal states and the set of visited states. This notion will be formally defined in the next section. Roughly speaking, it describes how different their distributions are. When the disparity does not decrease after a certain number of iterations, this often indicates that the system cannot approach the goal states, and it is better not to favor an expansion towards the exterior but a *refinement*, that is an exploration in the interior of the already visited regions.

Figure 4 shows the evolution of the disparity between the set P^k of visited states at the k^{th} iteration and the set G^k of goal states for the two examples. We observe that for the system of Example 1 which can reach any state in the state space, the visited states follow the goal states, and thus the disparity gets stabilized over time. However, in Example 2, where the system cannot reach everywhere, the disparity does not decrease for a long period of time, during which most of the goal states indicate unreachable directions.

Figure 4 shows the Voronoi diagram of a set of visited states. The boundary of the reachable set can be seen as an 'obstacle' that prevents the system from crossing it. Note that the Voronoi cells of the states on the boundary are large



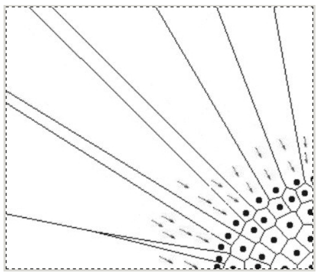


Fig. 4. Left: Disparity between the visited states and the goal states

(because they are near the large unvisited part of the state space). Hence, if the goal states are uniformly sampled over the whole state space, these large Voronoi cells have higher probabilities of containing the goal states, and thus the exploration is 'stuck' near the boundary, while the interior of the reachable set is not well explored.

To tackle this problem, we introduce the notion of disparity to describe the 'difference' in the distributions of two sets of points. The controllability problem can be detected by a large value of the disparity between the goal states and the visited states. We can thus combine **gRRT** with a disparity based sampling method, in order to better adapt to the dynamics of the system. This is the topic of the next section.

4 Disparity Guided Sampling

The notion of disparity between two point sets that we develop here is inspired by the star discrepancy. Indeed, by definition, the star discrepancy of a set P w.r.t. the box $\mathcal{B}$ can be seen as a comparison between P and an 'ideal' infinite set of points distributed all over $\mathcal{B}$. Let P and Q be two sets of points inside $\mathcal{B}$. Let J be a sub-box of $\mathcal{B}$ which has the same bottom-left vertex as $\mathcal{B}$ and the top-right vertex of which is a point inside $\mathcal{B}$. Let Γ be the set of all such sub-boxes. We define the local disparity between P and Q with respect to the sub-box J as: $\gamma(P,Q,J) = |\dfrac{A(P,J)}{||P||} - \dfrac{A(Q,J)}{||Q||}|$ where $A(P,J)$ is the number of points of P inside J and $||P||$ is the total number of points of P.

Definition 3 (Disparity). *The disparity between P and Q with respect to the bounding box $\mathcal{B}$ is defined as: $\gamma^*(P,Q,\mathcal{B}) = sup_{J\in\Gamma}\gamma(P,Q,J)$.*

The disparity satisfies $0 < \gamma^*(P,Q,\mathcal{B}) \leq 1$. A small value $\gamma^*(P,Q,\mathcal{B})$ means that the distributions of the sets P and Q over the box $\mathcal{B}$ are 'similar'. To illustrate

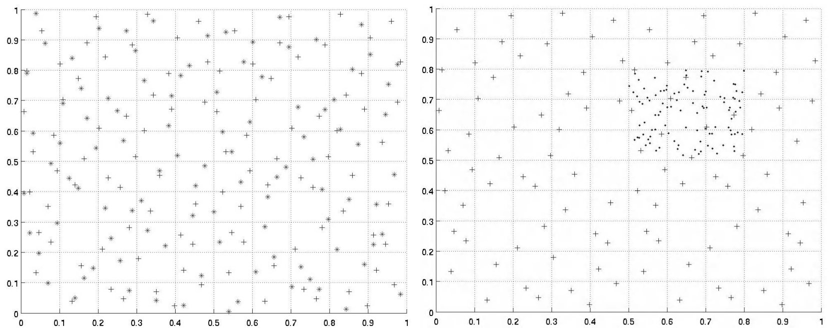


Fig. 5. Left: Faure sequence (+ signs) and Halton sequence ($*$ signs). Right: Faure sequence (+ signs) and another C pseudo-random sequence ($*$ signs).

our notion of disparity, we consider two well-known sequences of points: the Faure and Halton sequences [6,14], shown in Figure 5. Their disparity is 0.06, indicating that they have similar distributions. The second example is used to compare the Faure sequence and a set of 100 points concentratred in some small rectangle, and the disparity between them is large (0.54).

The exact computation of the disparity is as hard as the exact computation of the star discrepancy, which is due to the infinite number of the sub-boxes. We propose a method for estimating a lower and an upper bound for this new measure. Let Π be a box partition of $\mathcal{B}$. Let P, Q be two sets of points inside $\mathcal{B}$. For each elementary box $\boldsymbol{b} \in \Pi$ we denote $\mu_m(\boldsymbol{b}) = \max\{\mu_c(\boldsymbol{b}), \mu_o(\boldsymbol{b})\}$ where $\mu_c(\boldsymbol{b}) = \dfrac{A(P, \boldsymbol{b}^+)}{||P||} - \dfrac{A(Q, \boldsymbol{b}^-)}{||Q||}$, $\mu_o(\boldsymbol{b}) = \dfrac{A(Q, \boldsymbol{b}^+)}{||Q||} - \dfrac{A(P, \boldsymbol{b}^-)}{||P||}$. We also denote $c(\boldsymbol{b}) = \max\{|\dfrac{A(P, \boldsymbol{b}^-)}{||P||} - \dfrac{A(Q, \boldsymbol{b}^-)}{||Q||}|, |\dfrac{A(P, \boldsymbol{b}^+)}{||P||} - \dfrac{A(Q, \boldsymbol{b}^+)}{||Q||}|\}$.

Theorem 1. [Upper and lower bounds] *An upper bound $B_d(P, Q, \Pi)$ and a lower bound $C_d(P, Q, \Pi)$ of the disparity between P and Q are: $B_d(P, Q, \Pi) = \max_{\boldsymbol{b} \in \Pi}\{\mu_m(\boldsymbol{b})\}$ and $C_d(P, Q, \Pi) = \max_{\boldsymbol{b} \in \Pi}\{c(\boldsymbol{b})\}$.*

For each elementary box $\boldsymbol{b} = [\alpha_1, \beta_1] \times \ldots \times [\alpha_n, \beta_n] \in \Pi$, we define the $\mathcal{W}$-zone, denoted by $\mathcal{W}(\boldsymbol{b})$, as follows: $\mathcal{W}(\boldsymbol{b}) = \boldsymbol{b}^+ \setminus \boldsymbol{b}^-$.

Theorem 2. *A bound on the disparity estimation error is:*

$$B_d(P, Q, \Pi) - C_d(P, Q, \Pi) \leq \max_{\boldsymbol{b} \in \Pi} \max\{\frac{A(P, \mathcal{W}(\boldsymbol{b}))}{||P||}, \frac{A(Q, \mathcal{W}(\boldsymbol{b}))}{||Q||}\}.$$

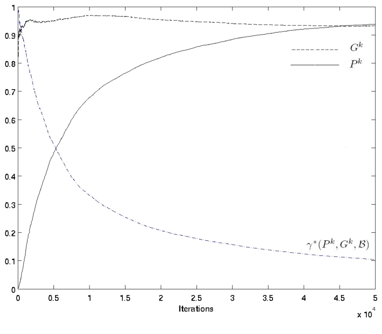
The above error bounds can be used to dynamically refine the partition. The proofs of the above results can be found in [3].

Disparity guided sampling. The essential idea of our disparity based sampling method is to detect when the dynamics of the system does not allow the tree to expand towards the goal states and then to avoid such situations by favoring a refinement, that is an exploration near the already visited states.

A simple way to bias the sampling towards the set P^k of already visited states is to reduce the sampling space. Indeed, we can make a bounding box of the set P^k and give more probability of sampling inside this box than outside it. Alternatively, we can guide the sampling using the disparity information as follows. The objective now is to reduce the disparity between the set G^k of goal states and the set P^k of visited states. Like the guiding method using the star discrepancy, we define for each elementary box $\boldsymbol{b}$ of the partition a function $\eta(\boldsymbol{b})$ reflecting the potential for reduction of the lower and upper bounds of the disparity between P^k and G^k. This means that we favor the selection of the boxes that make the distribution of goal states G^k approach that of the visited states P^k. Choosing such boxes can improve the quality of refinement. The formulation of the potential influence function for the disparity-based sampling method is similar to that for the coverage guided sampling.

A combination of the coverage guided and the disparity guided sampling methods is done as follows. We fix a time window T_s and a threshold ϵ. When using the coverage guided method, if the algorithm detects that the disparity between the G^k and P^k does not decrease by ϵ after T_s time, it switches to the disparity guided method until the disparity is reduced more significantly and switches back to the coverage guided method. Note that a non-decreasing evolution of the disparity is an indication of the inability of the system to approach the goal states. In an interactive exploration mode, it is possible to let the user to manually decide when to switch. As mentioned earlier, we call the resulting algorithm **agRRT** (the letter 'a' in this acronym stands for 'adaptive').

We use the examples in the previous section to demonstrate the coverage improvement of **agRRT**. Figure 6 shows that the final result for Example 1 obtained using **agRRT** has a better coverage than that obtained using **gRRT**. The solid curve represents the coverage of the set P^k of visited states and the dashed one the coverage of the set G^k of goal states. The dash-dot curve represents the disparity between G^k and P^k. The result obtained using **agRRT** for Example 2 is shown in Figure 7, which also indicates an improvement in coverage quality. The figure on the right shows the set of generated goal states, drawn in dark color. In this example, we can observe the adaptivity of the



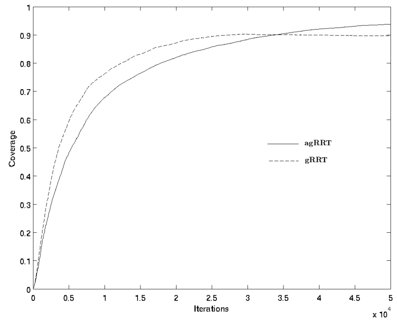


Fig. 6. Left: Test coverage of the result obtained using **agRRT** for Example 1

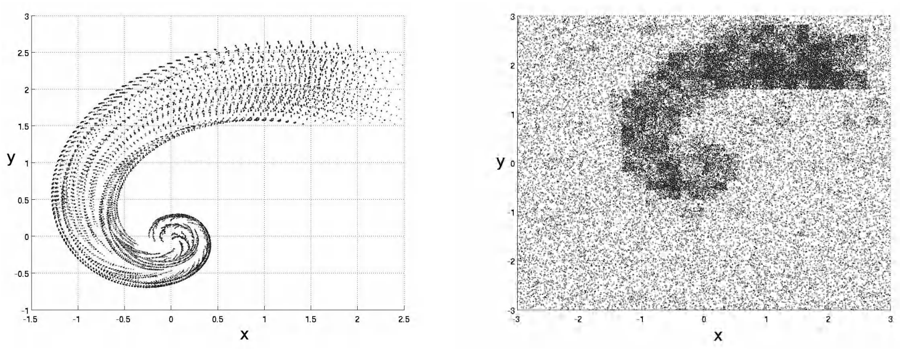


Fig. 7. Result obtained using **agRRT**. Left: visited states P^k, right: goal states G^k.

combination of **gRRT** and **agRRT**. Indeed, in the beginning, the **gRRT** algorithm was used to rapidly expand the tree. After some time, the goal states sampled from the outside of the exact reachable space do not improve the coverage, since they only create more states near the boundary of the reachable set. In this case, the disparity between P^k and G^k does not decrease, and the **agRRT** is thus used to enable an exploration in the interior of the reachable set. The interior the reachable set thus has a higher density of sampled goal states than the outside, as one can see in the figure.

5 Applications to Analog and Mixed-Signal Circuits

Using the above results, we implemented a test generation tool and tested it on a number of control applications, which proved its scalability to high dimensional systems [9]. In this implementation, all the sets encountered in the hybrid automaton definition are convex polyhedra. For circuit applications, we use the well-known RADAU algorithm for solving differential algebraic equations (DAE). We recall that solving high index and stiff DAEs is computationally expensive, and in order to evaluate the efficiency of the test generation algorithm, we have chosen two practical circuits with DAEs of this type. The three circuits we treated are: a transistor amplifier, a voltage controlled oscillator, and a Delta-Sigma modulator circuit. As described earlier, we use a criterion based on the disparity to automatically detect the controllability problem and to switch to the disparity guided strategy for some time. This criterion is necessary because it is not easy to anticipate whether the system under study is 'highly controllable' or not. However, prior knowledge of the system's behavior can be helpful in the decision of favoring one strategy over the other. Among the circuit benchmarks we treated, the Delta-Sigma circuit has an expansive dynamics and thus we could predict that the coverage-guided strategy is appropriate. However, for the transistor amplifier and the voltage controlled oscillator circuit, a more frequent use of the disparity guided strategy was necessary.

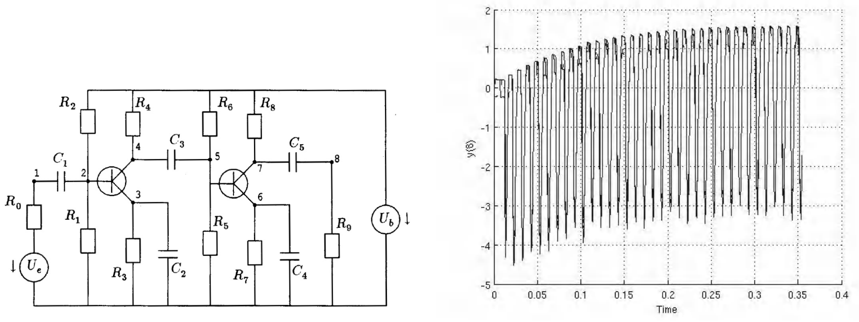


Fig. 8. Test generation result for the transistor amplifier

Transistor amplifier. The first example is a transistor amplifier model [4], shown in Figure 8, where the variable y_i is the voltage at node i; U_e is the input signal and $y_8 = U_8$ is the output voltage. The circuit equations are a system of non-linear DAEs of index 1 with 8 continuous variables.The circuit parameters are: $U_b = 6$; $U_F = 0.026$; $R_0 = 1000$; $R_k = 9000$, $k = 1, \ldots, 9$; $C_k = k10^{-6}$; $\alpha = 0.99$; $\beta = 10^{-6}$. The initial state $y_{init} = (0, U_b/(R_2/R_1 + 1), U_b/(R_2/R_1 + 1), U_b, U_b/(R_6/R_5 + 1), U_b/(R_6/R_5 + 1), U_b, 0)$.

To study the influence of circuit parameter uncertainty, we consider is a perturbation in the relation between the current through the source of the two transistors and the voltages at the gate and source $I_S = g(U_G - U_S) = \beta(e^{\frac{U_G - U_S}{U_F}} - 1) + \epsilon$, with $\epsilon \in [\epsilon_{min}, \epsilon_{max}] = [-5e-5, 5e-5]$. The input signal $U_e(t) = 0.1 sin(200\pi t)$. The acceptable interval of U_8 in the non-perturbed circuit is $[-3.01, 1.42]$. Once the initial transient period has settled down, the generated test case leads to an overshoot after 18222 iterations (corresponding to 1.1mn of computation time). The total computation time for generating 50000 states was 3mn. Figure 8 shows the generated states projected on U_8 over the first 0.03 seconds.

Voltage controlled oscillator. The circuit [7] is described by a system of DAEs with 55 continuous variables. The oscillating frequency of the variables v_{C_1} and v_{C_2} is a linear function of the input voltage u_{in}. We study the influence of a time-variant perturbation in C_2, modeled as an input signal, on this frequency. We show that, in addition to conformance relation, using this framework, we can test a property of the input/output relation. The oscillating period $T \pm \delta$ of v_{C_2} can be expressed using a simple automaton with one clock y in Figure 9. The question is to know if given an oscillating trace in $\mathcal{A}$, its corresponding trace in $\mathcal{A}_s$ is also oscillates with the same period. This additional automaton can be used to determine test verdicts for the traces in the computed test cases. If an observation sequence corresponds to a path entering the 'Error' location, then it causes a 'fail' verdict. Since we cannot use finite traces to prove a safety property, the set of obsevation sequences that cause a 'pass' verdict is empty, and therefore the remaining obsevation sequences (that do not cause a 'fail'

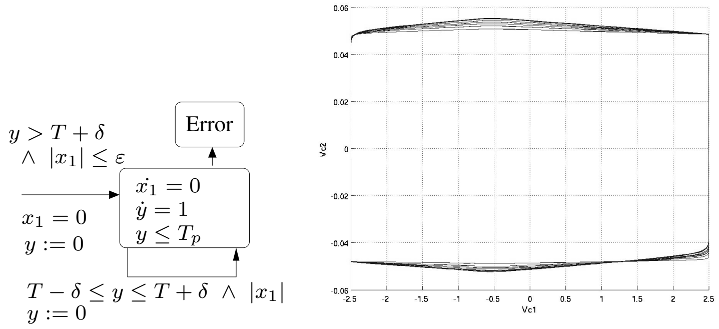

Error
$y > T + \delta \wedge |x_1| \leq \varepsilon$
$\dot{x_1} = 0$
$\dot{y} = 1$
$y \leq T_p$
$x_1 = 0$
$y := 0$
$T - \delta \leq y \leq T + \delta \wedge |x_1|$
$y := 0$
Vc2
Vc1

Fig. 9. Left: Automaton for an oscillation specification. Right: Variable v_{C_2} over time.

verdict) cause a 'inconclusive' verdict. We consider a constant input voltage $u_{in} = 1.7$. The coverage measure was defined on the projection of the state space on v_{C_1} and v_{C_2}. The generated test case shows that *after the transient time*, under a time-variant deviation of C_2 which ranges within $\pm 10\%$ of the value of $C_2 = 0.1e - 4$, the variables v_{C_1} and v_{C_2} oscillate with the period $T \in [1.25, 1.258]s$ (with $\varepsilon = 2.8e - 4$). This result is consistent with the result presented in [7]. The number of generated states was 30000 and the computation time was 14mn. Figure 9 shows the explored traces of v_{C_2} over time.

Delta-Sigma circuit. We consider a third-order Delta-Sigma modulator [10], which is a mixed-signal circuit. When the input sinusoid is positive and its value is less than 1, the output takes the +1 value more often and the quantization error is fed back with negative gain and 'accumulated' in the integrator $\frac{1}{z-1}$. Then, when the accumulated error reaches a certain threshold, the quantizer switches the value of the output to -1 to reduce the mean of the quantization error. A third-order Delta-Sigma modulator is modeled as a hybrid automaton, shown in Figure 10. The discrete-time dynamics of the system is as follows: $x(k + 1) = Ax(k) + bu(k) - sign(y(k))a$, $y(k) = c_3x_3(k) + b_4u(k)$ where $x(k) \in \mathbb{R}^3$ is the

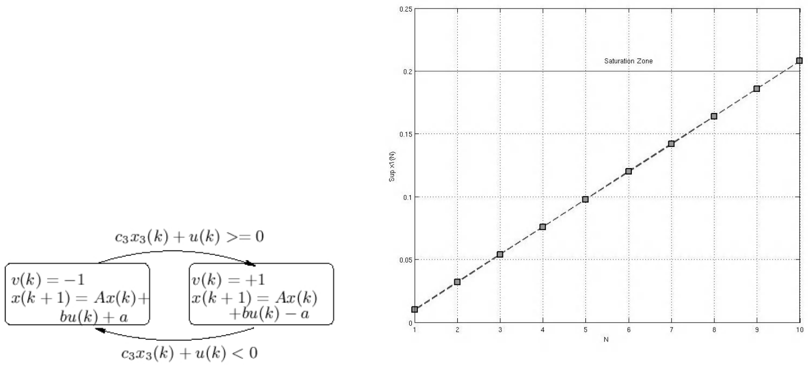


Fig. 10. Model of a third-order modulator and test generation result

integrator states, $u(k) \in \mathbb{R}$ is the input, $y(k) \in \mathbb{R}$ is the input of the quantizer. Thus, its output is $v(k) = sign(y(k))$, and one can see that whenever v remains constant, the system dynamics is affine continuous. A modulator is stable if under a bounded input, the states of its integrators are bounded. The test generation algorithm was performed for the initial state $x(0) \in [-0.01, 0.01]^3$ and the input values $u(k) \in [-0.5, 0.5]$. After exploring only 57 states, saturation was already detected. The computation time was less than 1 second. Figure 10 shows the values of $(\sup x_1(k))_k$ as a function of the number k of time steps. We can see that the sequence $(\sup x_1(k))_k$ leaves the safe interval $[-x_1^{sat}, x_1^{sat}] = [-0.2, 0.2]$, which indicates the instability of the circuit. This instability for a fixed finite horizon was also detected in [2] using an optimization-based method.

6 Related Work and Conclusions

In this paper, we described the **agRRT** algorithm which is a combination of the coverage guided test generation algorithm **gRRT** and a disparity guided algorithm. The latter uses the information about the disparity between the goal states and the visited states in order to steer the exploration towards the area where the dynamics of the system allows to better improve the test coverage. We provided some examples to show the efficiency of this guiding tool, in terms of coverage improvement. We also reported some experimental results where our test generation tool successfully treated a number of benchmarks in analog and mixed-signal circuits.

Concerning related work along this line, sampling the configuration space has been a fundamental problem in probabilistic motion planning. Our idea of guiding the simulation via the sampling process has some similarity with the sampling domain control [15]. In this work, the domains over which the goal points are sampled need to reflect the geometric and differential constraints of the system, and more generally, the controllability of the system. In [8], another method for biasing the exploration was proposed. The main idea of this method is to reduce the dispersion in an incremental manner. This idea is thus very close to the idea of our guiding method in spirit; however, their concrete realizations are different. This method tries to lower the dispersion by using K samples in each iteration (instead of a single sample) and then select from them a best sample by taking into account the feasibility of growing the tree towards it. Finally, we mention that the controllability issue was addressed in [5] where the number of successful iterations is used to define an adaptive biased sampling.

A number of directions for future research can be identified. First, we are interested in enriching our framework to capture partial observability and measurement imprecisions. To facilitate the application to practical circuits, we need a tool for automatic generation of hybrid automata from commonly-used circuit descriptions, such as SPICE netlists.

Acknowledgements. We thank E. Frazzoli, O. Maler, and S. LaValle for their valuable comments. This work is supported by the project ANR VAL-AMS.

References

1. Beck, J., Chen, W.W.L.: Irregularities of distribution. In: Acta Arithmetica, Cambridge University Press, Cambridge (1997)
2. Dang, T., Donze, A., Maler, O.: Verification of analog and mixed-signal circuits using hybrid systems techniques. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, Springer, Heidelberg (2004)
3. Dang, T., Nahhal, T.: Coverage-Guided Test Generation for Continuous and Hybrid Systems. Technical report, Verimag, Grenoble (2007)
4. Lubich, C., Hairer, E., Roche, M.: The numerical solution of differential-algebraic systems by Runge Kutta methods. Lecture Notes in Math., vol. 1409. Springer, Heidelberg (1989)
5. Esposito, J., Kim, J.W., Kumar, V.: Adaptive RRTs for validating hybrid robotic control systems. In: Workshop on Algorithmic Foundations of Robotics (2004)
6. Faure, H.: Discrépance de suites associées à un système de numération. Acta Arithmetica 41, 337–351 (1982)
7. Grabowski, D., Platte, D., Hedrich, L., Barke, E.: Time constrained verification of analog circuits using model-checking algorithms. Electr. Notes Theor. Comput. Sci. 153(3), 37–52 (2006)
8. Lindemann, S.R., LaValle, S.M.: Incrementally reducing dispersion by increasing Voronoi bias in RRTs. In: IEEE Conf. on Robotics and Automation (2004)
9. Nahhal, T., Dang, T.: Test coverage for continuous and hybrid systems. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 454–468. Springer, Heidelberg (2007)
10. Pérez-Verdú, B., Medeiro, F., Rodríguez-Vázquez, A.: Top-Down Design of High-Performance Sigma-Delta Modulators. Kluwer Academic Publishers, Dordrecht (2001)
11. Plaku, E., Kavraki, L.E., Vardi, M.Y.: Hybrid systems: From verification to falsification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, Springer, Heidelberg (2007)
12. Tan, L., Kim, J., Sokolsky, O., Lee, I.: Model-based testing and monitoring for hybrid embedded systems. In: Information Reuse and Integration IRI 2004 (2004)
13. Tretmans, J.: Testing concurrent systems: A formal approach. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 46–65. Springer, Heidelberg (1999)
14. Wang, X., Hickernell, F.: Randomized Halton sequences. Math. Comp. Modelling 32, 887–899 (2000)
15. Yershova, A., Jaillet, L., Simeon, T., LaValle, S.: Dynamic-domain rrts: Efficient exploration by controlling the sampling domain. In: IEEE Int. Conf. Robotics and Automation (2005)

Modeling Property Based Stream Templates with TTCN-3

Juergen Grossmann[1], Ina Schieferdecker[1], and Hans-Werner Wiesbrock[2]

[1] Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, D-10589 Berlin
{juergen.grossmann,ina.schieferdecker}@fokus.fraunhofer.de
[2] IT Power Consultants, Gustav-Meyer-Allee 25, D-13355 Berlin
hans-werner.wiesbrock@itpower.de

Abstract. Test evaluation and test assessment is a time consuming and resource intensive process. More than ever this holds for testing complex systems that emanate continuous or hybrid behavior. In this article we introduce an approach that eases the specification of black box tests for hybrid or continuous systems by means of signal properties applied for evaluation. A signal property allows the characterization of individual signal shapes. It is determined by the examination of the signal's value at time, the application of pre-processing functions (like first or higher order derivatives), and the analysis and detection of sequences of values that form certain shapes of a signal (e.g. local minima and maxima). Moreover we allow the combination of properties by logical connectives.

The solution provided in this paper is based on terms and concepts defined for Continuous TTCN-3 (CTTCN-3) [12, 11], an extension of the standardized test specification language TTCN-3 [4]. Thus, we treat signals as streams and integrate the notion of signal properties with the notion of stream templates like they are already defined in CTTCN-3. Moreover, we provide a formal foundation for CTTCN-3 streams, for a selected set of signal properties and for their integration in CTTCN-3.

1 Introduction

TTCN-3 [4, 6, 5] is a procedural testing language. In its current(and standardized) state TTCN-3 provides powerful means to test message-based and procedure-based system interactions. As such it is not capable of testing system that emanates continuous or hybrid behavior. To fill the gap and to transmit parts of the approved TTCN-3 methodology to continuous and hybrid systems as well, we introduced Continuous TTCN-3 (CTTCN-3) [12, 11] and enhanced the TTCN-3 core language to the requirements of continuous and hybrid behavior while introducing:

- the notions of time and sampling,
- the notions of streams, stream ports and stream variables, and
- the definition of an automaton alike control flow structure to support the specification of hybrid behavior.

While [12] concentrates on system stimulation and the integration of the newly introduced concepts with the existing TTCN-3 core language, the systematic evaluation of system reaction was not discussed in depth. In this article we will catch up and work out the notion of *signal properties* and *property based stream templates*. A signal property addresses a certain but abstract aspect of a signal shape (i.e. the signal's value at a certain point in time, the derivative of the signal, and certain behavioral aspects like rising edges, extremal values etc.). A property based stream template constitutes a predicate that is based on signal properties and can be used to specify the expected system behavior for a test run.

The specification of formal properties to denote the requirements on a hybrid system is well known from the theory of hybrid automata [1]. Given a set of formal system properties denoted in a temporal logic calculus, the reachability and liveness of the properties can by automatically checked if an appropriate system model exists [7, 8]. The Reactis tool environment [13] provides a similar approach to derive test cases from models that can be applied to the system under test (SUT).

In [3, 14] such predicates are used as an explicit part of a test specification to ease the assessment of a hybrid system's reaction. In [14] a systematic approach for the derivation of so called validation functions from requirements is described. The approach introduces the notion of signal properties and their respective concatenations to detect certain — sometime very complex — signal characteristics (e.g. value changes, increase and decrease of a signal as well as signal overshoots) during the execution of a black box test run. The solution is based on Matlab/Simulink. In [3] a graphical modeling tool is outlined that is dedicated to ease the specification of signal properties for the off-line evaluation of tests. Both approaches aim to systematically denote signal properties. In this article we concentrate on a proper integration of signal properties with the existing means for testing hybrid behaviour in CTTCN-3.

In Section 2 we will give a short overview of CTTCN-3. This includes the explanation of the concepts stream, stream port and stream template. Moreover, the overview includes the definition of a formal semantics for streams, which will be used later on to properly integrate the notion of signal properties. In Section 3 we will describe a guiding example to motivate our approach. In Section 4 we will introduce the term *signal property* and a suitable classification of signal properties, in Section 5 we will introduce the integration of property based stream templates with CTTCN-3, and Section 6 concludes the paper.

2 Continuous TTCN-3

CTTCN-3 is an extension of TTCN-3 that is properly specified in [12] and as yet a theoretical prototype. In the following we will provide a short introduction to the syntax and semantics of the main constructs in CTTCN-3.

2.1 Time

For CTTCN-3 we adopted the concept of a global clock and enhance it with the notion of sampling and sampled time. As in TTCN-3, all time values in CTTCN-3 are denoted as float values and represent time in seconds. For sampling, the discrete time model $t = k * \Delta$ is used. It has a fixed step size Δ with $t,\ \Delta \in \mathbb{R}^+$, $k \in \mathbb{N}$. Relative time, which is used for the definition of streams and templates, is considered to be completely synchronized to the global clock.

2.2 The Test System

The SUT is represented in terms of its interface — the so called test interface. A test interface is defined by a set of input and output ports. Each port can be characterized by its direction of communication (i.e. unidirectional input or output, or bidirectional), the data types being transported (e.g. boolean, integer, float, etc.), and its communication characteristics (i.e. message-based, procedure-based, or stream-based). In this article, we denote the input ports of a SUT as an n-tuple $\boldsymbol{x} = (x_1, x_2, .., x_n)$ and the output ports as an m-tuple $\boldsymbol{y} = (y_1, y_2, .., y_m)$ with $m, n \in \mathbb{N}$ (see Figure 1)[1]. Moreover, we define a set of data types called $\daleth$ to specify the information structure transferrable via ports. For each port x_n, y_m there is a set $X_n, Y_m \in \daleth$ defining the domain of the port.

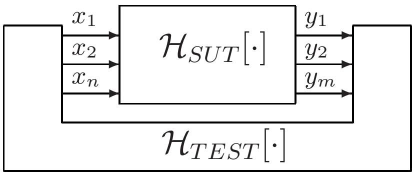


Fig. 1. An Black-box test system enclosing a SUT

System behavior is defined in terms of the given allocation of ports. Reactive system behavior can be denoted as an operator $\mathcal{T}[\cdot]$ that continuously operates on the inputs of the system [10]. The allocation of individual ports are defined by a function $f_{x_i}(t)$ over time. The complete System inputs over time — reflecting our definition of an SUT — are defined as an n-tuple,

$$\boldsymbol{x}(t) := (f_{x_1}(t), f_{x_2}(t), ..., f_{x_n}(t)) \quad with \quad \boldsymbol{x}(t) \in X_1^\epsilon \times X_2^\epsilon \times ... \times X_n^\epsilon.$$

The output of a system is defined by an equation system using the behavior operator $\mathcal{T}_{SUT}[\cdot]$.

$$\boldsymbol{y}(t) := \mathcal{T}_{SUT}[\boldsymbol{x}(t)] \quad with \quad \boldsymbol{y}(t) \in Y_1^\epsilon \times Y_2^\epsilon \times ... \times Y_m^\epsilon.$$

[1] A bidirectional port contributes both to the input and the output tuple of ports.

2.3 Streams

In contrast to scalar values, a stream [2, 9] represents the whole allocation history applied to a port. In CTTCN-3 the term stream is used to denote the data structure $s \in (STRM)_T$ that describes the complete history of data that yield as allocation of a certain port x_n, y_m. The index T denotes the type of a stream. It is defined as a cross product between a value domain $d \in \urcorner$ and the step size $\Delta \in \mathbb{R}^+$ with $T \in \urcorner \times \mathbb{R}^+$. In the following we only consider discrete (i.e. sampled) streams $s \in (DSTRM)_T \subset (STRM)_T$. A discrete stream s is represented by a structure $s := (\Delta, \langle m_k \rangle)$ where Δ represents the sample time, $\langle m_k \rangle$ a sequence of values (messages), and $s \in (DSTRM)_T$. The sequence of values is defined as follows:

$$\langle m_k \rangle := \{f_{x_i}(0), f_{x_i}(1 * \Delta), ..., f_{x_i}((k-1) * \Delta)\}$$

To obtain basic information on streams and their content we provide simple access operations. We distinguish between time-related and non-time-related access operations.

– For non-time-related access operations we use $\#s$ for the number of values and with $s[i],\ i \in \mathbb{N}$ we denote the i^{th} value in a stream s.
– Time-related access operations rely on a timing function $\tau_s(i) := (i-1) * \Delta$ with $i \in [1..(\#s)[$ that returns a time value for an arbitrary index value of a stream s. The operation $dur(s)$ returns the length of time for a stream s and is defined as $dur(s) = \tau_s(\#s)$. Further on we provide the operation $s@t$ to obtain the value m associated with an arbitrary point in time with $t \in \mathbb{R}^+$. The operation $s@t$ is defined as $s@(t) = s.i$ when $t \in [\tau_s(i), \tau_s(i+1)[$.

In CTTCN-3 we are able to explicitly declare *stream ports* and *stream variables* by the notion of *stream types* T. The step size Δ is defined using the keyword **sample**. Listing 1 shows the declaration of a sample and the declarations of a stream type, of two stream ports, as well as of a stream variable. Moreover the stream variable is initialized with a stream of infinite length.

Listing 1. Stream Types and Variables

```
sample(t)=1;
type stream float FloatStrm(t);
type port FloatOut stream {out FloatStrm}
type port FloatIn stream {in FloatStrm}

type component MyComponent {
  port FloatOut p1;
  port FloatIn p2;}

var FloatStrm myStrm:= sin(t);
```

2.4 Stream Templates

In TTCN-3, especially for the definition of the expected system reaction, the use of templates is encouraged. In [12] we advanced the notion of templates to be applicable to streams. We confined ourselves to templates for numerical streams and to the definition of upper and lower bounds only.

Similar to streams, stream templates $tp \in (TP)_T$ are classified by stream types T. Templates are generally applicable to streams of the same type or of compatible type[2]. In CTTCN-3 the application of a template to a stream or a stream port is carried out by either a *sense* statement (for the on-line evaluation of ports) or a *match* statement (for the offline evaluation of the data structure stream). The result of the application is dependent on the execution context. Inside the $carry-until$ statement, the template evaluation is carried out sample-wise, that is, it is defined as a function $\chi_{tp} : (DSTRM)_T \rightarrow (DSTRM)_{\mathbb{B}}$ where $r \in (DSTRM)_{\mathbb{B}}$ is a **stream** of boolean values $true$ or $false$.

Outside the carry-until construct the evaluation of a stream template is calculated as a whole, that is the complete stream is evaluated and the evaluation is defined as a function $\chi_{tp} : (DSTRM)_T \rightarrow \mathbb{B}$ where $r \in \mathbb{B}$ is one of the boolean values $true$ or $false$. In both cases the function $\chi_{tp}(s)$ is determined by the template definition TP. For more details concerning the meaning of stream template please refer to [12].

3 Guiding Example

The main objective of this article is to ease the specification of expected system behavior through the notion of signal properties (i.e. predicates) that, on the one hand can be used to closely describe the shape of individual signals, but also provide means to flexible address abstract characteristics of a signal. In order to motivate the concepts and constructs we present in this article we will start with a typical scenario that emanates from ECU testing in the automotive domain. In a drive case the tester activates the gas pedal, releases the pedal, and after a while he activates the break pedal. In the context of this example we are interested in the velocity control. Concerning the velocity, we expect a nearly linear increase at the beginning. That followed, the velocity remains constant for a short while to start slightly decreasing, and in the end it slows down to 0. Figure 2 shows a simple outline of our expectations.

In the following, we are looking for a feasible, that is abstract but exact, way to formerly describe our expectations. The crucial property we are interested in, is the signal's slope. Being more precisely, we expect the slope being nearly lets say 2.0 $[m/s^2]$ or nearly 0 or roughly -2.0 $[m/s^2]$. Moreover, we would like to address the respective durations and sequencing.

[2] Compatibility between stream types is dependent of the value domain $d \in \urcorner$ and the sampling domain $\Delta \in \mathbb{R}^+$. For the value domain we consider the given TTCN-3 compatibility rules. For the time domain we consider two types compatible when they obey the same sampling or one is a down sample of the other one.

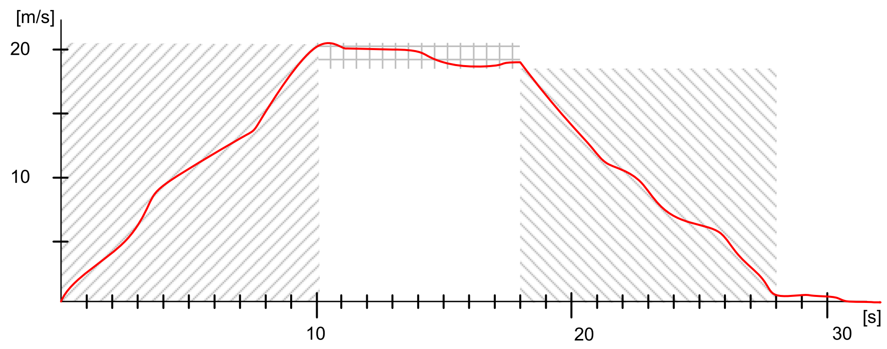


Fig. 2. Simple Shape of a Signal

In order to asses a concrete test result w.r.t. to our expectations, we have to denote our expectation in form of predicates, that closely characterizes the possible outcomes. Using the formalism sketched in the former paragraph, we end up with the following situation. The possible test outcome for some ECU is described by a real valued stream, i.e. the velocity measured at the sample times of our test run. The expectation that at the beginning the increase will be nearly linearly and we must check for it. We can formulate such predicates in terms of templates[3]. Heuristically we define:

Listing 2. Heuristic: Linear Increase

```
// in the time interval [0 s, 10 s],
// the derivative of the velocity is in [1.75..2.25]

template FloatStrm linearSlope_0_135@t:={
  @[0.0..10.0] := (differentiate(current) in [1.75..2.25])};
```

That is, the derivative in the time interval [0 s,10 s] is nearly constant. Regarding this example, one can obviously distinguish two different parts, the proper *predicate* and the *time scope*, that is the time interval the predicate is applied for. We could intuitively split this up and rewrite the expressions to:

Listing 3. Heuristic: Linear Increase, Time Scope and Predicate

```
template FloatStrm linearSlope@t:={
   (differentiate(current) in [1.75..2.25])}

template FloatStrm linearSlope_0_10@t:={
  @[0.0..10.0]:=linearSlope@t};
```

Furthermore, we would like to address the temporal segmentation of the signal. That is, after the part of linear increase the signal will remain nearly constant. We

[3] The introduced syntax is in fact an anticipation of means we will systematically introduce later.

may revert to the heuristic from above in order to characterize the signal's shape but we are not able to address the sequential split up, which is determined by the validity of properties. To address the activation and deactivation of time scopes w.r.t. to the evaluation results of templates applied before, we need to refer to the begin of the phase a signal, a property is valid for, and the respecting end of the phase. During on-line analysis the *carry until* construct in *C*TTCN-3 already provides means to realize the detection of such phases. In this article we concentrate on templates and provide a declarative approach. That is, we introduce the function *start of* and *end of* to address the points in time that represent the beginning and the end of the valid phase of a predicate (i.e. template).

Listing 4. Start and End Marks

```
sof(linearSlope)
// start time, from where the property holds for the stream
eof(linearSlope)
// end time of this property, after which the property
// doesn't hold anymore
```

By use of such functions we can now define how properties depend and evolve:

Listing 5. Heuristic Example

```
template FloatStrm  constantSlope@t:={
 (differentiate(current) in [-0.1..0.1])}

template FloatStrm linear_and_ConstSlope@t:={
  @[0.0..10.0] := linearSlope@t ,
  @[eof(linearSlope)+1..eof(linearSlope)+5]:= constantSlope};
```

It is obvious, that the provided means could be extended to achieve a proper description of the expected velocity curve in our automotive example. The crucial ingredients could be identified as

- templates on streams, which mimick properties resp. predicates of signal outcomes,
- time scope restrictions of such templates, and
- start and end markings of the durations limits.

In this article we will mainly focus on the necessary extensions of templates and show how they naturally integrate these with *C*TTCN-3.

4 Evaluation of Signals

A signal property is a formal description of certain defined attributes of a signal. This subsumes the signal value at a certain point in time, the increase and decrease of a signal, or the occurrence of extrema. Table 1 shows a selection of *basic properties* adopted from [3].

Table 1. Signal Properties

Property Name	Characteristic	Description	Locality
Signal Value	value = *exp*	the signal value equals *exp*	local
	value in [*range exp*]	the signal value is in [*range exp*]	local
Value Change	no	a constant signal	frame-local
	increase	an increasing signal	frame-local
	decrease	an decreasing signal	frame-local
Extremal Value	minimum	the signal has a local minimum	frame-local
	maximum	the signal has a local maximum	frame-local
Signal Type	step-wise	a step function	global
	linear	a partially linear signal	global
	flat	a partially flat signal	global

While the actual signal value is a property that is completely *local* (i.e. it is quantifiable without the history of the signal) the other properties are only allocatable when the predecessor values are considered. The latter is named *frame-local*, when the history can be limited to a certain frame and *global* when not. Local properties are adequate for on-line analysis in any case, frame-local properties are adequate, but only in consideration of the frame size. Large frames may constrain the real time capabilities of the test environment. Global properties are normally not meaningful applicable for on-line analysis because they depend on the complete signal. In this article we confine ourselves to local and frame-local properties.

To address frequencies, monotony and the exact amount of decrease or increase a signal has, we introduce the notion of preprocessing functions. A preprocessing function obtain a signal as input and provides a transformed signal as output. In systems engineering multiple meaningful preprocessing functions (e.g. derivation, high-pass filters, fourier transformation etc.) exist. In this article we only consider the derivation of a signal.

Finally, we distinguish basic properties that address one and only one of the characteristic from Table 1 and complex properties that address a combination of characteristics. Complex properties are constructed by use of logical connectives (e.g. negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), and implication ($\rightarrow$)). Moreover we aim to address the temporal evolution of a signal along the time axis. Hence, the specification of temporal order and temporal dependencies are necessary.

5 The CTTCN-3 Solution

CTTCN-3 already provides a limited set of means to check the properties of a signal. Signals are represented as streams and predicates that check properties, which are specified by use of so called stream templates. The respective concepts are introduced in [12] in detail and summarized in 2.4. To systematically meet the requirements from above these means have to be enhanced.

- To provide transformations that are needed for pre-processing of streams (see Section 5.2) we introduce so called predefined transformation functions on streams and show how they integrate in the definition of stream templates.
- To model templates that address the properties of streams (as well as for their pre-processed derivation) we propose to enhance the syntax of template definitions. We introduce the notion of a *predicate expressions* to closely specify values and *time scopes* to restrict the application of a predicate in time. The original form of a stream template definition is short form of the one we introduce in this paper.
- For the logical and temporal combination of predicates we will introduce the construction of complex templates by use of logical connectives and the ability to trigger on the activation and deactivation of templates.

We start with the definition of predefined transformation function to realize the pre-processing of streams to be analyzed. Afterwards, we emphasize on the construction of complex templates (i.e. on time scopes, predicate expressions and the syntactical integration of transformation functions in the definition), and on the specification of temporal dependencies between templates.

5.1 Pre-processing Functions

We propose to specify the basic pre-processing functions as so called predefined functions. Predefined functions are defined as part of the core language [4] and are meant to be provided by the CTTCN-3 runtime environment.

The function $differentiate$ returns the first order derivative of a signal. In CTTCN-3 signals are represented as streams. The derivative $s'=differentiate(s)$ of a given source stream s is — similar to the source stream — defined by a structure $s' := (\Delta, \langle m'_k \rangle)$ where Δ represents the sample time, $\langle m'_k \rangle$ a sequence of values, and $s' \in (DSTRM)_T$. The sequence of values is defined as a left side derivative:

$$\langle m'_i \rangle := \{m'_1, m'_2, m'_3, ..., m'_k\}$$

$$and$$

$$m'_i = \begin{cases} 0 & when\ i = 1 \\ \frac{m_i - m_{(i-1)}}{\Delta} & when\ i > 1 \end{cases}$$

Please note that $m'_1 = 0$ due to the fact that m_i is not defined for $i = 0$. In CTTCN-3 the function $differentiate$ is specified with the following signature.

differentiate(numeric_stream_type value) numeric_stream_type

5.2 The General Setup of Stream Templates

While Section 2.4 provides a short overview over the notion of stream templates like they are already defined in [12], this section revises the initial design and provides a much more detailed insight in the underlying concepts. We start with the description of the general setup of a stream templates. On basis of this we will systematically introduce new features to enhance the expressiveness of stream templates to become a powerful instrument for the evaluation of system response in hybrid system testing.

Nevertheless, as mentioned before, a template generally consists of a *time scope* and a *predicate expression*. The time scope constitutes the validity of the predicate in respect to timing. The predicate expression constraints the value side of a stream.

Listing 6. Stream Templates

```
template FloatStrm t4@:={
  @[0.0..30.0]     :=  [0.0..55.0]};

template FloatStrm t5@t:={
  @[0.0..10.0)      :=  [2*t],
  @[10.0..18.0)     :=  [19.0..21.0],
  @[18.0..28.0)     :=  [20-(0.2*t)]};
```

The templates in Listing 6 consist of time scopes (at the left hand side e.g. `[0.0..30.0]`) and predicates expressed by values or value ranges (at the right hand side e.g. `[0.0..55.0]`). The predicates address the evolution of signal values only, which is obviously not enough to properly meet the requirements from Section 3.

Moreover a template may be defined segment-wise, that is, it may have different predicates for different segments of time, each defined by time scopes that precede the respective predicate (see $t4$ in Listing 6). A segment definition may override a precedent segment definition when the respective time scopes overlap.

Predicates: A predicate is dedicated to characterizes the values of a stream on different levels of abstraction. In [12] we confined the notion of predicates to be simple relational expressions that are expressed by values or value ranges (e.g. `[0.0..55.0]` or `[20-(0.2*t)]` in Listing 6). In this article we enhance the notion of predicates to be more efficient in terms of signal properties and introduce the notion of *predicate expressions* (i.e. more complex relational expressions, templates itself and the logical composition of templates and relational expressions).

Time Scopes: The application of a time scope restricts the evaluation of predicates in time. It consists of a start event $\phi_{start} \in \Sigma$, that activates the evaluation of a predicate and an end event $\phi_{end} \in \Sigma$ that deactivates the evaluation.

Moreover we consider a timing function $\tau_\phi : \Sigma \rightarrow \mathbb{R}$ that returns the point in time when an event has occurred. Please note, for events the accuracy of the timing function is restricted by sampling. Hence, events are considered time-consuming and lasts for exactly one step size.

Concerning time scopes we distinguish between templates having a *global time scope* and templates having a *local time scope*. A template has a global time scope, when the time scope specification is omitted or when $\tau_\Sigma(\phi_{start}) = 0.0$ and ϕ_{end} does not occur (e.g. `@[0.0..infinity]`). A time scope is identified local when the time scope defines a finite time period or when $\tau_\Sigma(\phi_{start}) > 0.0$ (e.g. `@[0.2..10.0)`). The syntactical structure to denote time scopes is similar to the structure of value ranges already defined in Continuous TTCN-3[4].

5.3 Evaluation of Templates

The evaluation of streams is carried out by the application of a template to a stream or a stream port. The result of a stream evaluation is affected by the time scope and the predicate of the applied template as well as by the application statement. While *match* initiates a global evaluation of a stream, the *sense* operator allows the sample-wise evaluation (see Section 2.4).

Concerning the calculation of match and sense results, we propose a *tolerant evaluation* of templates. A tolerant evaluation only checks the defined time scope of a template. Hence, a template with a local time scope is evaluated to boolean values *true* as long the analysis affects samples that are outside the template's time scope. Regarding samples that are covered by the time scope, the result of the evaluation is determined by predicate. That is, it yields *true* when the predicate matches and $false$ when the predicate does not match. Let us consider $r \in \mathbb{B}$ to be the result of a template application to a stream $s \in (DSTRM)_T$. Moreover we define $\chi_p : T \rightarrow \mathbb{B}$ the evaluation of a stream value at a certain point in time t by a predicate p. We define tolerant evaluation with:

$$r@t = \begin{cases} \chi_p(s@t) & when\ t \in [\tau_\Sigma(\phi_{start}), \tau_\Sigma(\phi_{end})] \\ true & else \end{cases}$$

Please note, the tolerant evaluation of templates is caused by the match or by the sense operation and is not a property of the templates itself. Thus, complex predicates that themselves may consists of multiple embedded templates are internally calculated in a strict mode, that is the undefined segments remain undefined. Tolerant mode is only used when the outermost template is applied to a stream.

[4] That is, the outer bound of a time scope is denoted by "[" or "(" and ")" or "]". With "(" we define a left side open time scope, that is the occurrence time of an *event* itself is not included in the time scope. With "[" we define a left side closed time scope that includes the occurrence time of *event*. The meaning of ")" or "]" is analogue.

5.4 Complex Predicates

Unlike the original version of CTTCN-3 the revised version provides predicate expressions. A complex predicate may consists of:

- relational expressions,
- templates or template references, and
- templates or template references connected by logical connectives.

We start with the presentation of how relational expression integrate in our conception of predicates and continue with an explanation on how already defined and named templates can be used and combined to form more complex predicate.

Relational Expressions: The original form of a stream template comprise a predicate that consists of a simple relational expression (i.e. a stream value equals a template value or is in a range of values). The subject of predication is naturally the (current) stream to which the template is applied (i.e. by means of a CTTCN-3 match or sense statement). If we intend to use pre-processing functions inside the definition of templates, the subject of predication may not be the current stream under analysis but one of its pre-processed derivation. To be able to distinguish between different subjects we propose to explicitly denote a subject and to provide means to relate a given subject to a value predicate (e.g. a value expressions or a range expression). Precisely because a subject is always defined as a transformation on the current stream, we need a symbol that represents the access to the current stream and that can be used inside a template definition.

- Hence, we introduce the keyword *current* to represent the stream the template is currently applied to, and
- we introduce the operators "*in*" and " = ", that relate a subject (e.g. current or pre-processed derivations of current) to concrete value expression.

The operator " = " relates a given subject (Listing 6 the subject *current*) to a concrete value or a stream definition. The operator "*in*" does the same for ranges.

The significance of the new statements become clear regarding the templates $t6$ and $t7$ from Listing 7. Both integrate the application of the pre-processing function $differentiate$. While template $t4$ or $t5$ in Listing 6 can only be used to check whether the values of a stream are in a certain range, template $t6$ can be used to check whether the values of the derivation of a stream are between 1.75 and 2.25 and template $t7$ can be used to do similar for the second order derivation of a stream[5].

[5] The brackets that clasp around the predicate expressions are optional and are only used to provide readability. Thus, the syntax of the original CTTCN-3 stream template constructs can be considered as a short form of the new constructs exemplified in Listing 7.

Listing 7. Application of Pre-processing Functions

```
template FloatStrm t6@t:={
  @[0.0..10.0] := differentiate(current) in [1.75..2.25]};

template FloatStrm t7@t:={
  @[0.0..300.0] :=
     differentiate(differentiate(current)) in [-1.0..1.0]};
```

The Composition of Predicates: Besides the specification of relational expressions we allow the construction of templates by means of already defined templates and by logical expressions, which itself consist of logical connectives (i.e. and, or, not, and implies), templates, and relational expressions. Listing 8 presents the composition of templates to ensure a certain increase of a signal and also checks the allowed value domain.

Listing 8. Defining Complex Templates by Applying Logical Connectives

```
template FloatStrm t8@t := {t5 and t6};
```

Please note, time scopes of enclosed template definitions remain valid. This holds for references to individual templates as well as for logical expressions. Nevertheless, we allow the restriction of enclosed time scopes by the application of a new time scope for the enclosing template. Lets take a simple example[6]. Template $t10$ in Listing 9 addresses the already time scoped template $t9$ and restricts the resulting time scope to `@[0.0..1.0]`. In contrast to that, the enlargement of time scopes is not possible, thus the absolute time scope of $t11$ is not `@[0.0..10.0]` but `@[0.0...6.0]`.

Listing 9. Reusing Time Scopes

```
template FloatStrm t9@t:= {@[0.0..6.0]:=0}

template FloatStrm t10@t:= {@[0.0..1.0]:= t9};

template FloatStrm t11@t:= {@[0.0..10.0]:= t9};

template FloatStrm t12@t:= {@[2.0..10.0]:= t9};
```

Please also note, that the time expressions that are defined inside the embedded template will be synchronized with the activation of the enclosing template. That is, the absolute time scope of template $t12$ lasts from 2.0 to 8.0.

Due to the fact that time scopes are preserved, templates and logical expressions, which contain time scoped templates, already provide the ability to specify the temporal evolution of complex properties. Nevertheless, we propose a carefully reuse of time scoped templates to not get lost in complexity.

[6] As from now we leave the automotive example, we will come back to it later.

5.5 Complex Time Scopes

So far, time scope definitions rely on time events that are local to the template definition only. To become more flexible in respect to the definition of time scopes, we introduce flexible time scopes and the definition of time scopes that are bounded by the result of template evaluations, that is the activation and deactivation of a templates may rely on the evaluation of other templates.

Flexible Time Scopes: Introducing flexible time scopes we provide the ability to formulate a more flexible beginning and ending of a time scope. The bounds of a flexible time scope are denoted as a range of possible time values. That is, for both, the beginning and ending of a scope, two events are denoted. The lower bound event denotes the first time point that is allowed for beginning or ending and the second event denotes the last possible time point that is allowed for beginning or ending. Listing 10 presents two examples.

Listing 10. Flexible Time Scopes

```
template FloatStrm t13@t:={
  @[[0..4]..8] := differentiate(current) = 0};

template FloatStrm t14@t:={
  @[[0..4]..[8..10]] := t9};
```

The template $t13$ is activated between `[0..4]` and deactivated at 8. Applied to a stream it evaluates to *true* when the predicate (`differentiate(current) = 0`) at least is valid between 4 and 8. Please note, the time scope of embedded templates affects the evaluation of their enclosing templates, when they obey flexible time scopes, in a certain manner. Template $t14$ is activated between 0 and 4. The enclosed template (see Listing 9) exhibit a time scope with a length of 6 seconds (`@[0..6]`). The absolute time scope of the enclosing template depends on its actual activation. When it is activated between 0 and 2 the length of the absolute time scope is completely determined by the enclosed template. When the outer time scope is activated later, the enclosing time scope weakens the condition for deactivation (by `[8..10]`) and hence the possible duration of stream evaluation.

Dependent Time Scopes: With the means provided in the last two subsection we are already capable to activate and deactivate templates by means of the time scope of other templates. Nevertheless, the activation is directly connected to time. This subsection provides means to relates the definition of a time scope to the validation of templates. For this purpose we introduce the functions:

- *start of* or short `sof(template)` that fires an event when a template is evaluated to true for the first time and
- *end of* or short `eof(template)` that fires an event when a template was already true and either is evaluated to false or is deactivated.

Hence, we can define the activation of a template in dependence on other templates. Listing 11 shows such an example. Template $t17$ is activated when $t15$ switches becomes invalid and is deactivated when $t16$ becomes invalid. To determine the sof() and eof() events the strict evaluation of the templates $t15$ and $t16$ is necessary.

Listing 11. Temporal Order

```
template FloatStrm t15@t:=0;
template FloatStrm t16@t:=10;

template FloatStrm t17@t{
  @[eof(t15)..sof(t16)]:= sin(t)};
```

Moreover, we can combine the notion of flexible time scopes with the notion of dependent time scopes and pick up the example from Listing 5 and provide a more flexible version in Listing 12. Template $t20$ first checks for the phase with linear slope and expects the constant phase to start at least one second and at most 2 seconds after the first phase has ended.

Listing 12. Heuristic Example II

```
template FloatStrm  constantSlope@t:={
 (differentiate(current) in [-0.1..0.1])}

template FloatStrm t20@t:={
  @[0.0..10.0] := linearSlope@t ,
  @[[(eof(linearSlope)+1.0)..(eof(linearSlope)+2.0)]..
     (eof(linearSlope)+6.0)] := constantSlope@t};
```

6 Summary and Outlook

In this article we have discussed the application of predicates to characterize the properties of a signal. By means of a simple scenario from the automotive domain, we have illustrated the concepts that are needed to properly define such predicates. Moreover, we provided a list of properties to be checked and examined their adequacy for the on-line analysis of system reaction, that is the analysis during test runtime. The second part of this article provides a simple integration of the introduced concepts to CTTCN-3. We could show how signal predicates can be realized by means of CTTCN-3 stream templates. Moreover we have provided the necessary syntactical add-ons to denote complex templates, that is templates that are build upon other templates, and to model the temporal dependencies between template invocation.

More effective means like the introduction temporal logic operators (e.g. globally, exists, until, release etc.) and the specification of dependencies of templates that relates the properties of different signals to each other will be subject of further research.

References

[1] Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.-H.: Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems. In: Hybrid Systems, pp. 209–229 (1992)

[2] Broy, M.: Refinement of Time. In: Bertran, M., Rus, T. (eds.) Transformation-Based Reactive System Development, ARTS 1997, TCS, pp. 44–63 (1997)

[3] Gips, C., Hans-Werner, W.: Proposal for an Automatic Evaluation of ECU Output. In: Dagstuhl-Workshop MBEES 2007. Modellbasierte Entwicklung eingebetteter Systeme, Braunschweig, GER (2007)

[4] ETSI: ES 201 873-1 V3.2.1: Methods for Testing and Specification (MTS). The Testing and Test Control Notation Version 3, Part 1: TTCN-3 Core Language. Sophia Antipolis, France (February 2007)

[5] ETSI: ES 201 873-4 V3.2.1: Methods for Testing and Specification (MTS). The Testing and Test Control Notation Version 3, Part 4: TTCN-3 Operational Semantics. Sophia Antipolis, France (February 2007)

[6] ETSI: ES 201 873-5 V3.2.1: Methods for Testing and Specification (MTS). The Testing and Test Control Notation Version 3, Part 5: TTCN-3 Runtime Interfaces. Sophia Antipolis, France (February 2007)

[7] Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: HYTECH: A Model Checker for Hybrid Systems. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 460–463. Springer, Heidelberg (1997)

[8] Joshi, A., Heimdahl, M.P.E.: Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier. In: Winther, R., Gran, B.A., Dahll, G. (eds.) SAFECOMP Bd. 2005. LNCS, vol. 3688, pp. 122–135. Springer, Heidelberg (2005)

[9] Lehmann, E.: Time Partition Testing Systematischer Test des kontinuierlichen Verhaltens von eingebetteten Systemen. Berlin, TU-Berlin, Diss. (2004)

[10] Lim, J.S., Oppenheim, A.V.: Advanced topics in signal processing. Prentice-Hall, Inc., Upper Saddle River (1987)

[11] Schieferdecker, I., Bringmann, E., Grossmann, J.: Continuous TTCN-3: testing of embedded control systems. In: SEAS 2006. Proceedings of the 2006 international workshop on Software engineering for automotive systems, pp. 29–36. ACM Press, New York (2006)

[12] Schieferdecker, I., Grossmann, J.: Testing Hybrid Control Systems with TTCN-3, An Overview on Continuous TTCN-3. In: TTCN-3 STTT (2008)

[13] Sims, S., DuVarney, D.C.: Experience report: the reactis validation tool. In: Hinze, R., Ramsey, N. (eds.) ICFP, pp. 137–140. ACM, New York (2007)

[14] Zander-Nowicka, J., Schieferdecker, I., Pérez, A.M.: Automotive Validation Functions for On-line Test Evaluation of Hybrid Real-time Systems. In: IEEE 41st Anniversary of the Systems Readiness Technology Conference (AutoTestCon 2006) (2006); IEEE Catalog Number: 06CH37750C, ISBN 1-4244-0052-X

$\mathcal{THOTL}$: A Timed Extension of $\mathcal{HOTL}$[⋆]

Mercedes G. Merayo, Manuel Núñez, and Ismael Rodríguez

Dept. Sistemas Informáticos y Computación
Universidad Complutense de Madrid, 28040 Madrid, Spain
mgmerayo@fdi.ucm.es, mn@sip.ucm.es, isrodrig@sip.ucm.es

Abstract. $\mathcal{THOTL}$ represents a conservative extension of $\mathcal{HOTL}$ (Hypotheses and Observations Testing Logic) to deal with systems where time plays a fundamental role. We adapt some of the $\mathcal{HOTL}$ rules to cope with the new framework. In addition, we introduce several specific hypotheses and rules to appropriately express time assumptions. We provide a correctness result of $\mathcal{THOTL}$ with respect to a general notion of timed conformance.

1 Introduction

In order to determine the *correctness* of an implementation with respect to a given specification we can use a notion of *conformance*: An implementation *conforms* to a specification if the former shows a behavior *similar* to that of the latter. In order to check this property we may use formal testing techniques to extract tests from the specification. Each test represents a desirable behavior that the implementation under test (in the following IUT) must fulfill. In fact, the existence of such a formal framework facilitates the automation of the testing process. In order to limit the (possibly infinite) time devoted to testing, testers add some reasonable assumptions about the structure of the IUT. For example, the tester can assume that the implementation can be represented by means of a deterministic finite state machine, that it has at most n states, etc. In this line, a wide range of testing methodologies have been proposed which, for a specific set of initial hypotheses, guarantee that a test suite extracted from the specification is correct and complete to check the conformance of the IUT (e.g. [4,15,19,6]). However, a framework of hypotheses established in advance is very strict and limits the applicability of a specific testing methodology. For example, it could be desirable that, in a concrete environment, the tester make complex assumptions such as "*non-deterministic states of the implementation cannot show outputs that the machine did not show once the state has been tested 100 times.*" In a different scenario the tester could not believe this assumption but think that "*if she observes two sequences of length 200 and all their inputs and outputs coincide then they actually traverse the same IUT states.*" Let us remark that these

⋆ This research was partially supported by the Spanish MEC project WEST/FAST TIN2006-15578-C02-01 and the Marie Curie project MRTN-CT-2003-505121/ TAROT.

are *hypotheses* that the tester is assuming. Thus, she might be wrong and reach a wrong conclusion. However, this is similar to the case when the tester assumes that the implementation is deterministic or that it has at most n states and, in reality, this is not the case.

The logical framework $\mathcal{HOTL}$ [20,21] (*Hypotheses and Observations Testing Logic*) was introduced to cope with the rigidity of other testing frameworks. $\mathcal{HOTL}$ aims to assess whether a given set of observations implies the correctness of the IUT under the assumption of a given set of hypotheses. The methodology consists of two phases. The first phase consists in the classical application of tests to the IUT. By using any of the available methods in the literature, a test suite will be derived from the specification. If the application of this test suite finds an unexpected result then the testing process stops: The IUT is not conforming. However, if such a wrong behavior is not detected then the tester cannot be sure that the IUT is correct. In this case, the second phase begins, that is, the tester applies $\mathcal{HOTL}$ to infer whether passing these tests *implies* that the IUT is correct if a given set of hypotheses is assumed. If it does then the IUT is assumed to be correct; otherwise, the tester may be interested in either applying more tests or in assuming more hypotheses (in the latter case, on the cost of *feasibility*) and then applying the logic again until the correctness of the IUT is effectively granted.

$\mathcal{HOTL}$ provides two types of hypotheses: Concerning specific parts (states) of the IUT and concerning the whole IUT. In order to unambiguously denote the states regarded by the former, they will be attached to the corresponding observations that reached these states. For example, if the IUT was showing the sequence of outputs $o_1, o_2, \ldots, o_n$ as answer to the sequence of inputs $i_1, i_2, \ldots, i_n$, the tester may think that the state reached after performing i_1/o_1 is deterministic or that the state reached after performing the sequence $i_1/o_1, i_2/o_2$ is the same as the one reached after performing the whole sequence $i_1/o_1, i_2/o_2, \ldots, i_n/o_n$. In addition to using hypotheses associated to observations, the tester can also consider global hypotheses that concern the whole IUT. These are assumptions such as the ones that we mentioned before: Assuming that the IUT is deterministic, that it has at most n states, that it has a unique initial state, etc. In order to denote the assumption of this kind of hypotheses, specific logic predicates will be used.

Let us remark that even though we work with rules and properties, $\mathcal{HOTL}$, and its timed extension presented in this paper, is not related to *model checking* [8] since we do not *check* the validity of properties: We assume that they hold and we infer results about the conformity of the IUT by using this assumption. In the same way, this work is not related to some recent work on passive testing where the validity of a set of properties (expressed by means of *invariants*) is checked by passively observing the execution of the system (e.g. [3,14]).

The main goal of this paper is to extend $\mathcal{HOTL}$ to deal with timed systems. Even though there exist several proposals to test timed systems (for example, [22,11,7,10,9,12,5,13,2,17]), these proposals suffer from the same *rigidity* previously commented. The first decision to define the new framework, that we call $\mathcal{THOTL}$, is to consider a formal language to represent timed systems. The natural candidate would be to consider timed automata [1]. However, since

$\mathcal{HOTL}$ is oriented to deal with a language with a strict alternation between inputs and outputs, we decided to consider a timed extension of finite state machines in order to reuse, as much as possible, the definition of the predicates and rules. Intuitively, transitions in finite state machines indicate that if the machine is in a state s and receives an input i then it will produce and output o and it will change its state to s'. An appropriate notation for such a transition could be $s \xrightarrow{i/o} s'$. If we consider a timed extension of finite state machines, transitions as $s \xrightarrow{i/o}_d s'$ indicate that the time between receiving the input i and returning the output o is given by d, where d belongs to a certain time domain. Usually, d is considered to be a *simple* value (e.g. a non-negative real number). However, we would like to use a more sophisticated time domain. Our first choice was to consider a *stochastic* version of finite state machines [18] where a transition $s \xrightarrow{i/o}_\xi s'$ indicates that the time elapsed between i and o is equal to t with probability $F_\xi(t)$, where F_ξ is the probability distribution function associated with the random variable ξ. However, this choice strongly complicated the definition of some rules and the notion of observation. In fact, the added complication was such that it would deviate the attention from the main goal of the paper: Introduce time in $\mathcal{HOTL}$. Thus, we decided to choose a simpler approach but richer than singles values: Time intervals. Thus, a transition $s \xrightarrow{i/o}_{[d_1,d_2]} s'$ indicates that the time elapsed from the moment i is offered until the moment o is produced is at most d_2 time units and at least d_1 time units. Actually, time intervals represent a simplification of random variables where the different *weight* given to each possible time value is not quantified.

Once the language and a notion of timed conformance were fixed, we had to work on how to *adapt* $\mathcal{HOTL}$ to the new setting. Initially, we thought that this task would be straightforward, consisting in modifying some of the rules so that time values were appropriately added and dealt with. However, once we started to work in the proposal, we realized that to develop the new framework would be much more involved than a simple rewriting of the non-timed framework. First, we had to adapt the notion of *observation* to take into account not only assumptions about the possible time interval governing transitions but also to record the observed time values: Since we consider time intervals, different observations of the very same transition can produce different time values. Next, we had to modify the existing rules. The addition of time complicated the rules linked to the *accounting* of observations. Finally, we introduced new hypotheses to express specific temporal constraints.

This paper represents an extension of [16] where we presented a preliminary timed extension of $\mathcal{HOTL}$. In particular, [16] concentrated on adapting existing $\mathcal{HOTL}$ hypotheses and rules to cope with a timed model. However, new hypotheses and rules to deal with specific time issues, what strongly complicates the theoretical development, were not included in [16].

The rest of the paper is organized as follows. In Section 2 we introduce our extension of finite state machines to model timed systems and define two implementation relations. In Section 3, we define the basics of the new logical framework. In Section 4, we introduce new hypotheses and provide a correctness result. Finally, in Section 5 we present our conclusions and some directions for further research.

2 A Timed Extension of `FSMs`

In this section we present our timed extension of the classical finite state machine model. The main difference with respect to usual `FSMs` consists in the addition of *time* to indicate the lapse between offering an input and receiving an output. As we already indicated in the introduction, time intervals will be used to express time constraints associated with the performance of actions. First we need to introduce notation, related to time intervals and multisets, that we will use during the rest of the paper.

Definition 1. We say that $\hat{a} = [a_1, a_2]$ is a *time interval* if $a_1 \in \mathbb{R}_+$, $a_2 \in \mathbb{R}_+ \cup \{\infty\}$, and $a_1 \leq a_2$. We assume that for all $r \in \mathbb{R}_+$ we have $r < \infty$ and $r + \infty = \infty$. We consider that $\mathcal{I}_{\mathbb{R}_+}$ denotes the set of time intervals. Let $\hat{a} = [a_1, a_2]$ and $\hat{b} = [b_1, b_2]$ be time intervals. We write $\hat{a} \subseteq \hat{b}$ if we have both $b_1 \leq a_1$ and $a_2 \leq b_2$. In addition, $\hat{a} + \hat{b}$ denotes the interval $[a_1 + b_1, a_2 + b_2]$ and $\pi_i(\hat{a})$, for $i \in \{1, 2\}$, denotes the value a_i.

We will use the delimiters $\{\!|$ and $|\!\}$ to denote multisets. We denote by $\wp(\mathbb{R}^+)$ the multisets of elements belonging to $\mathbb{R}^+$. Given $H \in \wp(\mathbb{R}^+)$, for all $r \in \mathbb{R}^+$ we have that $H(r)$ denotes the *multiplicity* of r in H. Given two multisets $H_1, H_2 \in \wp(\mathbb{R}^+)$, $H_1 \uplus H_2$ denotes the *union* of H_1 and H_2, and it is formally defined as $(H_1 \uplus H_2)(r) = H_1(r) + H_2(r)$ for all $r \in \mathbb{R}^+$. □

Let us note that in the case of $[t_1, \infty]$ and $[0, \infty]$ we are abusing the notation since these intervals are in fact half-closed intervals, that is, they represent the intervals $[t_1, \infty)$ and $[0, \infty)$, respectively.

Definition 2. A *Timed Finite State Machine*, in the following `TFSM`, is a tuple $M = (\mathcal{S}, \texttt{inputs}, \texttt{outputs}, \mathcal{I}, \mathcal{T})$ where $\mathcal{S}$ is a finite set of states, `inputs` is the set of input actions, `outputs` is the set of output actions, $\mathcal{T}$ is the set of transitions, and $\mathcal{I}$ is the set of initial states.

A transition belonging to $\mathcal{T}$ is a tuple $(s, s', i, o, \hat{d})$ where $s, s' \in \mathcal{S}$ are the initial and final states of the transition, respectively, $i \in \texttt{inputs}$ and $o \in \texttt{outputs}$ are the input and output actions, respectively, and $\hat{d} \in \mathcal{I}_{\mathbb{R}_+}$ denotes the possible time values the transition needs to be completed. We usually denote transitions by $s \xrightarrow{i/o}_{\hat{d}} s'$.

We say that $(s, s', (i_1/o_1, \ldots, i_r/o_r), \hat{d})$ is a *timed trace*, or simply *trace*, of M if there exist $(s, s_1, i_1, o_1, \hat{d}_1)$,…, $(s_{r-1}, s', i_r, o_r, \hat{d}_r) \in \mathcal{T}$, such that $\hat{d} = \sum \hat{d}_i$.

We say that $((i_1/o_1, \ldots, i_r/o_r), \hat{d})$ is a *timed evolution* of M if there exists $s_{in} \in \mathcal{I}$ such that $(s_{in}, s', (i_1/o_1, \ldots, i_r/o_r), \hat{d})$ is a trace of M. We denote by $\texttt{TEvol}(M)$ the set of timed evolutions of M. In addition, we say that $(i_1/o_1, \ldots, i_r/o_r)$ is a *non-timed evolution*, or simply *evolution*, of M and we denote by $\texttt{NTEvol}(M)$ the set of non-timed evolutions of M.

Let us consider $s, s' \in \mathcal{S}$. We say that the state s' is *reachable* from s, denoted by $\texttt{isReachable}(M, s, s')$, if either there exist $u, i, o, \hat{d}$ such that $s \xrightarrow{i/o}_{\hat{d}} u \in \mathcal{T}$ and $\texttt{isReachable}(M, u, s')$ holds, or $s = s'$. The set $\texttt{reachableStates}(M, s)$ contains all $s' \in \mathcal{S}$ such that $\texttt{isReachable}(M, s, s')$.

Let $s \in \mathcal{S}$ and $i \in \texttt{inputs}$. The set $\texttt{outs}(M, s, i)$ contains the outputs that can be produced in s in response to i, that is, $\{o \mid \exists\, s', \hat{d} : s \xrightarrow{i/o}_{\hat{d}} s' \in \mathcal{T}\}$.

Finally, we say that $s \in \mathcal{S}$ is *deterministic*, denoted by $\texttt{isDet}(M, s)$, if there do not exist $s \xrightarrow{i/o'}_{\hat{d}} s', s \xrightarrow{i/o''}_{\hat{d}'} s'' \in \mathcal{T}$ such that $o' \neq o''$ or $s' \neq s''$. □

Intuitively, a transition $(s, s', i, o, \hat{d})$ indicates that if the machine is in state s and receives the input i then, after a time belonging to the interval $\hat{d}$, the machine emits the output o and moves to s'. Traces are sequences of transitions. The time associated with the trace is computed by adding the intervals associated with each of the transitions conforming the trace. We allow machines to be non-deterministic. We assume that both implementations and specifications can be represented by appropriate TFSMs and we consider that IUTs are *input-enabled.* During the rest of the paper we will assume that a generic specification is given by $spec = (\mathcal{S}_{spec}, \texttt{inputs}_{spec}, \texttt{outputs}_{spec}, \mathcal{I}_{spec}, \mathcal{T}_{spec})$.

Next we introduce our first timed implementation relation. In addition to the *untimed* conformance of the implementation, we require some time conditions to hold. Intuitively, an IUT is conforming if it does not *invent* behaviors for those traces that can be executed by the specification and time values are as expected.

Definition 3. Let I and S be TFSMs. We say that I *conforms to* S, denoted by $I\ \texttt{conf}\ S$, if for all $\rho_1 = (i_1/o_1, \ldots, i_{n-1}/o_{n-1}, i_n/o_n) \in \texttt{NTEvol}(S)$, with $n \geq 1$, we have $\rho_2 = (i_1/o_1, \ldots, i_{n-1}/o_{n-1}, i_n/o'_n) \in \texttt{NTEvol}(I)$ implies $\rho_2 \in \texttt{NTEvol}(S)$. We say that I *conforms in time* to S, denoted by $I\ \texttt{conf}_{int}\ S$, if $I\ \texttt{conf}\ S$ and for all $e \in \texttt{NTEvol}(I) \cap \texttt{NTEvol}(S)$ and $\hat{d} \in \mathcal{I}_{\mathbb{R}_+}$, we have that $(e, \hat{d}) \in \texttt{TEvol}(I)$ implies $(e, \hat{d}) \in \texttt{TEvol}(S)$. □

Even though this is a very reasonable notion of conformance, a black-box testing framework disallows us to check whether the corresponding time intervals coincide. The problem is that we cannot compare in a direct way timed requirements of the *real* implementation with those established in the specification. In fact, we can *see* the time interval defining a given transition in the specification, but we cannot do the same with the corresponding transition of the implementation, since we do not have access to it. Thus, we have to give a more *realistic* implementation relation based on a finite set of observations. We will present an implementation relation being less *accurate* but *checkable.* Specifically, we will check that the observed time values (from the implementation) belong to the time interval indicated in the specification.

Definition 4. Let I be a TFSM. We say that $((i_1/o_1, \ldots, i_n/o_n), t)$ is an *observed timed execution* of I, or simply *timed execution*, if the observation of I shows that the sequence $(i_1/o_1, \ldots, i_n/o_n)$ is performed in time t.

Let $\Phi = \{e_1, \ldots, e_m\}$ be a set of input/output sequences and let us consider a multiset of timed executions $H = \{\!|(e'_1, t_1), \ldots, (e'_n, t_n)|\!\}$. We say that $\texttt{Sampling}_{(H,\Phi)} : \Phi \longrightarrow \wp(\mathbb{R}^+)$ is a *sampling application* of H for Φ if for all $e \in \Phi$ we have $\texttt{Sampling}_{(H,\Phi)}(e) = \{\!| t \mid (e,t) \in H |\!\}$. □

Timed executions are input/output sequences together with the time that it took to perform the sequence. In a certain sense, timed executions can be seen as *instances* of the evolutions that the implementation can perform. Regarding the definition of sampling applications, we just associate with each evolution the multiset of observed execution time values.

Definition 5. Let I and S be two TFSMs, H be a multiset of timed executions of I, and $\Phi = \{e \mid \exists\, t : (e,t) \in H\} \cap \texttt{NTEvol}(S)$. For all non-timed evolution $e \in \Phi$ we define the *sample interval* of e in H as

$$\widehat{S}_{(H,e)} = [\texttt{min}(\texttt{Sampling}_{(H,\Phi)}(e)), \texttt{max}(\texttt{Sampling}_{(H,\Phi)}(e))]$$

We say that I $H-$*timely conforms* to S, denoted by $I\ \texttt{conf}^{H}_{int}\ S$, if $I\ \texttt{conf}\ S$ and for all $e \in \Phi$ and $\hat{d} \in \mathcal{I}_{\mathbb{R}_+}$ we have that $(e, \hat{d}) \in \texttt{TEvol}(S)$ implies $\widehat{S}_{(H,e)} \subseteq \hat{d}$. □

3 $\mathcal{THOTL}$: An Extension of $\mathcal{HOTL}$ with Time

In this section we present the new formalism $\mathcal{THOTL}$. This framework represents an extension and adaption of $\mathcal{HOTL}$ to cope with systems where time plays a fundamental role. While some of the rules remain the same (the rules dealing with the internal *functional* structure of the implementation), $\mathcal{THOTL}$ constitutes a complete *new* formalism. Next, we briefly describe the main contributions with respect to $\mathcal{HOTL}$. First, we have to redefine most components of the logic to consider temporal aspects. Observations will include the time values that the IUT takes to emit an output since an input is received. Additionally, the model will be extended to take into account the different time values appearing in the observations for each input/output outgoing from a state. We will add new hypotheses to allow the tester to represent assumptions about temporal behaviors concerning both specific states and the whole IUT. Finally, we will modify the deduction rules as well as include new rules to add the new hypotheses to the models. During the rest of the paper Obs denotes the multiset of observations collected during the preliminary interaction with the IUT while Hyp denotes the set of *hypotheses* the tester has assumed. In this latter set, we will not consider the hypotheses that are implicitly introduced by means of observations.

3.1 Temporal Observations

In our framework we consider that temporal observations follow the format $ob = (a_1, i_1/o_1/t_1, a_2, \ldots, a_n, i_n/o_n/t_n, a_{n+1}) \in \texttt{Obs}$. This expression denotes that when the sequence of inputs $i_1, \ldots, i_n$ was proposed from an initial state of the implementation, the sequence $o_1, \ldots, o_n$ was obtained as response in

$t_1, \ldots, t_n$ time units, respectively. Moreover, for all $1 \leq j \leq n+1$, a_j represents a set of *special attributes* concerning the state of the implementation reached after performing $i_1/o_1, \ldots, i_{j-1}/o_{j-1}$ in *this* observation. Attributes denote our assumptions about this state. The attributes belonging to the set a_j are of the form $\texttt{imp}(q)$ or $\texttt{det}$, where $\texttt{imp}(q)$ denotes that the implementation state reached after $i_1/o_1, \ldots, i_{j-1}/o_{j-1}$ is associated to a *state identifier name* q and $\texttt{det}$ denotes that the implementation state reached after $i_1/o_1, \ldots, i_{j-1}/o_{j-1}$ in this observation is deterministic. State identifier names are used to match equal states: If two states are associated with the same state identifier name then they represent the *same* state of the implementation. The set of all state identifier names will be denoted by $\mathcal{Q}$. In addition, attributes belonging to a_{n+1} can also be of the form $\texttt{spec}(s)$ denoting that the implementation state reached after $i_1/o_1, \ldots, i_n/o_n$ is such that the subgraph that can be reached from it is isomorphic to the subgraph that can be reached from the state s of the specification. Thus, the behavior of the implementation from that point on is known and there is no need to check its correctness. In addition to the previous attributes, already defined in $\mathcal{HOTL}$, temporal observations may include a new type of attribute that represents our assumption about the time interval in which a transition can be performed. For all $1 < j \leq n$, the attributes in the set a_j can be also of the form $\texttt{int}(\hat{d})$, with $\hat{d} \in \mathcal{I}_{\mathbb{R}_+}$. Such an attribute denotes that the time that the implementation takes from the state reached after performing $i_1/o_1, \ldots, i_{j-1}/o_{j-1}$, to emit the output o_j after it received the input i_j belongs to the interval $\hat{d}$. We assume that this attribute cannot appear in the set a_1, since the implementation is in an initial state, and no actions have taken place yet.

3.2 Model Predicates

Temporal observations will allow to create *model predicates* that denote our knowledge about the implementation. A model predicate is denoted by $\texttt{model}\,(m)$, where $m = (\mathcal{S}, \mathcal{T}, \mathcal{I}, \mathcal{A}, \mathcal{E}, \mathcal{D}, \mathcal{O})$. As in $\mathcal{HOTL}$, $\mathcal{S}$ is the set of states appearing in the model, $\mathcal{T}$ is the set of transitions appearing in the graph of the model, $\mathcal{E}$ is the set of equalities relating states belonging to $\mathcal{S}$, $\mathcal{D}$ is the set of deterministic states, and $\mathcal{O}$ is the set of observations we have used so far for the construction of this model. In addition, $\mathcal{I}$ is the set of states that are initial in the model. Two additional symbols may appear in $\mathcal{I}$. The first special symbol, α, denotes that any state in $\mathcal{S}$ could eventually be initial. The second symbol, β, denotes that not only states belonging to $\mathcal{S}$ but also other states not explicitly represented in $\mathcal{S}$ could be initial. Finally, $\mathcal{A}$ is the set of *accounting registers* of the model. An accounting register is a tuple $(s, i, outs, f, \delta, n)$ denoting that in state $s \in \mathcal{S}$ the input i has been offered n times and we have obtained the outputs belonging to the set $outs$. Besides, for each transition departing from state s and labelled with input i, the function $f : \mathcal{T} \longrightarrow \mathbb{N}$ returns the number of times the transition has been observed. If, due to the hypotheses that we consider, we infer that the number of times we observed an input is high enough to believe that the implementation cannot react to that input either with an output that was not produced before or leading to a state that was not taken before, then the value n is set to $\top$. In this case, we say that the behavior at state s for the input i is *closed*. The only change we need to introduce with respect to $\mathcal{HOTL}$ is that each register includes a function $\delta : \mathcal{T} \longrightarrow \mathcal{I}_{\mathbb{R}_+} \times \wp(\mathbb{R}^+)$ that computes for each transition departing from state s with input i and output $o \in outs$ the time interval, according to our knowledge up to now, in which the transition could be performed and the set of time values the implementation took to perform the transition. If no assumptions about the interval are made by means of temporal observations, it will be set to $[0, \infty]$. In the case of transitions not fulfilling the conditions about s, i, and $outs$, an arbitrary value is returned. Let us remark that as new hypotheses and temporal observations are included in the model, the intervals will be reduced.

3.3 Basic $\mathcal{THOTL}$ Rules

First, we will include a new rule denoting how a model can be constructed from a temporal observation.

$$(tobser)\frac{ob = (a_1, i_1/o_1/t_1, a_2, \ldots, a_n, i_n/o_n/t_n, a_{n+1}) \in \mathtt{Obs} \ \wedge \ s_1, \ldots, s_{n+1} \text{ fresh states}}{\begin{array}{ll} & \{s_1, \ldots, s_{n+1}\} \cup \mathcal{S}', \\ & \{s_1 \overset{i_1/o_1}{\longrightarrow} s_2, \ldots, s_n \overset{i_n/o_n}{\longrightarrow} s_{n+1}\} \cup \mathcal{T}', \{s_1, \beta\}, \\ \mathtt{model} & \{(s_j, i_j, \{o_j\}, f_{s_j}\delta_{s_j}, 1) \mid 1 \leq j \leq n\} \cup \mathcal{A}', \\ & \{s_j \ \mathtt{is} \ q_j | 1 \leq j \leq n+1 \ \wedge \ \mathtt{imp}(q_j) \in a_j\}, \\ & \{s_j \mid 1 \leq j \leq n+1 \ \wedge \ \mathtt{det} \in a_j\} \cup \mathcal{D}', \{ob\} \end{array}}$$

where $f_{s_j}(tr)$ is equal to 1 if $tr = s_j \overset{i_j/o_j}{\longrightarrow} s_{j+1}$ and equal to 0 otherwise; and

$$\delta_{s_j}(tr) = \begin{array}{ll} (\hat{d}, \{\!\{t_j\}\!\}) & \mathtt{if} \ tr = s_j \overset{i_j/o_j}{\longrightarrow} s_{j+1} \ \wedge \ \mathtt{int}(\hat{d}) \in a_{j+1} \\ ([0,\infty], \{\!\{t_j\}\!\}) & \mathtt{if} \ tr = s_j \overset{i_j/o_j}{\longrightarrow} s_{j+1} \ \wedge \ \mathtt{int}(\hat{d}) \notin a_{j+1} \\ ([0,\infty], \emptyset) & \text{otherwise} \end{array}$$

The sets of states, transitions, accounting registers, and deterministic states will be extended with some extra elements, taken from the specification, if the tester assumes that the last state of the observation is isomorphic to a state of the specification (i.e., $\mathtt{spec}(s)$, for some $s \in \mathcal{S}_{spec}$). The new states and transitions $\mathcal{S}'$ and $\mathcal{T}'$, respectively, will copy the structure existing among the states that can be reached from s in the specification. The new accounting, $\mathcal{A}'$, will denote that the knowledge concerning the new states is *closed* for all inputs, that is, the only transitions departing from these states are those we copy from the specification and no other transitions will be added in the future. Additionally, accounting registers will reflect the time intervals associated to the transitions that are copied from the specification. Finally, those model states that correspond to deterministic specification states will be included in the set $\mathcal{D}'$ of deterministic states of the model. The formal definition of $\mathcal{S}'$, $\mathcal{T}'$, $\mathcal{A}'$, and $\mathcal{D}'$ follows. If there does not exist s' such that $\mathtt{spec}(s') \in a_{n+1}$ then $(\mathcal{S}', \mathcal{T}', \mathcal{A}', \mathcal{D}') = (\emptyset, \emptyset, \emptyset, \emptyset)$.

Otherwise, that is, if $\texttt{spec}(s) \in a_{n+1}$ for some $s \in \mathcal{S}_{spec}$, let us consider the following set of states:

$$U = \{u_j \mid u_j \text{ is a fresh state } \wedge\ 1 \leq j < |\texttt{reachableStates}(spec, s)|\}$$

and a bijective function $g : \texttt{reachableStates}(spec, s) \longrightarrow U \cup \{s_{n+1}\}$ such that $g(s) = s_{n+1}$. Then, $(\mathcal{S}', \mathcal{T}', \mathcal{A}', \mathcal{D}')$ is equal to

$$\left(\begin{array}{c} U, \\ \{g(s') \xrightarrow{i/o} g(s'') \mid \exists\, \hat{d} : s' \xrightarrow{i/o}_{\hat{d}} s'' \in \mathcal{T}_{spec} \ \wedge\ \texttt{isReachable}(spec, s, s')\}, \\ \left\{ \left(\begin{array}{c} u, i, \\ \texttt{outs}(spec, g^{-1}(u), i), \\ f_u^i, \delta_u^i, \top \end{array} \right) \left| \begin{array}{l} u \in U \cup \{s_{n+1}\} \ \wedge\ i \in \texttt{inputs}_{spec} \wedge \\ \exists u' \in U, o \in \texttt{outputs}_{spec} :\ u \xrightarrow{i/o} u' \in \mathcal{T}' \end{array} \right. \right\}, \\ \{g(s') | \texttt{isReachable}(spec, s, s') \ \wedge\ \texttt{isDet}(spec, s')\} \end{array}\right)$$

where $f_u^i(tr)$ is equal to 1 if there exist o', u' such that $tr = u \xrightarrow{i/o'} u' \in \mathcal{T}'$ and equal to 0 otherwise; and

$$\delta_u^i(tr) = \begin{array}{ll} (\hat{d}, \{\!|a|\ a \in \hat{d}|\!\}) & \texttt{if}\ \exists\, o', u' : tr = u \xrightarrow{i/o'} u' \in \mathcal{T}' \ \wedge \\ & \qquad g^{-1}(u) \xrightarrow{i/o'}_{\hat{d}} g^{-1}(u') \in \mathcal{T}_{spec} \\ ([0, \infty], \emptyset) & \quad \text{otherwise} \end{array}$$

We can join different models, created from different observations, into a single model by means of the *(fusion)* rule. The components of the new model are the union of the components of each model.

$$(\textit{fusion})\frac{\begin{array}{l}\texttt{model}\,(\mathcal{S}_1, \mathcal{T}_1, \mathcal{I}_1, \mathcal{A}_1, \mathcal{E}_1, \mathcal{D}_1, \mathcal{O}_1) \ \wedge \\ \texttt{model}\,(\mathcal{S}_2, \mathcal{T}_2, \mathcal{I}_2, \mathcal{A}_2, \mathcal{E}_2, \mathcal{D}_2, \mathcal{O}_2) \ \wedge\ \mathcal{O}_1 \cap \mathcal{O}_2 = \emptyset\end{array}}{\texttt{model}\ \ \mathcal{S}_1 \cup \mathcal{S}_2, \mathcal{T}_1 \cup \mathcal{T}_2, \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{E}_1 \cup \mathcal{E}_2, \mathcal{D}_1 \cup \mathcal{D}_2, \mathcal{O}_1 \cup \mathcal{O}_2}$$

The iterative application of this rule allows us to join different models created from different temporal observations into a single model.

At this point, the inclusion of those hypotheses that are not covered by observations will begin. During this new phase, we will usually need several models to represent all the FSMs that are compatible with a set of observations and hypotheses. Some of the rules use the modelElim function. If we discover that a state of the model coincides with another one, we will eliminate one of the states and will allocate all of its constraints to the other one. The modelElim function modifies the components that define the model, in particular the accounting set. A similar function appeared in the original formulation of $\mathcal{HOTL}$. However, due to the inclusion of time issues, this function must be adapted to deal with the new considerations. The modelElim function is constructed in two steps. First, we define how to modify the *accounting*. The $\texttt{countElim}(\mathcal{A}, s_1, s_2)$ function shows how an accounting $\mathcal{A}$ is updated when a state s_2 is modified because it is equal to another state s_1. Basically, we move all the accounting information from s_2 to s_1. The new accounting set is constructed by joining two sets. The first set denotes the accounting for all states different from s_1 and s_2. A register $(s, i, outs, f, \delta, n) \in \mathcal{A}$, with $s \neq s_1$ and $s \neq s_2$, will change only if there exists a registered transition from s to s_2 with input i, that is, if $f(s \xrightarrow{i/o} s_2) > 0$ for some $o \in outs$. In this case, the information provided by f must denote the replacement of s_2 by s_1: We set $f(s \xrightarrow{i/o} s_2) = 0$ and we increase the value of $f(s \xrightarrow{i/o} s_1)$. Additionally, we need to update the temporal information provided by the δ function. Finally, all information concerning s_2 is removed. Let us note that this operation may lead to inconsistent models from a temporal point of view. It can happen that for the state s with input i the pairs $\delta(s \xrightarrow{i/o} s_1) = (\hat{d_1}, H_1)$ and $\delta(s \xrightarrow{i/o} s_2) = (\hat{d_2}, H_2)$ are incompatible, either because $\hat{d_1} \cap \hat{d_2} = \emptyset$ or because $H_1 \uplus H_2 \not\subseteq \hat{d_1} \cap \hat{d_2}$, where $\uplus$ denotes the union of multisets introduced in Definition 1.

Definition 6. Let $p = (\hat{d_1}, H_1)$, $q = (\hat{d_1}, H_1) \in \mathcal{I}_{\mathbb{R}_+} \times \wp(\mathbb{R}^+)$. We denote by $p + q$ the pair defined as

$$p + q = \begin{cases} (\hat{d_1} \cap \hat{d_2}, H_1 \uplus H_2) & \texttt{if } \hat{d_1} \cap \hat{d_2} \neq \emptyset \ \wedge \ H_1 \uplus H_2 \subseteq \hat{d_1} \cap \hat{d_2} \\ \texttt{error} & \text{otherwise} \end{cases}$$

Let $\mathcal{A}$ be a set of *accounting registers* and s_1, s_2 be states. Then, we have

$$\texttt{countElim}(\mathcal{A}, s_1, s_2) =$$

$$\left\{ (s, i, outs, f', \delta', n) \;\middle|\; \begin{array}{l} s \notin \{s_1, s_2\} \wedge (s, i, outs, f, n) \in \mathcal{A} \wedge \\ f'(s \xrightarrow{i/o} s') = \begin{array}{ll} f(s \xrightarrow{i/o} s') & \texttt{if } s' \neq s_1, s_2 \\ f(s \xrightarrow{i/o} s_1) + f(s \xrightarrow{i/o} s_2) & \texttt{if } s' = s_1 \\ 0 & \texttt{if } s' = s_2 \end{array} \\ \wedge \\ \delta'(s \xrightarrow{i/o} s') = \begin{array}{ll} \delta(s \xrightarrow{i/o} s') & \texttt{if } s' \neq s_1, s_2 \\ \delta(s \xrightarrow{i/o} s_1) + \delta(s \xrightarrow{i/o} s_2) & \texttt{if } s' = s_1 \\ ([0, \infty], \emptyset) & \texttt{if } s' = s_2 \end{array} \end{array} \right\}$$

$$\bigcup$$

$$\left\{ \begin{array}{c} s_1, i, \\ outs_1 \cup outs_2, \\ f', \delta', n \end{array} \;\middle|\; \begin{array}{l} \exists\, p, q, g, h, \delta_1, \delta_2 : \\ \quad ((s_1, i, outs_1, g, \delta_1, p) \in \mathcal{A} \ \vee \ (s_2, i, outs_2, h, \delta_2, q) \in \mathcal{A}) \wedge \\ n = \quad \{m \mid (s, i, outs, f, \delta, m) \in \mathcal{A}, s \in \{s_1, s_2\}\} \wedge \\ f'(tr) = \quad \{f(tr) | (s, i, outs, f, \delta, m) \in \mathcal{A}, s \in \{s_1, s_2\}\} \wedge \\ \delta'(tr) = \quad \{\delta(tr) | (s, i, outs, f, \delta, m) \in \mathcal{A}, s \in \{s_1, s_2\}\} \end{array} \right\}$$

We assume that for all $n \in \mathbb{N}$ we have $n + \top = \top$. □

The previous function is auxiliary to define how to eliminate a state s when we discover that this state is equal to another state s'. In the following we will denote by $[x/y]$ the renaming of any occurrence of x by y. As before, we can reach an *inconsistent* result. For example, if the behavior of the state s with input i is *closed*, that is, $(s, i, outs, f, \top) \in \mathcal{A}$, and s' has an outgoing transition labelled with i/o, being $o \notin outs$, then the model resulting by joining s and s' would be inconsistent because it would not preserve the closed behavior of s. In this case, an empty set of models will be returned (see case (a) of the following definition). The same happens if there exists $(s, i, outs, f, \delta, m) \in \texttt{countElim}(\mathcal{A}, s_1, s_2)$ such that for some $tr \in \mathcal{T}[s_2/s_1]$ we have $\delta(tr) = \texttt{error}$ (see case (b)). Otherwise, the new model is obtained by *substituting* all occurrences of the state to be eliminated by the state that will stay (see case (c)). In the next definition we use the following property: For all index $j \in \{1, 2\}$, the expression $3 - j$ always denotes the other number of the set.

Definition 7. Let $m = (\mathcal{S}, \mathcal{T}, \mathcal{I}, \mathcal{A}, \mathcal{E}, \mathcal{D}, \mathcal{O})$ be a model and $s_1, s_2 \in \mathcal{S}$. We define the predicate $\texttt{modelElim}(m, s_1, s_2)$ as $\texttt{models}\,(\mathcal{M})$, where $\mathcal{M}$ is constructed as follows:

(a) If there exist $i, outs, f, o, s$, and $j \in \{1, 2\}$ such that $s_{3-j} \xrightarrow{i/o} s \in \mathcal{T}$, $(s_j, i, outs, f, \top) \in \mathcal{A}$, and $o \notin outs$, then $\mathcal{M} = \emptyset$.
(b) If there exists $(s, i, outs, f, \delta, m) \in \texttt{countElim}(\mathcal{A}, s_1, s_2)$ such that for some $tr \in \mathcal{T}[s_2/s_1]$ we have $\delta(tr) = \texttt{error}$, then $\mathcal{M} = \emptyset$.
(c) Otherwise, $\mathcal{M} = \left\{ \begin{pmatrix} \mathcal{S}\backslash\{s_2\}, \mathcal{T}[s_2/s_1], \mathcal{I}[s_2/s_1], \\ \texttt{countElim}(\mathcal{A}, s_1, s_2), \mathcal{E}[s_2/s_1], \\ \mathcal{D}[s_2/s_1], \mathcal{O} \end{pmatrix} \right\}$

The previous function can be generalized to operate over *sets* of states as follows. Let $S \subseteq \mathcal{S}$ be a set of states. We have

$$\texttt{modelElim}(m, s, S) = \begin{array}{ll} \{m\} & \texttt{if } S = \emptyset \\ \bigcup_{j=1}^{k} \texttt{modelElim}(m'_j, s, \{s_2, \ldots, s_n\}) & \texttt{if } S = \{s_1, \ldots, s_n\} \end{array}$$

where $\{m'_1, \ldots, m'_k\} = \texttt{modelElim}(m, s, s_1)$. □

The rest of the rules belonging to $\mathcal{HOTL}$ do not vary in their formulation. It is only necessary to consider that those rules using `countElim` have to consider the temporal version of this function (that is, temporal issues are *transparent* in the formulation of those rules).

4 New $\mathcal{THOTL}$ Hypotheses

In this section we will extend the repertory of hypotheses to include assumptions about temporal constraints of the transitions. The tester may assume that "the IUT cannot spend more (less) than t time units for producing the output o after it receives the input i" or "the pair i/o never takes more (less) than t time units."

This kind of hypotheses affects the whole IUT, so they will be included in the set `Hyp`. Besides, we need to define specific rules to apply the different hypotheses to our models and reflect how they are affected by their application.

Some of the new rules use the `updTime` function. Its role is to update the accounting set in the model to reflect the constrains established by the considered temporal hypotheses. The new accounting set is constructed by joining two sets. The first set denotes the accounting for all registers of states that either do not belong to S or have an input that does not belong to I. These registers will never change. The second set denotes the new registers. A register will change only if there exists a registered transition from $s \in S$ with input $i \in I$ such that $o \in O \cap outs$, that is, if $f(s \xrightarrow{i/o} s') > 0$ for some s'. In this case, the information provided by δ must denote the temporal constraint. One more time, we can reach an inconsistency if $\delta(s \xrightarrow{i/o} s') = (\hat{d}, H)$ and the temporal restriction imposed by the hypothesis, given by a time interval $\hat{d}'$, is incompatible due to either $\hat{d} \cap \hat{d}' = \emptyset$ or $H \not\subseteq \hat{d} \cap \hat{d}'$. In this case, the function δ will return `error`. Then, the resulting model would be inconsistent because it would not preserve the timed behavior for some transition. In this case, an empty set of models will be returned by the function.

First, we introduce a function to update the temporal functions of the accounting registers.

Definition 8. Let $\mathcal{A}$ be a set of *accounting registers*, $S \subseteq \mathcal{S}$, $I \subseteq \texttt{inputs}_{spec}$, $O \subseteq \texttt{outputs}_{spec}$, and $\hat{d}' \in \mathcal{I}_{\mathbb{R}_+}$. We define $\texttt{accTime}(\mathcal{A}, S, I, O, \hat{d}')$ as the set of accounting registers

$$\left\{ (s', i', outs, f, \delta, n) \;\middle|\; (s' \notin S \vee i' \notin I \;\vee outs \cap O = \emptyset) \;\wedge\; (s', i', outs, f, \delta, n) \in \mathcal{A} \right\}$$

$$\bigcup$$

$$\left\{ \begin{array}{c} s, i, outs, \\ f, \delta', n \end{array} \;\middle|\; \begin{array}{l} \exists\, f, outs, \delta, n : \\ \quad (s, i, outs, f, \delta, n) \in \mathcal{A} \;\wedge\; s \in S \wedge i \in I \;\wedge\; outs \cap O \neq \emptyset \;\wedge \\ \quad \delta'(tr) = \begin{cases} \delta(s' \xrightarrow{i'/o'} s'') & \texttt{if } s' \notin S \vee i' \notin I \;\vee o' \notin outs \cap O \\ (\hat{d} \cap \hat{d}', H) & \texttt{if } \delta(s \xrightarrow{i/o} s') = (\hat{d}, H) \wedge \\ & \qquad \hat{d} \cap \hat{d}' \neq \emptyset \;\wedge\; H \subseteq \hat{d} \cap \hat{d}' \\ \texttt{error} & \text{otherwise} \end{cases} \end{array} \right\}$$ □

Let $m = (\mathcal{S}, \mathcal{T}, \mathcal{I}, \mathcal{A}, \mathcal{E}, \mathcal{D}, \mathcal{O})$ be a model. We define $\texttt{updTime}(m, S, I, O, \hat{d}')$ as the model

(a) $\emptyset$, if there exist $(s, i, outs, f, \delta, n) \in \texttt{accTime}(\mathcal{A}, S, I, O, \hat{d}')$ and $tr \in \mathcal{T}$ such that $\delta(tr) = \texttt{error}$;
(b) $(\mathcal{S}, \mathcal{T}, \mathcal{I}, \texttt{accTime}(\mathcal{A}, S, I, O, \hat{d}'), \mathcal{E}, \mathcal{D}, \mathcal{O})$, otherwise.

Next we introduce new hypotheses. The first two ones allow us to assume that the implementation never takes more (less) than t time units to produce the output o since it receives the input i.

$$(maxTime)\frac{\texttt{model}\,(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A},\mathcal{E},\mathcal{D},\texttt{Obs})\ \wedge\ \texttt{maxTime}(i,o,t)\in\texttt{Hyp}}{\texttt{models}\,(\texttt{updTime}(m,\mathcal{S},\{i\},\{o\},[0,t]))}$$

$$(minTime)\frac{\texttt{model}\,(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A},\mathcal{E},\mathcal{D},\texttt{Obs})\ \wedge\ \texttt{minTime}(i,o,t)\in\texttt{Hyp}}{\texttt{models}\,(\texttt{updTime}(m,\mathcal{S},\{i\},\{o\},[t,\infty]))}$$

The tester can assume that the performance of the pair i/o always consumes exactly t time units.

$$(oclockTime)\frac{\texttt{model}\,(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A},\mathcal{E},\mathcal{D},\texttt{Obs})\ \wedge\ \texttt{oclockTime}(i,o,t)\in\texttt{Hyp}}{\texttt{models}\,(\texttt{updTime}(m,\mathcal{S},\{i\},\{o\},[t,t]))}$$

The logic $\mathcal{THOTL}$ allows to consider other temporal hypotheses about the IUT. For example, the predicate $\texttt{alwMax}(t)$ (resp. $\texttt{alwMin}(t)$) assumes that all pair i/o is performed in at most (resp. at least) t time units.

$$(alwaysMax)\frac{\texttt{model}\,(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A},\mathcal{E},\mathcal{D},\texttt{Obs})\ \wedge\ \texttt{alwMax}(t)\in\texttt{Hyp}}{\texttt{models}\,(\texttt{updTime}(m,\mathcal{S},\mathcal{I},\mathcal{O},[0,t]))}$$

$$(alwaysMin)\frac{\texttt{model}\,(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A},\mathcal{E},\mathcal{D},\texttt{Obs})\ \wedge\ \texttt{alwMin}(t)\in\texttt{Hyp}}{\texttt{models}\,(\texttt{updTime}(m,\mathcal{S},\mathcal{I},\mathcal{O},[t,\infty]))}$$

Another plausible assumption is that all actions that can be performed from a state s spend at least/most t time units.

$$(allOutMax)\frac{\texttt{model}\,(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A},\mathcal{E},\mathcal{D},\texttt{Obs})\ \wedge\ \texttt{allOutMax}(s,t)\in\texttt{Hyp}}{\texttt{models}\,(\texttt{updTime}(m,\{s\},\mathcal{I},\mathcal{O},[0,t]))}$$

$$(allOutMin)\frac{\texttt{model}\,(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A},\mathcal{E},\mathcal{D},\texttt{Obs})\ \wedge\ \texttt{allOutMin}(s,t)\in\texttt{Hyp}}{\texttt{models}\,(\texttt{updTime}(m,\{s\},\mathcal{I},\mathcal{O},[t,\infty]))}$$

The $\texttt{allTimes}(n)$ hypothesis allows to assume that if an input/output pair is produced n times at a given state then all the time values that the implementation may take at this state to perform this pair will belong to the interval delimited by the minimum and maximum observed time values.

$$(allTimes)\frac{\texttt{model}\,(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A},\mathcal{E},\mathcal{D},\texttt{Obs})\ \wedge\ n\in\mathbb{N}\ \wedge\ \texttt{allTimes}(n)\in\texttt{Hyp}}{\texttt{models}\,(\{(\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{A}',\mathcal{E},\mathcal{S},\texttt{Obs})\})}$$

where

$$\mathcal{A}' = \{(s,i,outs,f,\delta,n')|(s,i,outs,f,\delta,n')\in\mathcal{A}\ \wedge\ n'<n\}$$

$$\bigcup$$

$$\left\{(s,i,outs,f,\delta,n')\ \middle|\ \begin{array}{l}(s,i,outs,f,\delta,n')\in\mathcal{A}\ \wedge\ n'\geq n\ \wedge\\ \forall\ o\in outs:\quad \{f(tr)|tr=s\xrightarrow{i/o}s'\}<n\end{array}\right\}$$

$$\bigcup$$

$$\left\{(s,i,outs,f,\delta_s^i,n')\ \middle|\ \begin{array}{l}(s,i,outs,f,\delta,n')\in\mathcal{A}\ \wedge\ n'\geq n\ \wedge\\ \exists\ o\in outs:\quad \{f(tr)|tr=s\xrightarrow{i/o}s'\}\geq n\end{array}\right\}$$

and

$$\delta_s^i(s_1 \xrightarrow{i'/o'} s_2) = \begin{cases} \delta(s_1 \xrightarrow{i'/o'} s_2) & \texttt{if } s_1 \neq s \vee i' \neq i \vee \\ & \quad \{f(tr) | tr = s \xrightarrow{i/o'} s'\} < n) \\ (\hat{d}, \pi_2(\delta(s_1 \xrightarrow{i'/o'} s_2))) & \text{otherwise} \end{cases}$$

$$\hat{d} = \begin{array}{l} \texttt{min}\{t | \exists\, s_1 \xrightarrow{i'/o'} s' : t \in \pi_2(\delta(s_1 \xrightarrow{i'/o'} s'))\}, \\ \texttt{max}\{t | \exists\, s_1 \xrightarrow{i'/o'} s' : t \in \pi_2(\delta(s_1 \xrightarrow{i'/o'} s'))\} \end{array}$$

The new accounting set $\mathcal{A}'$ is the union of three set of registers. The first set collects those registers that do not change because they correspond to states whose outgoing transitions have been observed less than n times. The second set gathers the registers associated to states whose outgoing transitions have been observed at least n times, but all input/output pairs attached to them have been observed less than n times. The third set introduces the assumption for those registers $(s, i, outs, f, \delta, n')$ that present more than n observations of some pair i/o in the transitions outgoing from s. In this case, we must update the temporal function. For all transition tr leaving the state s and labelled by i/o we change the associated interval. This interval, $\hat{d}$, has as lower (resp. upper) bound the minimum (resp. maximum) time values observed in all the transitions outgoing from s and labelled by i/o.

We have shown some rules that may lead to inconsistent models. In some of these cases, an empty set of models is produced, that is, the inconsistent model is eliminated. Before granting conformance, we need to be sure that at least one model belonging to the set is consistent. $\mathcal{HOTL}$ already provides us with a rule that labels a model as *consistent*. Let us note that the inconsistences created by a rule can be detected by the subsequent applications of rules. Thus, a model is free of inconsistencies if for any other rule either it is not applicable to the model or the application does not modify the model (that is, it deduces the same model). Due to space limitations we do not include the details of this rule (the formal definition can be found in [21]).

Similar to $\mathcal{HOTL}$, in order to check whether a model conforms to the specification we have to take into account that only the conformance of consistent models will be considered. In addition, given a consistent model, we will check its conformance with respect to the specification by considering the *worst* instance of the model, that is, if this instance conforms to the specification then any other instance extracted from the model does so. This worst instance is constructed as follows: For each state s and input i such that the behavior of s for i is not closed *and* either s is not deterministic or no transition with input i exists in the model, a new *malicious* transition is created. The new transition is labelled with a special output *error* that does not belong to $\texttt{outputs}_{spec}$. This transition leads to a new state $\perp$ having no outgoing transitions. Since the specification cannot produce the output *error*, this worst instance will conform to the specification only if the unspecified parts of the model are not relevant for the correctness of the IUT it represents.

Definition 9. Let $m = (\mathcal{S}, \mathcal{T}, \mathcal{I}, \mathcal{A}, \mathcal{E}, \mathcal{D}, \texttt{Obs})$ be a model. We define the *worst temporal instance* of the model m with respect to the considered specification *spec*, denoted by $\texttt{worstTempCase}(m)$, as the $\texttt{TFSM}$

$$\mathcal{S} \cup \{\bot\}, \texttt{inputs}_{spec}, \texttt{outputs}_{spec} \cup \{error\},$$

$$\left\{ s \xrightarrow{i/o}_{\hat{d}} s' \;\middle|\; \begin{array}{l} s \xrightarrow{i/o} s' \in \mathcal{T} \wedge \\ \exists\, outs, f, \delta, n : (s, i, outs, f, \delta, n) \in \mathcal{A} \wedge \\ o \in outs \;\wedge\; \pi_1(\delta(s \xrightarrow{i/o} s')) = \hat{d} \end{array} \right\}$$

$$\bigcup$$

$$\left\{ s \xrightarrow{i/error}_{[o,\infty]} \bot \;\middle|\; \begin{array}{l} s \in \mathcal{S} \;\wedge\; i \in \texttt{inputs}_{spec} \wedge \\ \nexists\, outs : (s, i, outs, f, \delta, \top) \in \mathcal{A} \wedge \\ (s \notin \mathcal{D} \vee \;\nexists\, s', o : s \xrightarrow{i/o} s') \end{array} \right\}, \mathcal{I}$$

□

Thus, the rule for indicating the correctness of a model is

$$(correct)\frac{\begin{array}{c} m = (\mathcal{S}, \mathcal{T}, \mathcal{I}, \mathcal{A}, \mathcal{E}, \mathcal{D}, \texttt{Obs}) \;\wedge\; \texttt{consistent}(m) \wedge \\ H = \texttt{reduce}(\texttt{Obs}) \;\wedge\; \texttt{worstTempCase}(m)\ \texttt{conf}_{int}^{H}\ spec \end{array}}{\texttt{models}\,(\{\texttt{correct}(m)\})}$$

where

$$\texttt{reduce}(\texttt{Obs}) = \{(i_1/o_1/t_1, \ldots, i_n/o_n/t_n) | (a_1, i_1/o_1/t_1, \ldots, a_n, i_n/o_n/t_n, a_{n+1}) \in \texttt{Obs}\}$$

Now we can consider the conformance of a set of models. A set conforms to the specification if all the elements do so and the set contains at least one element. Note that an empty set of models denotes that all the models were inconsistent.

$$(allCorrect)\frac{\texttt{models}\,(\mathcal{M}) \;\wedge\; \mathcal{M} \neq \emptyset \;\wedge\; \mathcal{M} = \{\texttt{correct}(m_1), \ldots, \texttt{correct}(m_n)\}}{\texttt{allModelsCorrect}}$$

Now that we have presented the set of deduction rules, we introduce a correctness criterion. In the next definition, in order to uniquely denote observations, fresh names are assigned to them.

Definition 10. Let *spec* be a $\texttt{TFSM}$, $\texttt{Obs}$ be a set of observations, and $\texttt{Hyp}$ be a set of hypotheses. Let $A = \{ob = o \mid ob \text{ is a fresh name } \wedge\; o \in \texttt{Obs}\}$ and $B = \{h_1 \in \texttt{Hyp}, \ldots, h_n \in \texttt{Hyp}\}$, where $\texttt{Hyp} = \{h_1, \ldots, h_n\}$.

If the deduction rules allow to infer $\texttt{allModelsCorrect}$ from the set of predicates $C = A \cup B$, then we say that C *logically conforms to spec* and we denote it by C $\texttt{logicConf}$ *spec.* □

In order to prove the validity of our method, we have to relate the deductions obtained by using our logic with the notion of conformance introduced in Definition 5. The *semantics* of a predicate is described in terms of the set of TFSMs that fulfill the requirements given by the predicate; given a predicate p, we denote this set by $\nu(p)$. Due to space limitations we cannot include the definition of ν (despite the differences, the construction is similar to that in [21] for classical finite state machines). Let us consider that P is the conjunction of all the considered observation and hypothesis predicates. Intuitively, the set $\nu(P)$ denotes all the TFSMs that can produce these observations and fulfill these hypotheses, that is, all the TFSMs that, according to our knowledge, can *define* the IUT. So, if our logic deduces that these TFSMs conform to the specification (i.e., `allModelsCorrect` is obtained) then the IUT actually conforms to the specification.

Theorem 1. Let *spec* be a TFSM, Obs be a set of observations, and Hyp be a set of hypotheses. Let $A = \{ob = o \mid ob \text{ is a fresh name } \wedge \ o \in \texttt{Obs}\} \neq \emptyset$ and $B = \{h_1 \in \texttt{Hyp}, \ldots, h_n \in \texttt{Hyp}\}$, where $\texttt{Hyp} = \{h_1, \ldots, h_n\}$. Let $C = A \cup B$ be a set of predicates and $H = \texttt{reduce}(\texttt{Obs})$. Then, $C\,\texttt{logicConf}\,spec$ iff for all TFSM $M \in \nu(\bigwedge_{p \in C})$ we have $M\ \texttt{conf}_{int}^{H}\ spec$ and $\nu(\bigwedge_{p \in C}) \neq \emptyset$.

Corollary 1. Let IUT and *spec* be TFSMs, Obs be a set of observations, and Hyp be a set of hypotheses. Let $A = \{ob = o \mid ob \text{ is a fresh name } \wedge \ o \in \texttt{Obs}\} \neq \emptyset$ and $B = \{h_1 \in \texttt{Hyp}, \ldots, h_n \in \texttt{Hyp}\}$, where $\texttt{Hyp} = \{h_1, \ldots, h_n\}$. Let $C = A \cup B$ and $H = \texttt{reduce}(\texttt{Obs})$. If $IUT \in \nu(\bigwedge_{p \in C})$ then $C\,\texttt{logicConf}\,spec$ implies $IUT\ \texttt{conf}_{int}^{H}\ spec$. If there exists $M \in \nu(\bigwedge_{p \in C})$ such that $M\ \texttt{conf}_{int}^{H}\ spec$ does not hold then $C\,\texttt{logicConf}\,spec$ does not hold.

5 Conclusions and Future Work

In this paper we have presented $\mathcal{THOTL}$: A timed extension of $\mathcal{HOTL}$ to deal with systems presenting temporal information. What started as a simple exercise, where only a couple of rules were going to be modified, became a much more difficult task. The inclusion of time complicated not only the original framework, with a more involved definition of the *accounting* and the functions that modify it, but added some new complexity with the inclusion of new rules. The first task for future work is to produce a longer version of this paper were all the issues that either could not be included or could not be explained with enough details, are considered. This includes to elaborate on the semantics of predicates. The second task is to construct, taking as basis the current paper and [18], a stochastic version of $\mathcal{HOTL}$.

References

1. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
2. Batth, S.S., Rodrigues Vieira, E., Cavalli, A., Uyar, M.Ü.: Specification of timed EFSM fault models in SDL. In: Derrick, J., Vain, J. (eds.) FORTE 2007. LNCS, vol. 4574, pp. 50–65. Springer, Heidelberg (2007)
3. Bayse, E., Cavalli, A., Núñez, M., Zaïdi, F.: A passive testing approach based on invariants: Application to the WAP. Computer Networks 48(2), 247–266 (2005)
4. Bosik, B.S., Uyar, M.Ü.: Finite state machine based formal methods in protocol conformance testing. Computer Networks & ISDN Systems 22, 7–33 (1991)

5. Brandán Briones, L., Brinksma, E.: Testing real-time multi input-output systems. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 264–279. Springer, Heidelberg (2005)
6. Brinksma, E., Tretmans, J.: Testing transition systems: An annotated bibliography. In: Cassez, F., Jard, C., Rozoy, B., Dermot, M. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 187–195. Springer, Heidelberg (2001)
7. Cardell-Oliver, R.: Conformance tests for real-time systems with timed automata specifications. Formal Aspects of Computing 12(5), 350–371 (2000)
8. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2000)
9. En-Nouaary, A., Dssouli, R.: A guided method for testing timed input output automata. In: Hogrefe, D., Wiles, A. (eds.) TestCom 2003. LNCS, vol. 2644, pp. 211–225. Springer, Heidelberg (2003)
10. Fecko, M.A., Uyar, M.Ü., Duale, A.Y., Amer, P.D.: A technique to generate feasible tests for communications systems with multiple timers. IEEE/ACM Transactions on Networking 11(5), 796–809 (2003)
11. Higashino, T., Nakata, A., Taniguchi, K., Cavalli, A.: Generating test cases for a timed I/O automaton model. In: 12th Int. Workshop on Testing of Communicating Systems, IWTCS 1999, pp. 197–214. Kluwer Academic Publishers, Dordrecht (1999)
12. Huang, G.-D., Wang, F.: Automatic test case generation with region-related coverage annotations for real-time systems. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 144–158. Springer, Heidelberg (2005)
13. Krichen, M., Tripakis, S.: An expressive and implementable formal framework for testing real-time systems. In: Khendek, F., Dssouli, R. (eds.) TestCom 2005. LNCS, vol. 3502, pp. 209–225. Springer, Heidelberg (2005)
14. Ladani, B.T., Alcalde, B., Cavalli, A.R.: Passive testing - a constrained invariant checking approach. In: Khendek, F., Dssouli, R. (eds.) TestCom 2005. LNCS, vol. 3502, pp. 9–22. Springer, Heidelberg (2005)
15. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines: A survey. Proceedings of the IEEE 84(8), 1090–1123 (1996)
16. Merayo, M.G., Núñez, M., Rodríguez, I.: A brief introduction to $\mathcal{THOTL}$. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 501–510. Springer, Heidelberg (2007)
17. Merayo, M.G., Núñez, M., Rodríguez, I.: Formal testing from timed finite state machines. Computer Networks 52(2), 432–460 (2008)
18. Núñez, M., Rodríguez, I.: Towards testing stochastic timed systems. In: König, H., Heiner, M., Wolisz, A. (eds.) FORTE 2003. LNCS, vol. 2767, pp. 335–350. Springer, Heidelberg (2003)
19. Petrenko, A.: Fault model-driven test derivation from finite state models: Annotated bibliography. In: Cassez, F., Jard, C., Rozoy, B., Dermot, M. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 196–205. Springer, Heidelberg (2001)
20. Rodríguez, I., Merayo, M.G., Núñez, M.: A logic for assessing sets of heterogeneous testing hypotheses. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, pp. 39–54. Springer, Heidelberg (2006)
21. Rodríguez, I., Merayo, M.G., Núñez, M.: $\mathcal{HOTL}$: Hypotheses and observations testing logic. Journal of Logic and Algebraic Programming 74(2), 57–93 (2008)
22. Springintveld, J., Vaandrager, F., D'Argenio, P.R.: Testing timed automata. Theoretical Computer Science 254(1-2), 225–257 (2001); Previously appeared as Technical Report CTIT-97-17, University of Twente (1997)

Model-Based Firewall Conformance Testing

Achim D. Brucker[1], Lukas Brügger[2,⋆], and Burkhart Wolff[3]

[1] SAP Research, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany
achim.brucker@sap.com

[2] Information Security, ETH Zurich, 8092 Zurich, Switzerland
lukas.bruegger@inf.ethz.ch

[3] Universität des Saarlandes, 66041 Saarbrücken, Germany
wolff@wjpserver.cs.uni-sb.de

Abstract. Firewalls are a cornerstone of todays security infrastructure for networks. Their configuration, implementing a firewall policy, is inherently complex, hard to understand, and difficult to validate.

We present a substantial case study performed with the model-based testing tool HOL-TestGen. Based on a formal model of firewalls and their policies in higher-order logic (HOL), we first present a derived theory for simplifying policies. We discuss different test plans for test specifications. Finally, we show how to integrate these issues to a domain-specific firewall testing tool HOL-TestGen/FW.

Keywords: Security Testing, Model-based Testing, Firewall, Conformance Testing.

1 Introduction

It is common knowledge today that unrestricted access to the Internet is a security risk. Firewalls, i. e., active network elements that can filter network traffic, are a widely used tool for controlling the access to computers in a (sub)network and services implemented on them. In particular, firewalls filter (based on different criteria) undesired traffic, e. g., TCP/IP packets, out of the data-flow going to and from a (sub)network. Of course, their intended behavior, i. e., the *firewall policy*, varies from network to network according to the needs of its users. Therefore, firewalls can be configured to implement various security policies. Since configuring and maintaining firewalls is a highly error-prone task, the question arises how they can be tested systematically.

Several approaches for the generation of test-cases are well-known: while *unit-test* oriented test generation methods essentially use preconditions and postconditions of system operation specifications, *sequence-test* oriented approaches essentially use temporal specifications or automata based specifications of system behavior. Usually, firewalls combine both aspects: whereas firewall policies are static, the underlying *network protocols* may depend on protocol states which some policies are aware of. This combination of complexity and criticality makes firewalls a challenging and rewarding target for security testing.

⋆ This work was partially funded by BT Group plc.

In this paper, we present a case study for the HOL-TestGen system: we model firewall policies as packet filtering functions or transformations over them in higher-order logic (HOL). Thus, we can cover data-oriented as well as temporal security policies. In contrast to a previous paper [5], where the underlying theoretical questions of this case study were settled (how can reactive sequence-testing be realized in a unit-test framework?), we present in this paper our firewall test-theory in greater detail. In particular, we present three aspects which we consider crucial for the treatment of larger test problems: first, we show how theory-specific rules can be safely derived, which greatly simplifies the partition space and its computation; in our case, this applies to the simplification of policies. Second, we present ways to express test-purposes within different test-plans of the same test specification leading to different test-cases. Third, we show how these various forms of support may be wrapped together to build a domain-specific extension of HOL-TestGen called HOL-TestGen/FW, i. e., a tool for model-based firewall conformance testing.

This paper is structured as follows: after introducing the foundations in Section 2 we will introduce our formal model of firewall configurations in Section 3. Thereafter, in Section 4, we show how this model can be used for model-based test-case generation and, moreover, build the basis for a domain specific test tool: HOL-TestGen/FW. In Section 5 we will present experimental data on several case-studies and develop test strategies. Finally, in Section 6, we compare to related work and draw conclusions.

2 Background

2.1 Isabelle and Higher-Order Logic

Isabelle [11] is a generic theorem prover; new object logics can be introduced by specifying their syntax and natural deduction inference rules. Among other logics, Isabelle supports HOL (called Isabelle/HOL), which we choose as basis for HOL-TestGen.

Higher-order logic (HOL) [1, 6] is a classical logic with equality enriched by total higher-order functions. HOL is a language typed with Hindley/Milner polymorphism; type variables are denoted by α, β, γ, etc. HOL 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, maps, 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.

2.2 The HOL-TestGen System

HOL-TestGen is an *interactive*, i. e., semi-automated, test tool for specification based tests built upon Isabelle/HOL. Its theory and implementation has been

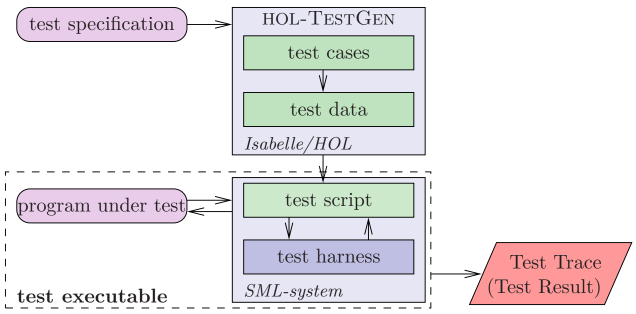


Fig. 1. Overview of the Standard Workflow of HOL-TestGen

described elsewhere [3–5]; here, we briefly review main concepts and outline the standard workflow. The latter is divided into four phases: writing the *test specification* TS, generation of *test cases* TC along with a *test theorem* for TS, generation of *test data* TD, i. e., constraint-free instances of TC, and the *test execution* (*result verification*) phase involving runs of the "real code" of the program under test. Figure 1 illustrates the overall workflow. Once a test theory is completed, documents can be generated that represent a formal test plan. The test plan contains the test theory, test specifications, configurations of the test data and test script generation commands, possibly extended by proofs for rules that support the overall process and can be processed either in batch mode or interactively. After test data generation, HOL-TestGen produces a *test script* driving the test using the provided *test harness*.

The core of the test-case generation procedure lies in the introduction of case splits up to a certain depth for each free or universally quantified variable in the test specification; depth and form of the case split depend on the type of the variable. The resulting formula is transformed into a CNF that is normalized. The test-data generation procedure is a constraint-solving technique based on a combination of arithmetic reasoning, simplification, and the insertion of random values for variables occurring in the test theorem.

2.3 Firewalls in a Nutshell

In a network, e. g., based on TCP/IP, a message from A to B is encapsulated in *packets* which contain the content of the message and routing information. The routing information of a packet mainly contains its source address (where does the packet come from), its destination address (where should the packet go to) and the protocol (e. g., http, smtp) used on top of the transport layer.

In its simplest form, a firewall is just a *stateless packet filter* which simply filters (i. e., rejects or accepts) traffic from one network to another based on the destination address, source address and the protocol, the *policy* used. The policy is the specification (usually given in a configuration file) of the firewall that describes which packets should be accepted and which should be rejected. In some cases, stateless filtering is not enough, some application protocols, like FTP or most of the protocols used for Internet telephony such as Voice over IP (VoIP) have an internal state of which the firewall must be aware of. For example, some connections are only allowed within a specific state of the protocol.

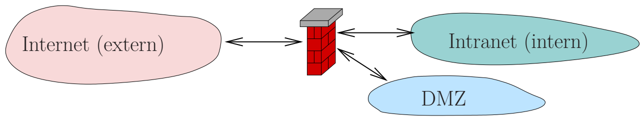


Fig. 2. A simple firewalling scenario

Figure 2 illustrates a widely-used setup of a firewall, separating three networks: the external Internet, the internal network that has to be protected (intranet) and a network that is somewhat in-between, the demilitarized zone (DMZ). The DMZ is usually used for servers (e. g., the Web server and the Mail server) that should be accessible both from the outside (Internet) and the from internal network (intranet) and thus underlie a more relaxed policy than the intranet. Table 1 shows a simple description of a firewall policy as it can be found in configuration files. Such a policy description uses a first-fit pattern matching

Table 1. A simple Firewall Policy

source	destination	protocol	action
DMZ	intranet	any	deny
Internet	DMZ	smtp	allow
Internet	DMZ	http	allow
intranet	DMZ	smtp	accept
intranet	DMZ	imaps	accept
intranet	Internet	http	accept
any	any	any	deny

strategy, i. e., the first match overrides later ones. For example, a packet from the Internet to the intranet is rejected (it only matches the last line of the table) whereas an smtp-packet from the intranet to the DMZ is accepted (fourth line of Table 1). The lines of such a table are also called *rules*; together, they describe the *policy* of a firewall. Since we consider *policy descriptions* as a kind of concrete syntax for *policies*, we will use these terms synonymously.

3 Modeling Firewalls in HOL

In this section, we present a formalization of firewall policies and present a formal theory for their simplification. Our model is inspired by the abstractions used by firewall configuration languages for TCP/IP networks, e. g., iptables (`http://www.netfilter.org`). The formalization is parametrized over the representation of network addresses for which we will discuss some alternatives in Section 4.

3.1 A Formal Firewall Model

Packets and Networks. As a prerequisite for modeling firewall policies, we need a formal model of protocols, packets and networks: we model protocols as an abstract datatype, e. g., the most common ones are declared by:

$$\text{protocol} := \text{ftp} \mid \text{http} \mid \text{voip} \mid \text{smtp} \mid \text{imap} \mid \text{imaps} \mid \text{unknown}.$$

As we do not want to depend on a specific representation of addresses and content, we introduce the abstract types $\alpha\,\text{src}$ and $\alpha\,\text{dest}$ of type $\alpha\,\text{adr}$ for the source and destination addresses and $\beta\,\text{content}$ for the content. We also introduce a unique identifier id for each packet. Thus, the type of a packet is defined as:

$$(\alpha, \beta)\,\text{packet} := \text{id} \times \text{protocol} \times \alpha\,\text{src} \times \alpha\,\text{dest} \times \beta\,\text{content}.$$

Further, we define projectors, e. g., getId, getSrc, for accessing the different components of a packet directly. As a next step, we model networks, or just *nets*, and parts thereof (*subnets*). To be as abstract as possible at this stage, we model nets as an axiomatic type class [11]. For the purpose of this paper, it suffices to know that a net is a set of sets of addresses, i. e.,

$$\alpha\ \text{subnet} := (\alpha :: \text{net})\,\text{set set}$$

where $(\alpha :: \text{net})$ requires that the types we use to instantiate α are members of the type class net. This definition allows us to model firewall policies that restrict the traffic between subnetworks and also between single hosts (addresses). For checking, if a given address is part of a subnet, we define the following operator:

$$a \sqsubset S \equiv \exists s \in S.\, (a \in s) \qquad \text{with type } \alpha\,\text{adr} \Rightarrow \alpha\,\text{subnet} \Rightarrow \text{bool}\,.$$

Firewall Policies. From an abstract point of view, a policy is a partial mapping of packets to decisions, e. g., deny or accept. Moreover, the datatype

$$\alpha\,\text{out} := \text{accept}\,\alpha \mid \text{deny}$$

allows also to model the modifications of return packets;[1] Our model can capture address-translation techniques (network address translation (NAT)) realized by some firewalls as well. The type of a policy follows directly from this:

$$(\alpha, \beta)\,\text{policy} := (\alpha, \beta)\,\text{packet} \rightharpoonup ((\alpha, \beta)\,\text{packet})\,\text{out}$$

where $\tau \rightharpoonup \tau'$ denotes the partial mapping (i. e., a type synonym for $\tau \Rightarrow \tau'$ option). Rules and policies have the same type, i. e., we can introduce a type synonym

$$(\alpha, \beta)\,\text{rule} := (\alpha, \beta)\,\text{policy}$$

[1] Usually, firewall policies describe more fine-grained how packets are denied, e. g., packets can be silently discarded (often called *drop*) or packets can be rejected (resulting in an error message on the sender side).

for rules. Moreover, an override operator for partial mappings ($\_ \oplus \_$) allows for nicely combining several rules to a policy. For example, $r_1 \oplus r_2$ combines the rules r_1 and r_2 where r_1 overrides (has higher precedence than) r_2. We can define several *generic rule combinators* at this abstract level that substantially simplify the formalization of concrete policies. For example, the usual two "catch-all" rules for accepting or denying all traffic are expressed as:

$$\begin{aligned} \text{allowAll}\, p &\equiv \text{Some}(\text{accept}\, p) && \text{with type } (\alpha,\beta)\,\text{rule, and} \\ \text{denyAll}\, p &\equiv \text{Some}(\text{deny}) && \text{with type } (\alpha,\beta)\,\text{rule.} \end{aligned}$$

Many other combinators for restricting traffic based on its source, destination, or protocol can already be defined on this abstraction level. A rule allowing all packets coming from subnet s can be defined as

$$\text{allowAllFrom}\, s \equiv \text{Some}\,\text{allowAll} \upharpoonright_{\left\{p \mid (\text{getSrc}\, p) \sqsubset s\right\}}$$

with type $(\alpha :: \text{net})\,\text{subnet} \Rightarrow (\alpha,\beta)\,\text{rule}$, and where $\_ \upharpoonright \_$ is the restriction operator on partial mappings.

IPV4. At this point, we decide for one possible packet address format, namely IPv4 addresses together with ports. In this setting, an address consists of a unique 32 bit number, represented as four-tuple and a port:

$$\begin{aligned} \text{ipv4Ip} &:= \text{int} \times \text{int} \times \text{int} \times \text{int}\,, \\ \text{port} &:= \text{int}\,, \\ \text{ipv4} &:= \text{ipv4Ip} \times \text{port}. \end{aligned}$$

Based on these definitions, we can define further combinators that are specific to TCP/IP addresses, i. e., they can accept or reject packets based on an IP address and a port.

3.2 Modeling Our Running Example

Our abstract firewall model, presented in the last section, allows for the direct formalization of the informal policy given in Table 1. First we have to define the subnets of type ipv4 subnet, based on their IP address ranges, e. g.:

$$\begin{aligned} \text{intranet} &\equiv \left\{\left\{\left((a,b,c,d),p\right) \middle| \, (a = 172) \wedge (b = 168)\right\}\right\} \text{and} \\ \text{dmz} &\equiv \left\{\left\{\left((a,b,c,d),p\right) \middle| \, (a = 172) \wedge (b = 16) \wedge (c = 70)\right\}\right\}. \end{aligned}$$

Grouping the rules of our informal policy with the same source and same destination, we define:

$$\text{DmzIntranet} \equiv \text{denyAllFromTo dmz intranet}$$

$$\begin{aligned}\text{InternetDMZ} &\equiv \text{allowProtFromTo smtp internet dmz}\\ &\quad \oplus \text{allowProtFromTo https internet dmz}\\ \text{IntranetDMZ} &\equiv \text{allowProtFromTo smtp intranet dmz}\\ &\quad \oplus \text{allowProtFromTo imaps intranet dmz}\\ \text{IntranetInternet} &\equiv \text{allowProtFromTo http intranet internet}\end{aligned}$$

The complete policy can then be defined as follows:

$$\begin{aligned}\text{policy} \equiv \text{DmzIntranet} \oplus \text{IntranetDMZ} \oplus \text{IntranetInternet}&\\ \oplus \text{InternetDMZ} \oplus \text{denyAll}.& \qquad (1)\end{aligned}$$

This definition implies that the firewall does not take the port numbers into account for its filtering decision. Of course there do also exist combinators for that case and we could, alternatively, define:

$$\text{IntranetInternet} \equiv \text{allowProtFromPortTo http 80 Intranet internet.}$$

3.3 Simplifying Firewall Policies

The presented formalization of firewall policies is obviously not a canonical one, i. e., the same policy can be represented by many (syntactically) different maps. As an example, consider the following equivalence:

$$\text{denyAllFromTo } X\ Y \oplus \text{denyAll} = \text{denyAll}$$

where X and X are variables that are implicitly universally quantified, i. e., this equivalence holds for all possible values of X and Y. This observation leads to the idea of using rewriting techniques for simplifying firewall policies while preserving their semantics. For this purpose, we developed a set of equivalence rules that can be used by the built-in simplifier of the underlying Isabelle system. In more detail, the simplifier is configured to use such equivalences over the policy combinators as rewrite rules (from left to right). As we provide a large set of policy combinators, we also need a quite large set of equivalences over them. Thus, we can only summarize the rule set here.

One category of such equivalences are the ones handling global coverage, as the example above. Other such examples include the following:

$$\text{allowProtFromTo } p\ X\ Y \oplus \text{allowAllFromTo } X\ Y = \text{allowAllFromTo } X\ Y.$$

Another category of equivalences reduces the number of individual rules as it summarizes similar rules like in the following example:

$$\begin{aligned}\text{allowProtFromTo}\, p\ A\ B \oplus \text{allowProtFromTo}\, q\ A\ B&\\ = \text{allowProtsFromTo } \{p, q\} A\ B.&\end{aligned}$$

Together with several theorems about the associativity and commutativity of disjunct rules, we can derive a powerful policy simplification theory within the framework. E.g., our example policy can be simplified from seven to four individual rules. In detail, the policy gets simplified to

$$\begin{aligned}&\text{allowProtsFromTo}\ \{\text{smtp}, \text{http}\}\ \text{internet}\ \text{dmz}\\ &\oplus \text{allowProtsFromTo}\ \{\text{smtp}, \text{imaps}\}\ \text{intranet}\ \text{dmz}\\ &\oplus \text{allowProtFromTo}\ \text{http}\ \text{intranet}\ \text{internet}\\ &\oplus \text{denyAll}.\end{aligned}$$

The representation of a firewall policy (i. e., the selection of basic rules) influences both the runtime performance of a firewall and the decision during test-case generation. The simplifier-set presented in this section is aimed at reducing the number of decision points and thus makes the policy easier to test (see Section 5), i. e., results in smaller sets of test cases. On the other hand, the introduction of additional decision points (e. g., allowing to throw packets away earlier during the matching phase) can increase the overall performance of a firewall. Of course, such an optimization needs to take knowledge about the traffic into account and thus cannot be automated as easily.

4 Testing Firewall Policies

4.1 Testing Stateless Firewalls

The *test specification* for the stateless firewall case is now within reach: basically, we just state that the *firewall under test (fut)* has the same filtering function behavior as our combined policy from Definition 1:

$$\boxed{\mathit{fut}(x) = \text{policy}(x)}$$

However, this test specification is too general as we are only interested in a subset of the possible packets. The main reason is that firewalls sit *between* the subnets and therefore do not observe all the traffic. In particular, they will not observe packets with the source and destination within the same subnet.

Using a general logic as underlying framework, we can easily constrain the test space accordingly. If we want to test the firewall depicted in Figure 2, we use the auxiliary predicate:

$$\begin{aligned}\text{notInSameNet}\ \text{x} \equiv\ &(\text{srcOf}\ \text{x} \sqsubset \text{internet} \longrightarrow \neg\,\text{destOf}\ \text{x} \sqsubset \text{internet})\\ &\wedge (\text{srcOf}\ \text{x} \sqsubset \text{intranet} \longrightarrow \neg\,\text{destOf}\ \text{x} \sqsubset \text{intranet})\\ &\wedge (\text{srcOf}\ \text{x} \sqsubset \text{dmz} \longrightarrow \neg\,\text{destOf}\ \text{x} \sqsubset \text{dmz})\end{aligned}$$

This predicate of type *packet* → *bool* checks if the source and the destination of a packet are not within the same subnet. The test specification is revised to:

$$\boxed{\text{notInSameNet}\ (x) \longrightarrow \mathit{fut}(x) = \text{policy}(x)}$$

Depending on the *test purpose*, other constraints for the test specification are possible. For example, we might focus on test-data which represent packets sent to one single host, or only test-data for specific protocols. Thus, there may be different test specifications expressing different test purposes for the same policy. Distinguishing test purposes is especially useful for networks with firewalls at different points still enforcing one global policy.

After writing the test specification, some theorem proving techniques are necessary to bring the test theorem into a form which is suitable for the test-case generation; as these techniques are always the same for the default applications, this can be done fully automatically. In more detail, this includes the unfolding of the policy and of the rules. Technically, one can either unfold the constraints of the test specification before or after the test-case generation. In the first case, the undesired cases will not be generated while in the latter case they will be discharged later.

The test-case generation procedure, possibly followed by a simplification, produces after about 45 minutes running-time on a modestly equipped workstation, a list of 258 test-cases, among them the following two:

1. A test-case where a packet to the intranet must be denied by the firewall if the protocol is neither imap nor smtp:

$$\frac{X_2 = 172 \longrightarrow X_3 \neq 168 \qquad X_1 \neq \text{imap} \qquad X_1 \neq \text{smtp}}{\begin{array}{r}\mathit{fut}(X_4, X_1, ((X_2, X_3, X_5, X_6), X_7), ((172, 168, X_8, X_9), X_{10}), X_{11}) \\ = \text{Some deny}\end{array}}$$

   Here, the assumptions represent constraints for the concrete values chosen for the variables. In particular, the first part of the assumptions guarantees that the test-data generated from this test-case have a source network that is not equal to the destination network.
2. A test-case where the firewall should accept an smtp packet from the intranet to the dmz:

$$\begin{aligned}&\mathit{fut}(X_{12}, \text{smtp}, ((172, 168, X_{13}, X_{14}), X_{15}), ((172, 16, 70, X_{16}), X_{17}), X_{18})\\ &\;= \text{Some}(\text{accept}(X_{12}, \text{smtp}, ((172, 168, X_{13}, X_{14}), X_{15}),\\ &\qquad\qquad ((172, 16, 70, X_{16}), X_{17}), X_{18}))\end{aligned}$$

We proceed with the test-data generation, which generates a constraint-free, ground instance of each test-case by random-constraint-solving. Unlike other examples, this step is quite trivial in our case and the computation time is negligible compared to the time used in the test-case generation. The procedure basically has to guess concrete values for the variables of the test-cases. Some of them are constrained (e. g., X_1 above), others are completely unconstrained (e. g., X_{12} above). Here is a sample of the generated test-data:

1. $\mathit{fut}(12, \text{http}, ((7, 13, 12, 0), 6), ((172, 168, 2, 1), 4), \text{content}) = \text{Some deny}$
2. $\mathit{fut}(8, \text{smtp}, ((172, 168, 12, 13), 12), ((172, 16, 70, 10), 6), \text{content})$
   $= \text{Some}(\text{accept}(8, \text{smtp}, ((172, 168, 12, 13), 12), ((172, 16, 70, 10), 6), \text{content}))$

The test data can be fed into a real firewall test driver. To sum up, a classical unit test scenario is adequate for stateless packet filters.

4.2 Testing Stateful Firewall Policies

For protocols like FTP, a stateless firewall can only provide a very limited form of network protection. The reason for this is, that FTP is based on a dynamic negotiation of a port number which is then used as channel to communicate the file content between the sender and the receiver. Thus, a stateful firewall is needed to observe the inner state of the port negotiation. Testing stateful firewalls, where the filter functions change, requires test-sequence generation also supported by HOL-TestGen.

The detailed model of a stateful firewall and protocols is presented in [5]. The basic idea is to keep the definition of policy but extend the model by a state, which consists of a pair of a *history* of accepted packets and the *current policy*. A state transition is a mapping from the packet that fired the transition and the current state to a new state. A state machine which models a specific stateful protocol can then be defined using predefined combinators.

In the stateful case, the test-data are *lists* of packets. The test specification can then be combined with the stateless case, e. g., perform a stateless testing before and after successful execution of the file transfer protocol (FTP). Clearly, the role of test purposes is even more important here.

4.3 Network Models

Besides the network model presented in Section 3.1, there are two notable alternatives; in the sequel, we evaluate them with respect to the generation time and test-case numbers.

Networks as a Datatype. A possible abstraction from the network representation is to model subnets as elements of a finite datatype. This reads as follows:

$$\text{datatype networks} = \text{dmz} \mid \text{intranet} \mid \text{internet}$$

This model, built with ports or without, reduces the task of the test-case generation drastically; instead of case-splits for four independent Integers, only one is introduced. However, this representation abstracts away the possibility of single hosts since an additional random number generator at the level of the test-driver will be necessary to replace abstract networks against concrete addresses. Thus, different instances for one single host will be generated at runtime of the test.

IPv4 Addresses as one single Integer. Although typically represented as a four-tuple of (mathematical) Integers, an IP address is technically simply a 32-bit bitvector. A replacement is straightforward: subnets are expressed as ranges of Integers. This formalization is compromise between the other two: the possibility to model single hosts is kept while the representation is simplified drastically. We provide a conversion between these two representations.

4.4 A Domain-Specific Test Tool for Firewall Policies

So far, we presented a theory of networks, protocols and policies together with their use for generating test-cases. As HOL-TestGen is built on the framework of Isabelle with a general plug-in mechanism, HOL-TestGen can be customized to implement domain-specific, model-based test tools in its own right. As an example for such a domain-specific test-tool, we developed HOL-TestGen/FW which extends HOL-TestGen by:

1. a theory (or library) formalizing networks, protocols and firewall policies,
2. domain-specific extensions of the generic test-case procedures (tactics), and
3. support for an export format of test-data for external tools such as [13].

Figure 3 shows the overall architecture of HOL-TestGen/FW.

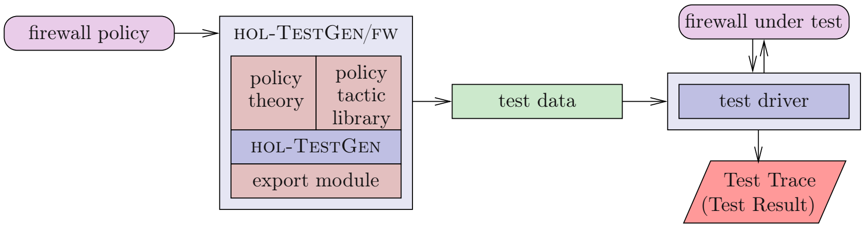


Fig. 3. The HOL-TestGen/FW architecture

In fact, item 1 defines the formal semantics (in HOL) of a specification language for firewall policies; see Section 3 for details. On the technical level, this library also contains simplification rules together with the corresponding setup of the constraint resolution procedures.

With item 2 we refer to domain-specific processing encapsulated the general HOL-TestGen test-case generation. Since test specifications in our domain have a specific pattern consisting of a limited set of predicates and policy combinators, this can be exploited in specific pre-processing and post-processing of an optimized version of the procedure, now tuned for stateless firewall policies. Moreover, there are new control parameters for the simplification (see Section 3.3).

With item 3, we refer to an own XML-like format for exchanging test-data for firewalls, i. e., a description of packets to be send together with the expected behavior of the firewall. This data data can be imported in a test-driver for firewalls, for example [13]. This completes our toolchain which, thus, supports the execution of test data on firewall implementations based on test cases derived from formal specifications.

5 Evaluation and Discussion

In this section, we report on experiments with HOL-TestGen/FW in several case studies. We will discuss the influence of different model parameters (e. g., network models, complexity of policies) on the number of test-cases and generation time. We conclude with a comparison of different test-strategies.

5.1 Case Studies

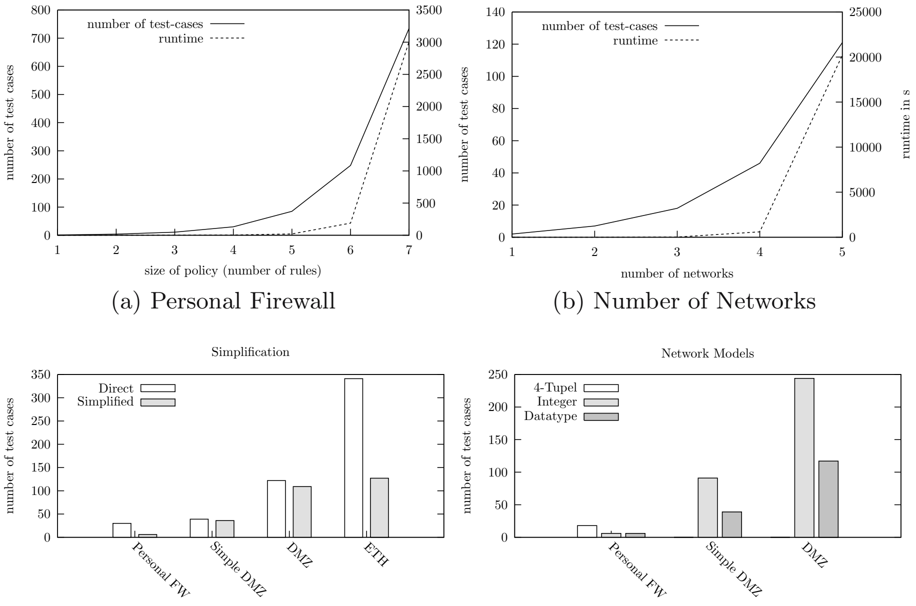
In the following, we will briefly summarize the results of the following scenarios:

Personal Firewall: We model a firewall running on a workstation, i. e., we only have two subnets, one representing the Internet and one representing a single workstation. This scenario is often called a personal firewall. A simple default policy for this model is to deny all traffic from the Internet to the workstation and to allow all traffic in the other direction.

Simple DMZ: A standard setup (similar to the example introduced in Section 2) with one internal network (intranet) that cannot be accessed from the Internet and one demilitarized zone which contains the servers that should be accessible from both the Internet and the intranet.

DMZ: We extend the "Simple DMZ" scenario by one crucial detail: the policy of each server in the DMZ is specified individually. Technically, this corresponds to the introduction of sub-subnets.

ETH: We model a firewall that is used in the computer science department at ETH Zurich: a real world example "as is." This example demonstrates that our approach is applicable for real world scenarios.



(a) Personal Firewall (b) Number of Networks

(c) Simplification (d) Comparing different Network Models

Fig. 4. Evaluation

5.2 Influence of the Policy Size

We used the "Personal Firewall" case study for investigating how the size of a policy influences the number of test-cases and the overall time needed for calculating them. In this experiment, we keep the network setting and the network model fix, and vary the policy by successively adding (non-overlapping) rules opening new ports from the workstation to the Internet.

Figure 4a suggests that both the number of test-cases and the time increases substantially with respect to the number of rules in a policy for two subnets.

5.3 Influence of the Number of Networks

By successively extending the "Personal Firewall" case study with additional workstations (subnets) using the same basic policy for every workstation (deny all traffic from the other networks to the workstation and allow all traffic from the workstation to the other networks), we studied how the number of networks influences the number of test-cases needed for testing the scenario thoroughly.

Figure 4b suggests that the number of subnets and hosts has a significant influence on the number of test-cases.

5.4 Influence of Policy Simplification

As we have seen, the complexity of a policy (number of rules) has a great consequence on the number of test-cases needed for full coverage of the specification. For justifying our policy simplification strategy, we compared the number of test-cases for several case studies (see Figure 4c).

In particular, the ETH Zurich firewall example shows that the effect of simplification for real-world scenarios can be dramatic. Of course, as explained in Section 3.3, our policy simplification can increase the network latency of deployed policies; and of course, not every policy can be simplified.

5.5 Influence of the Network Model

For investigating the influence of the network model (see Section 4.3) on the number of test-cases, we used three typical models and compared the number of generated test-cases for each model. Not surprisingly, Figure 4d reveals that the choice of the network model is most relevant. The choice to model an address as a four-tuple of Integers is on the one hand very nice as it is the closest one to the real-world representation. The costs, however, are substantial: in case of both DMZ examples, we stopped the computation after 6 hours with no result. Furthermore, we get a lot of additional test-cases which are probably in most scenarios superfluous. The possibility to model subnets as a datatype is clearly the simplest one and also significantly reduces computation time. However, we loose the possibilities to model hierarchical network topologies directly. For example, addressing single hosts, as in the DMZ example, requires special handling in the definition of the notInSameNet predicate. Therefore, we prefer the option of representing addresses by a single Integer: with only a little more complexity than in the datatype case, we keep the same expressive power as in the four-tuple case.

5.6 Discussion

Figure 4 summarizes the different scenarios; the details of all case studies appearing in this section and all measured data are part of the HOL-TestGen distribution (`http://www.brucker.ch/projects/hol-testgen/`). The presented case studies have demonstrated that our tool is well applicable to real world scenarios.

A classical goal of test-case generation is to minimize the number of test-cases. On the one hand, as the test-data generation itself needs only to be done once for every policy, one could argue that the time needed for computing the test-cases is not that important. On the other hand, von Bidder [13] reports that even only replaying test-data on a real firewall implementation already takes substantial time. Therefore, optimizing the set of test-cases is important, while preserving good test-coverage. Based on this experiments, we suggest to follow different test strategies, depending on the concrete test purpose:

– Our experience shows that testing, with complete decision coverage, of policies that are highly-optimized in terms of network latency, seems to be very expensive. As a compromise, we suggest to simplify the policy for testing purposes and to generate test-cases with respect to this simplified policy. While this approach still guarantees path coverage with respect to the specification, it does not cover all paths of the implementation. Nevertheless, by increasing the number of test-data chosen for every test-case, the coverage of the implementation could be increased. Overall, this approach results in a combination of model-based testing and random testing.
– For highly critical applications it might be worthwhile to install several firewalls with small policies that can be tested thoroughly. Moreover, as every firewall can be optimized for the small number of networks it connects, we expect a performance gain on the implementation level. Of course, installing several firewalls increases at least the initial costs; as maintaining small policies is much easier, we would expect that the total cost of ownership of such a setting is, at maximum, only a little worse than one centralized firewall. In fact, a similar setting was chosen for the network of ETH Zurich, where every research group has its own firewall.

6 Conclusion and Related Work

6.1 Related Work

Firewall testing is a widespread research topic which reflects their importance in todays security infrastructures. Whereas we focus on conformance testing, many research approaches are focused on vulnerability and policy-independent implementation testing. Surprisingly, we did not find any work on random testing the conformance of a firewall with respect to a policy; this is in stark contrast to software testing where random testing has gained a certain popularity.

Several approaches for specification-based testing of firewalls have been proposed. For example, El-Atawy et al. [7, 8] present a policy segmentation technique where a policy is represented as a tree. They also give some measurements to the segments such that important segments can be tested more rigorously. They also present a policy generation technique. Jürjens and Wimmel [9] propose a specification-based testing of firewalls which employs a formal model of the network and automatically derives test-cases. It does however not really describe in which way these test-cases are generated. Furthermore, it does not support an ordering of the firewall rules. Bishop et al. [2] describe a formal model of protocols in HOL. Their level of abstraction is however much lower than ours and is therefore less suited for testing of policy conformance. Marmorstein and Kearns [10] propose a policy-based host classification which can be used to detect errors and anomalies in a firewall policy. Senn et al. [12, 13] propose a simple language for specifying firewall policies and a framework for testing firewalls at the implementation level. While the framework includes tools for generating test-cases with respect to a protocol specification, it lacks support for test-case generation based on policies.

6.2 Conclusion and Future Work

We presented a family of case study for HOL-TestGen consisting of a formal model of firewall policies in HOL, several problem-specific test plans and domain-specific tool support for generating test-cases. Our integrated approach allows for both the formal analysis of a policy, e. g., certain properties could be proven interactively, their automatic simplification, and the automatic generation of tests for real firewall implementations. We believe to have presented a successful application of our methods and tools to real-world scenarios.

The general domain of testing firewalls is both technically challenging and as well a rewarding target of security testing. Moreover, the variety of different firewall implementations, all based on the same set of network protocol specifications, makes firewalls especially well-suited for a model-based testing based on abstractions.

The presented work can be extended into various directions. In [5], we used sequence testing techniques for testing stateful firewalls using HOL-TestGen. This allows for the integration of unit and sequence testing, i. e., in every state of a test sequence a unit-test is executed.

On the theoretical side, our framework could be used for formally analyzing different separation techniques on the level of networks. For example, a common technique for reducing the amount of test-cases is to partition the policy into different fragments where every fragment only covers one pair of subnetworks. While this clearly reduces the amount of test labor, it also introduces a new kind of test hypothesis ("traffic between two networks does not influence the policy for other networks") into the system. Overall, this would allow to analyze formally the effects of a heuristic that is usually applied in an ad-hoc manner in firewall testing.

On the practical side, several extensions can be made: first, our integrated test-harness generator could be configured for generating test-data in a format that is suitable as input for the tools developed in the context of [13]. This would allow for testing the conformance of deployed firewalls. Secondly, our policy specification can be used for generating configuration artifacts (e. g., scripts for iptables) for real firewall implementations and thus HOL-TestGen/FW could be used for testing and configuring real firewalls. And Thirdly, an integration of HOL-TestGen/FW into standard firewall configuration tools is possible. Like the second approach, this would allow for a policy specification language that is used for testing and configuration of a real firewall.

Acknowledgment

We would like to thank Martin Sedler for valuable discussions about the firewall settings at the computer science department, ETH Zurich (D-INFK/ETH).

References

[1] Andrews, P.B.: Introduction to Mathematical Logic and Type Theory: To Truth through Proof, 2nd edn. Kluwer Academic Publishers, Dordrecht (2002)

[2] Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., Wansbrough, K.: Engineering with logic: HOL specification and symbolic-evaluation testing for Tcp implementations. In: Gregory Morrisett, J., Peyton Jones, S.L. (eds.) POPL, pp. 55–66. ACM Press, New York (2006)

[3] Brucker, A.D., Wolff, B.: HOL-TestGen 1.0.0 user guide. Technical Report 482, ETH Zurich, April (2005a)

[4] Brucker, A.D., Wolff, B.: Symbolic test case generation for primitive recursive functions. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 16–32. Springer, Heidelberg (2005)

[5] Brucker, A.D., Wolff, B.: Test-sequence generation with HOL-TestGen. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 149–168. Springer, Heidelberg (2007)

[6] Church, A.: A formulation of the simple theory of types. Journal of Symbolic Logic 5(2), 56–68 (1940)

[7] El-Atawy, A., Ibrahim, K., Hamed, H., Al-Shaer, E.: Policy segmentation for intelligent firewall testing. In: NPSec 2005, pp. 67–72. IEEE Computer Society, Los Alamitos (2005)

[8] El-Atawy, A., Samak, T., Wali, Z., Al-Shaer, E., Lin, F., Pham, C., Li, S.: An automated framework for validating firewall policy enforcement. In: POLICY 2007, pp. 151–160. IEEE Computer Society, Los Alamitos (2007)

[9] Jürjens, J., Wimmel, G.: Specification-based testing of firewalls. In: Bjørner, D., Broy, M., Zamulin, A.V. (eds.) PSI 2001. LNCS, vol. 2244, pp. 308–316. Springer, Heidelberg (2001)

[10] Marmorstein, R., Kearns, P.: Firewall analysis with policy-based host classification. In: LISA 2006, pp. 4–4. USENIX Association (2006)

[11] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)

[12] Senn, D., Basin, D., Caronni, G.: Firewall conformance testing. In: Khendek, F., Dssouli, R. (eds.) TestCom 2005. LNCS, vol. 3502, pp. 226–241. Springer, Heidelberg (2005)

[13] von Bidder, D.: Specification-based Firewall Testing. Ph.D. Thesis, ETH Zurich, ETH Diss. No. 17172. Diana von Bidder's maiden name is Diana Senn (2007)

VCSTC: Virtual Cyber Security Testing Capability – An Application Oriented Paradigm for Network Infrastructure Protection

Guoqiang Shu, Dongluo Chen, Zhijun Liu, Na Li, Lifeng Sang, and David Lee

Department of Computer Science and Engineering, the Ohio State University
Columbus, OH 43210, USA
{shug,chendon,liuzh,lina,sangl,lee}@cse.ohio-state.edu

Abstract. Network security devices are becoming more sophisticated and so are the testing processes. Traditional network testbeds face challenges in terms of fidelity, scalability and complexity of security features. In this paper we propose a new methodology of testing security devices using network virtualization techniques, and present an integrated solution, including network emulation, test case specification and automated test execution. Our hybrid network emulation scheme provides high fidelity by host virtualization and scalability by lightweight protocol stack emulation. We also develop an intermediate level test case description language that is suitable for security tests at various network protocol layers and that can be executed automatically on the emulated network. The methodology presented in this paper has been implemented and integrated into a security infrastructure testing system for US Department of Defense and we report the experimental results.

Keywords: Network Modeling, Network Emulation, Security Testing, Test Automation, Virtualization.

1 Introduction

Security, reliability and interoperability are indispensable in today's distributed heterogeneous information infrastructures. These properties rely on the correct functioning of the increasingly complicated security devices, such as traditional firewall, security switch, intrusion detection system (IDS) and so on [9,11,13,17]. For modern security devices, testing is no longer considered to involve only the vendor because software components such as user configuration and plug-in have become significant [1,21]. On the other hand, since critical secure devices are often to be deployed at sensitive environment (e.g. government or military network), it is not appropriate to test them using the real network. Instead, target system and configuration are tested using special testbed before deployment. The last several decades have witnessed a great number of mature IP network testbed solutions [2,10], with the focus of integration testing and performance testing. In this work we study several key challenges and solutions particularly in testing network security devices.

First, the nature of security testing demands a high level of fidelity between the testbed and the real environment in order to compose realistic test scenario and obtain meaningful assessment. This cannot be achieved by the content-insensitive traffic generation paradigm often adopted in performance testing. Particularly, it is desirable that the testbed mimics the characteristics of the real network including topology, host machine properties and the protocol stack. Fidelity of protocol stack is especially important for (1) generation of realistic background traffic [20], and (2) designing test cases at transport or higher layer, or executing real applications.

While duplicating the real environment or approaches of this nature guarantees fidelity, it could be extremely expensive to scale. An enterprise network normally contains at least hundreds of physical hosts with heterogeneous configurations. Naturally a question to ask here is how many hosts will be involved in a test? While testing for properties like address blocking may require only a few, other features such as resilience against Distributed Deny-of-Service (DDoS) attack could involve much more. A promising direction toward loyal and scalable solution is to employ rapidly developing virtualization techniques [16,19]. Virtual machine solutions such as VMware ESX server can increase the testbed size roughly by an order while still preserving all the applications; and lighterweight protocol stack virtualization methods can scale much better (e.g. virtual Honeynet [14,15]) at the cost of sacrificing some real applications. In this work we propose the integration of both based on the following assumption: even in a test case involving a large number of hosts, the subset that has to run real application simultaneously is often very small. Our experience of developing a firewall/IDS test suite justifies this assumption.

The last but not least notable issue is automation. Security tests usually employ precisely specified sequence of actions from various principals, which essentially requires coordination of the external network, internal network and the device under test itself. The test system should hide this control complicacy to the end user. In addition, as the security features of sophisticated device span over multiple network layers, the test description mechanism should provide corresponding capability and at the same time facilitate automatic test execution.

Motivated by the above insights we propose Virtual Cyber Security Testing Capability (VCSTC) – a novel methodology of testing security device and the associated application solutions with high fidelity, scalability and usability. VCSTC methodology aims at a broad category of target systems that could be deployed at the boundary (gateway) of a local, usually an organizational or enterprise network. The main security-related functionality of such systems includes multi-directional access control, intrusion detection, virus/worm detection, vulnerability analysis and mitigation. Examples of such devices available on the market include Cisco ASA family, Top Layer Secure Command and many lower-end consumer security appliances. Two networks are involved in using and hence testing such devices: an internal network to be protected, and an untrusted external network. Typically there are four steps in testing: create a model for both internal and external network; emulate the two models on a testbed; develop test cases; and execute the test cases on the testbed. VCSTC methodology spans over all four steps, although model construction step is not related to testing and therefore is not the focus on this paper. To the best of our knowledge our approach is the first to integrate network emulation and automated testing, and it is distinguished from the existing security testbed solutions (such as DETER [2]) by the following two key aspects.

Hybrid network emulation: To test a security device deployed at network boundary, both internal and external network are emulated using the mixture of network host and protocol stack virtualization techniques. The emulated network contains two types of nodes: a small number of FAT nodes that are fully featured virtualized network hosts, and a large number of THIN nodes each having only a virtual TCP/IP stack. Configuration of emulated networks is automatically done according to the user network model. Hosts involved in a test case will be mapped to emulated nodes (FAT or THIN) depending on what is executed on that host. By leveraging the advantage of these two virtualization methods, our hybrid testbed achieves the best balance between loyalty and scalability. We show that our methodology can support up to one thousand emulated nodes on a commodity computer.

Test description language: In order to cater the diverse features of security device VCSTC uses an intermediate level test case description language to facilitate the specification of Point of Control and Observation (PCO) at various layers: IP, Socket and Application. PCOs could be deployed at any host and the target device, while the actually deployment and control are automatic. This language is tightly based on a programming language to provide virtually unlimited expressivity. A test case is dynamically compiled into a native executable before execution by the test driver. We evaluate this scheme by both manual test case generation and using the language as the target of model-based formal test generation methods.

The rest of this paper is organized as follows. In section 2 we provide an overview of VCSTC methodology as well as the testing system architecture. Then the two main contributions are discussed in more details. Section 3 introduces our test description language; the hybrid network emulation approach is elaborated in Section 4. The second part of the paper reports our extensive evaluation of the methodology during the development of a testing platform for the U.S. Department of Defense (DoD) [13]. Section 5 presents our experience of manual and automatic test generation, as well as a brief remark on the performance. Finally we discuss some on-going and future work in Section 6.

2 Testing Methodology and System Architecture

In VCSTC methodology there are two essential components in security testing: model and test cases. They are independent and developed separately. The network model should contain sufficient information to emulate a real network, and in the meantime not associated with any special devices and therefore generally reusable. VCSTC supports several methods to build a network model: it could be automatically synthesized using network management protocols (e.g. SNMP) and collected network traces, or using random network topology generation; or manually using prevalent modeling language such as UML with software tool assistance.

Test cases could also be constructed through various ways. A test case mainly specifies two things: (1) a set of PCOs and their deployment (2) a sequence of actions of the PCOs with the expected outcome. Test cases could be made abstract by defining parameters of numerical type or special type like IP address. Such abstract test cases could be concretized by selecting a set of parameter values. To execute a test on a given network model (assuming they are compatible) we first compile it together with all supporting libraries into an executable. Next the network model is automatically emulated and PCOs are deployed at the designated hosts, each of which is controlled by the test driver through a private communication channel. After the test case finishes a log file with all network activities during its execution is returned to the tester along with the test verdict for evaluation. In this framework high fault coverage can be achieved by the combination of three approaches - selecting different network models; generating test cases from a model of the feature under test to cover it more comprehensively; and selecting many combinations of parameter value for an abstract test case.

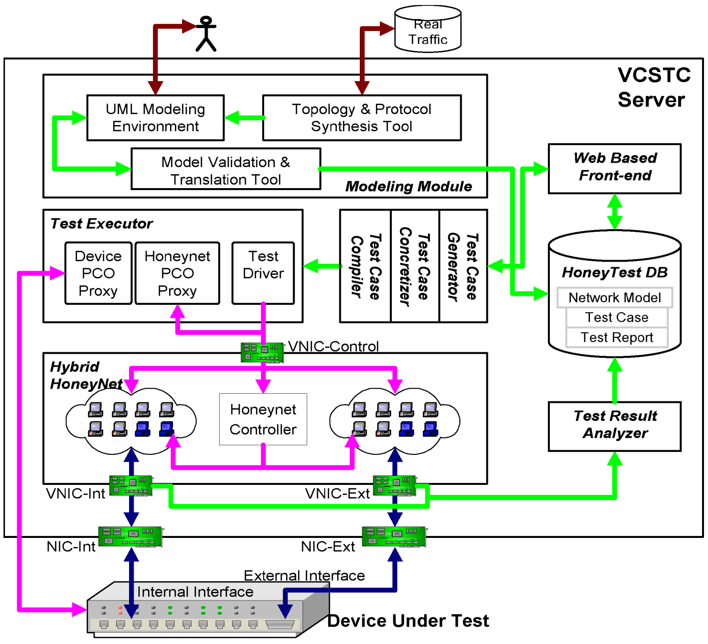


Fig. 1. Architecture of VCSTC platform capsulated within a single server that connects to both internal and external interface of a target device

Figure 1 shows the architecture of a full-featured testing system we developed that realizes the VCSTC methodology. The system is an out-of-box product that could be contained within a single server. The modeling module provides an UML compatible environment for creating and validating network models. Models are stored in a database after validation. The operational environment implements a streamline consisting test preparation, test execution and test result processing, all of which are exposed to the user via a Web-based interface. The test generation module accepts abstract test cases (as textual file) generated by the tester or from a formal model. They are then concretized based on a certain parameter selection policy and eventually compiled into a native binary file. The test executor is responsible of creating emulated network and executing test cases on it. The emulated network is essentially a virtual honeynet with hybrid nodes (details in Section 4) implemented using a pool of virtual machines with a central controller. The whole testing workflow is automated to the extent that testers interact with VCSTC server only by providing network models and test cases from Web interface. The test executor hides the complexity of controlling the network emulator and PCOs from the users.

Now we briefly describe the network configuration on the server. Three types of virtualized network interfaces connect the emulator with other components. There are many virtual Network Interface Cards (NIC) of each type grouped together by virtual networks. The private interface VNIC-Control is used by the test executor to control the honeynet and all PCOs. This type of interface is totally invisible to the test cases and the target device. The other two types of interfaces VNIC-Int and VNIC-Ext are bridged with the physical NICs NIC-Int and NIC-Ext on the server, which are connected to the internal port and external port of the target device, respectively. This setup enables emulation of both the internal and the external network, while all emulated packets generated during testing are transformed into real packets at the corresponding VNIC before delivered to the device. Note that for simplicity Figure 1 only shows one internal path to the device whereas additional NIC and VNIC could be deployed similarly according to the requirements for testing. Network traffic - both real and emulated - passing all VNIC will be monitored during test execution and readable test reports such as Message Sequence Chart (MSC) are generated afterwards for further analysis.

3 Test Specification Language

VCSTC uses its own notation for test case specification. The rationale of introducing the new notation is not to replace the traditional high level test specification language such as TTCN-3 or MSC; instead our main incentive is to provide a flexible way of developing test sequences related to security features at all layers of network protocol stack. Toward this goal our language is tightly based on a native programming language such that any valid statement of the host programming language could be embedded in the test code, providing encoding of an input symbol, for instance. A test case is a textual file with multiple declarative sections (described below) and a test code section. We show later that the proposed intermediate level language could be used to interpret test sequences from more abstract formal models such as EFSM [6].

The most important element in our language is PCO. A test case defines multiple PCOs on various places and controls their behavior. A PCO is deployed on either a host of internal/external network or the target device. Every PCO on network has one of three types: (1) Packet PCO sends and receives raw network packet of TCP, UDP or IP protocol by taking over the network interface of the host. (2) Socket PCO manages one TCP or UDP socket. (3) Application PCO handles one user application. It reads and writes to the application through its standard input/output channel. Table 1 summarizes the three types of PCO. A PCO must be bound on a host (i.e. an IP address) before it can be function, and the mapping could be done by various ways. The PCO definition might supply a fixed IP address if the test case is design for some specific network models, or otherwise the binding could be done in test code by calling run-time API. Note that multiple PCOs with different types could co-exist on the same host except for Packet PCO due to the nondeterministic behavior under the situation of multiple network capturers.

Table 1. Three types of PCOs

Type of PCO	Packet PCO	Socket PCO	App. PCO
Method of control	Send/Receive raw TCP/UDP/IP packet	Read/Write TCP/UDP socket	Execute native application
Number on each node	One	Many	Many
Blocking I/O	No	Yes	No

The actions of PCOs are defined in the test code section, which eventually returns a test verdict. A test case could contain parameters and become abstract. Abstract test case cannot be executed before assigning parameter values. Our language supports parameters of bounded Integer type and IP address type, and a test concretization algorithm implements parameter value selection according to certain coverage criteria such as random sampling, boundary coverage and so forth. After the test case is concretized it is automatically transformed into the host language for compilation. In this phase auxiliary code such as test case initialization and cleanup routines is generated and weaved together with the test code. During compilation the VCSTC runtime library proving essential functionalities and all user defined libraries are linked. In practice, a lot of reusable routines (e.g. Malware simulation, special packet generation) are encapsulated in the form of library so that the test code could focus on the logic, that is, the sequence of actions from PCO. The VCSTC runtime library implements all types of PCO and the proxies used to control them remotely from the test executor.

Now we discuss more details by an example. Figure 2 shows a test case that checks whether a security device provides an outgoing source IP black list of sufficient length. The test code uses Java as host language and implements a rather straightforward logic: a Web server is started on an external host by an Application PCO (line 8). There are many (NCLIENT) internal hosts with a Socket PCO on each (line 7). Both the server and clients are selected randomly from the network (lines 17-18). The device is controlled by a Device PCO (line 9). At the beginning the test case launches the server and clears the black list (line 22), followed by a check (lines 24-28) to see whether all clients can reach the server. Next the black list is configured through Device PCO (line 29), and we retry the connections again, expecting that no client can successfully reach the serer. Any client's success in connecting at this time (line 32) proves the black list useless and therefore the test case returns the verdict failure. The test case contains two Integer parameters (lines 2-5): number of clients and server ports, which are to be assigned according to a user policy. Note that the execution of the test case is fully automated except for Device PCO. Configuration of target device might need manual activity, depending on the interface a specific device is providing. Many venders provide programmable configuration mechanism, which could be utilized by VCSTC runtime to fully automate test execution.

We close this section by a remark on the relationship between test case and network model. Test cases like the one in Figure 2 do not depend on any network-specific properties such as background traffic and therefore could be executed on any network models. The only implicit constraint is that the network must contain enough (in this case NCLIENT) distinct hosts – this will be checked statically by the test executor when loading the test case. On the other hand, the test case might also explicitly specify a list of compatible network model names if necessary.

```
#TESTCASE OBL_Length_TCP
#PARAM {
   int{[0,512]} NCLIENTS;
   int{[0,1024], [50000,51000]} PORT;
}
#PCO {
   SOCKET pco_client[NCLIENTS];
   APP pco_server;
   DEVICE pco_device;
}
#PACKET {}
#VERDICT {
   success, failure, unknown, timeout
}
#TESTBODY
{
  bind_PCO(pco_client, INTERNAL, RANDOM|NONDUP);
  bind_PCO(pco_server, EXTERNAL, RANDOM);

  log("Testing length of black-list using TCP");
  pco_server.execute("httpd", PORT);
  pco_device.config("clear black list");
  Vector black_list = new Vector<InetAddress>();
  for (int x= 0;x<NCLIENTS;x++) {
          if(!pco_client[x].connect
                  (pco_server.getIP(),PORT)) return unknown;
          black_list.add(pco_client[x].getIPAddress());
          pco_client[x].close();
  }
  pco_device.config("add to black list", black_list);
  wait(3000);
  for (int x= 0;x<NCLIENTS;x++) {
          if(pco_client[x].connect(pco_server.getIP(),
                  PORT)) return failure;
  }
  return success;
}
```

Fig. 2. Example of a simplified abstract test case

4 Network Host and Protocol Stack Virtualization

A test case is executed on an emulated network that from the view of the device under test is the same as a real network. Emulated network is created from a network model using hybrid network virtualization approach. As we mention in Section 1, VCSTC mitigates the key challenge of scalability by using a hybrid virtual honeynet. Honeynet [18] is used recently as a best practice of network emulation for the purpose of attack identification. We adapt a hybrid honeynet design where two virtualization techniques are used together to achieve the balance of scalability and fidelity. First we distinguish two terms used in this section: *logical* node and *physical* node. A logical node is a network host, either external or internal, defined in a test case. A logical node is identified by its IP address. A physical node is a network host in the emulated network. The test executor maintains a mapping from logical nodes to physical nodes and deploys the PCOs according to this mapping. In our scheme of hybrid honeynet there are two types of physical nodes in the emulated network:

- **FAT node:** A FAT physical node is emulated by a complete virtualized host machine. Thanks to the advanced virtualization techniques such node can accommodate any application running at the real host. A repository of pre-configured (e.g. with different Operating System and/or applications) virtual machine images are stored at the server while the honeynet controller selects the proper ones to load. Unfortunately, host virtualization is still very expensive and we cannot afford to emulate the whole network using FAT nodes alone.

- **THIN node:** A THIN physical node is emulated by virtualizing only a TCP/IP protocol stack but not the actual resources of a host. Software solutions such as Honeyd [14] accomplish this by overriding the IP protocol stack on a single (possibly virtual) machine and claiming responsibility for a range of IP addresses. Socket based program could be executed on top of the virtualized protocol stack appearing to the outside as running with its own address. This approach is lightweight and therefore very scalable. The cost however, is that the function of PCO deployed on them is limited. Since all programs launch on THIN nodes share the same physical machine and therefore its resources, there is obviously a potential problem of interference. The exact constraints are determined by the virtualization tool used. In our system with Honeyd as protocol stack emulator if a logical node is mapped to a THIN node, then application PCOs on it can only execute a special type of socket-based EFSM simulation program synthesized from user network traces (see Section 6).

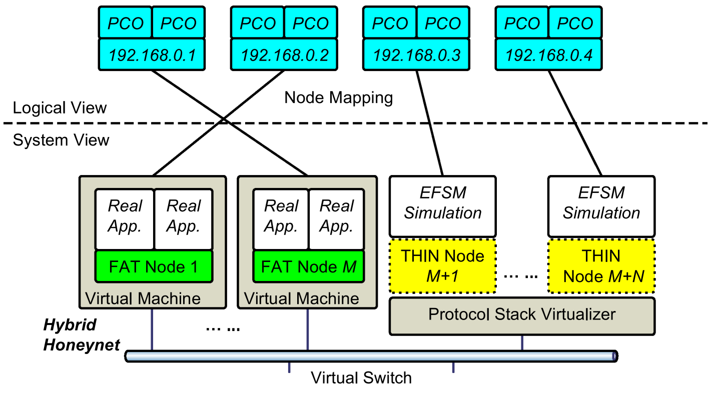


Fig. 3. Internal structure of a hybrid honeynet with two types of nodes

We create and configure a mixture of these two types of physical nodes in the emulated network, as shown in Figure 3. From the view of the emulator, we have a small number *M* of FAT nodes and a protocol stack virtualizer supporting *N* THIN nodes. These heterogeneous physical nodes are connected by a virtual network switch and form a honeynet. On the other hand, the heterogeneity is made transparent to the test cases. That is, all logical nodes are the same in terms of PCO capabilities. The mapping between logical and physical node is first created by the test executor before a test case is loaded, and is adjusted dynamically by network reconfiguration under some circumstances. The separation of logical nodes and physical nodes has two obvious advantages. First the honeynet resource provisioning could be changed anytime - for example adding more FAT nodes - without affecting any test case. Second, in most test cases only a small number of hosts run real applications (and therefore require to be mapped to a FAT node) simultaneously, despite that the total number of hosts is large. In such situations when a logical node does not have any activities we can remap it to a THIN node at runtime. When the current physical node provisioning can no longer support the execution of a test case, the test executor will get a runtime error and hence returns failure. Below we describe some heuristic guidelines practiced by the test executor for static and dynamic mapping.

Guideline 1 (Static): If a logical node does not have Application PCO, always map it to a THIN node, because Packet and Socket PCOs could both be supported.

Guideline 2 (Static): If a logical node has both Socket and Application PCO, map it to a FAT node if there is one available, otherwise map to a THIN node. Nodes with Application and Packet PCO have lower priority of mapping to FAT node. This is because Packet PCO is easier to migrate dynamically than Socket PCO.

Guideline 3 (Dynamic): Before an Application PCO executes a real user application, mapping need to be adjusted if the logical node is currently mapped to a THIN node. If there is Socket PCO with established TCP connections at this time, we report failure because we cannot migrate TCP connection across physical nodes. Otherwise, if there is an unmapped FAT node, it is remapped to the logical node. If all FAT nodes are already mapped, we check whether one of them could be swapped to a THIN node, that is, on the current owning logical node no Application PCO is executing and no Socket PCO is connected. If this condition is satisfied, honeynet controller will reconfigure the network (i.e. IP address) and switch the mapping of two logical nodes, therefore allow the user application to be executed on a FAT node. If no FAT node satisfies this condition, failure is reported.

As an example of test case, in Figure 2 we have an array of logical nodes (client) with only Socket PCO and another node (server) with only Application PCO. The mapping for this test case is trivial since only one FAT node is needed for the server node and all clients are mapped to THIN nodes.

Figure 4 shows a more illustrative example. In this test case we have two client nodes with Application PCO and a server node with Application PCO. Table 2 shows the node mapping at several key timing points when the emulated network contains unlimited THIN nodes but only 2 FAT nodes. Before executing the test case, the first two nodes (client[0] and client[1]) get the FAT nodes and the rest (including server[0]) get THIN nodes. Before the server starts (line 13), it needs to be remapped to a FAT node, and client[0] could be swapped out since it is not active. Similarly when client[0] needs to launch its program reconfiguration happens again, swapping it with client[1]. Finally client[1] launches a program, now since client[0]'s PCO has terminated its application, it could be switched to a THIN node and client[1] gets the FAT node. Note that the jitter of mapping in this example is quite unrealistic since in practice the server contains much more FAT nodes.

```
1.  ......
2.  #PCO {
3.          APP pco_client[4];
4.          APP pco_server;
5.          DEVICE pco_device;
6.  }
7.  ......
8.  #TESTBODY
9.  {
10.  bind_PCO(pco_client, INTERNAL, RANDOM|NONDUP);
11.  bind_PCO(pco_server, EXTERNAL, RANDOM);
12.  ......
13.  pco_server.execute_service("IIS6.0");
14.  ......
15.  pco_client[0].execute("lynx","domain.com/page.cgi");
16.  ......
17.  pco_client[0].terminate();
18.  ......
19.  pco_client[1].execute("iexplore","domain.com/page.cgi");
20.  ......
21.  pco_client[1].terminate();
22.  ......
23. pco_server.terminate();
24. }
```

Fig. 4. Example of test case with dynamic node remapping

Table 2. Node mapping of the test case with 2 FAT nodes

	Line 10	Line 13	Line 17	Line 19
server	THIN	FAT-1	FAT-1	FAT-1
client[0]	FAT-1	THIN	FAT-2	THIN
client[1]	FAT-2	FAT-2	THIN	FAT-2

Dynamic network reconfiguration also involves a reconnection between the PCO proxy (in the test executor) and the physical node through the network interface VNIC-Control of the honeynet (Figure 1). When the new IP address becomes usable on the physical node, the PCO proxy will disconnects the old PCO and connect the new one. On a separate issue, we are currently investigating suitable process migration schemes supporting dynamic remapping including live TCP connections, which fully take the advantage of the hybrid network design.

5 Experiments and Evaluation

The proposed VCSTC methodology has been fully applied in the development of a real security testing platform for the U.S. DoD (Department of Defense). The purpose of this project is to provide critical network infrastructure owners with an effective and easy-to-use mechanism to assess the suitability of a security device or solution with respect to their own infrastructure before investment. In this section we report our experience and evaluation during the development of this platform. We start from a brief overview of the system configuration and some simple practice in Section 5.1; then Section 5.2 summarizes our effort of integrating automatic test generation techniques. Our system supports generating test cases (in our test description language) from two popular formal models – Parameterized EFSM and Simplified Firewall Rule Language. We also present performance evaluation of the system installed on commodity hardware in order to justify its feasibility and scalability.

5.1 System Configuration and Basic Operations

As discussed earlier, the whole system could be deployed on a single machine, i.e. HoneyNet server (Figure 1). We choose a typical hardware configuration: a Dell Precision 690 workstation with two Xeon 3.2 GHz Due Core CPUs and 2GB memory. The server has two Gigabit physical NICs (NIC-Int and NIC-Ext). Both modeling module and test executor are implemented in Java 1.5 and Jpcap (a packet manipulation utility). The hybrid honeynet is composed of 5 VMware virtual machines running Ubuntu Linux as guest Operating System – 4 of them with 256MB virtual memory each are used as FAT nodes and the last one with 512MB virtual memory runs Honeyd 1.5 to emulate up to 1024 THIN nodes. The system is used to test several security devices on the market, and our performance evaluation is conducted using Netgear ProSafe FVS318 VPN Firewall/Switch.

We use both network models synthesized from real network and randomly generated large models. For real network, we derive a model from a testbed of the WAN-in-Lab project [7] developed by Caltech. This testbed has 4 Cisco routers with SNMP capability. The whole model contains 39 subnets and totally about 40 distinguished hosts with services available. We imagine the target device is about to be deployed at the gateway of this network and manually develop a small test suite that covers the classic access control and content filtering features common to typical Firewall and IDS. It takes a Java developer two days after one day's training to write about 50 test cases (Table 3). Using these test cases, the tester is able to verify precisely the details of many features of the device that is stated very informally and vaguely from its user manual. For instance, one of the Anti-virus test cases discovers that the device cannot enforce malicious URL blocking when the URL is encoded in HEX form (e.g. "www.abc.com/x.e%78e" for "www.abc.com/x.exe"), which effectively renders this URL blocking feature useless. Based on this experience we consider our test description language efficient and of good usability.

Table 3. Firewall/IDS Test Suite

Firewall Feature	Inbound filtering	24 Test cases
	Outbound filtering	24 Test cases
	Port Forwarding	4 Test cases
	Dynamic filtering	1 Test cases
Anti-virus features		2 Test cases
Intrusion detection features		3 Test cases

5.2 Automatic Test Case Generation

The applicability of our methodology could be broadened by leveraging the advanced test generation methods. We make an effort to integrate them into our methodology. We investigate automated translation from test sequences derived from formal model to the VCSTC test description language. The first model we implement is EFSM. Our test system provides a GUI to specify a feature of the device using EFSM with parameters in I/O message. Figure 5 shows an example of a single port blocking feature with two states. From the EFSM model test sequences could be automatically derived using various approaches, such as checking sequences from reachability analysis (Figure 5 shows the reachability graph when the range of *port* variable has 3 values). We then translate each test sequence into a test case file and then generate an incomplete user library that defines the I/O symbols of the model. Figure 6 shows a section of the test case corresponding to the sequence {*Set_Block*[0]/-, *Visit*[0]/-, *Set_Unblock*/-, *Visit*[0]/*Resp*} and an empty method definition for input symbol *Set_Block*, for which the test designer is responsible of providing the code to implement this input symbol using the PCOs on internal and external hosts. Note this only needs to be done once and then shared by all test cases for the same model.

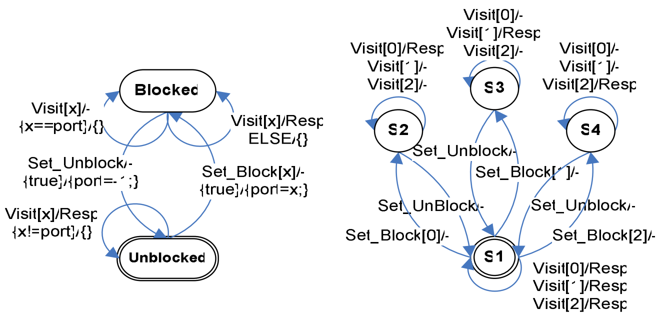


Fig. 5. A simple EFSM model of port blocking feature (left) and the corresponding reachability graph as a FSM (right)

Similarly our system supports generating test cases from firewall configurations. We use a simple grammar to describe firewall rules following classic semantics [1,11]. A rule contains a predicate based packet filter and an action and a configuration is an ordered list of rules. After a user inputs a firewall configuration, test cases with input packets are automatically generated. The elements in the packet filter can be either a value or a wildcard ("*"), and furthermore the user might ask the test case concretization process to select a value by specifying it as a parameter of the rule. For example, a configuration as specified below is composed of two rules *A* and *B*. The test case generated from this configuration will contain three parameters (i.e. Src port

```
#TESTBODY
{
   input_Set_Block(pco_ext, pco_int, pco_device,0);
   ASSERT(output(pco_ext, pco_int, pco_device) == NULL);
   input_Visit(pco_ext, pco_int, pco_device,0);
   ASSERT(output(pco_ext, pco_int, pco_device) == NULL);
   input_Set_Unblock(pco_ext, pco_int, pco_device);
   ASSERT(output(pco_ext, pco_int, pco_device) == NULL);
   input_Visit(pco_ext, pco_int, pco_device,0);
   ASSERT(output(pco_ext, pco_int, pco_device) == Resp);
   return success;
}
#USES LIB_Simple_Port_Blocking
```

```
public class LIB_Simple_Port_Blocking
{
   ……
   public void input_Set_Block(PCO pco_ext, PCO pco_int,      PCO
   pco_device) {
         … //user provides implementation of input symbol;
   }
   ……
}
```

Fig. 6. Test code and library generated from EFSM test sequences

in *A*, Protocol in *B* and Src port in *B*) and a pair of Socket PCO binding on internal and external network, respectively.

A: "Allow TCP from [10.0.0.1:*Param*$_1$] to [*.*] through External"
B: "Deny *Param*$_2$ from [*:*Param*$_3$] to [192.168.0.2:80] through External"

The test code first enables this configuration through Device PCO, and then essentially sends a packet enabling a subset of rules to see whether the device under test takes the expected action. Clearly the subset of rules triggered by a particular packet depends on the parameter value of all rules. In our example, A and B could be enabled together if $Param_2$=TCP and $Param_1 = Param_3$. In fact when both *A* and *B* are enabled they conflict with each other, and it is to the interest of the tester how the device will handle. The test case concretization process produces parameter assignments in such a way that most rule subsets are covered. Due to space limit we omit the detail of the algorithm and test case generated. From our experiences of integrating the two formal models, we believe that our methodology is promising for a variety of application domains related to network security testing.

5.3 Performance Evaluation

Finally we remark on the performance evaluation of our system. First we clarify that VCSTC is not targeted for performance testing or load testing therefore it is not designed to meet hard real-time requirements. The purpose of evaluation is instead to justify the feasibility of our design for network model and tests of practical scale. We use a series of micro-benchmarks to measure various aspects of the system, with the focus on the performance penalty incurred by using hybrid network virtualization. The first performance penalty comes from initializing the emulated network. For FAT

# of THIN Node	16	64	256	1024
Startup Time	6.67s	24.13s	30.59s	55.43s

(a)

Message Size	4KB	8KB	16KB	32KB	64KB
Trans. Time	<1ms	1.00ms	2.13ms	3.88ms	7.87ms

(b)

# Connections	1	2	3	4
Bandwidth (each con.)	6.80Mb/s	2.95Mb/s	2.08 Mb/s	1.90 Mb/s

(c)

Fig. 7. (a) Startup time of hybrid honeynet with different network size. (b) Transmission time of PCO control messages. (c) TCP transmission bandwidth between external and internal network nodes with 1-4 simultaneous connections measured by iPerf.

nodes the controller reset/reload loads a virtual machine image which takes constant time; then the Honeyd engine virtualizes the pool of THIN nodes and launch the PCO on each node. The Honeyd start up time is proportional to the number of THIN nodes as shown in Figure 7 (a), for instance a network of 1024 nodes could take up to 1 minute to initialize. Note that under certain situations it is unnecessary to reinitialize network for each test case, specifically when all test cases share a network model and they all cleanup properly. In addition, communication cost between the test driver and the PCO is not neglectable because the control message sent might carry a data portion (e.g. a packet to send from that PCO). We measure the transmission time with various message sizes shown as Figure 7 (b). There is no difference between FAT and THIN nodes since the same control channel is used.

The packet dispatching mechanism used by protocol stack virtualization tools (i.e. Honeyd) also causes delay in data transmission involving a THIN node. Basically all socket function calls are delegated to the tool and go through internal tunneling, which forms a global bottleneck. We use a benchmarking tool iPerf to measure the bandwidth of concurrent TCP connections between external and internal nodes. If both are FAT nodes the bandwidth for a single link is 8.89Mb/s; if one side is THIN node, it is downgraded as shown in Figure 7(c). We believe that this bandwidth limitation is not critical to validity of most security related tests.

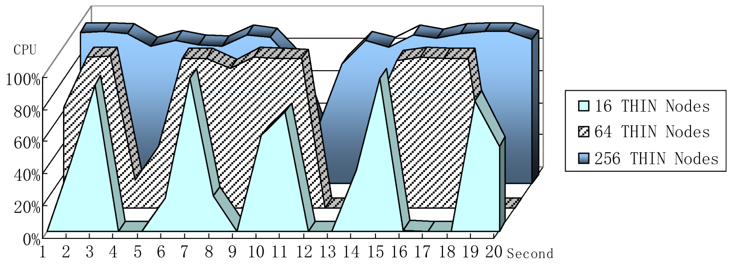


Fig. 8. Honeyd CPU load percentage for three networks of THIN nodes each sending UDP traffic at 2KB/s

Another simple benchmark is designed to evaluate approximately the work load of testing server. The dominating factor here again is large number of THIN nodes virtualized by Honeyd. We create network of different size, then let each node send UDP packet at a given rate to random destination node. This scenario corresponds to a typical test case where all logical nodes carry symmetric tasks. Figure 8 shows the CPU usage of the VMware guest OS running Honeyd during a window of 20 seconds. When the network is small (16 nodes) an average 33.4% CPU time is used while a large network (1024 nodes) is likely to saturate the CPU (85.8%). While admittedly being a coarse measurement, this shows that our system is capable of running fairly large models.

6 Discussion

In this work we present a new security testing approach, VCSTC using network host and protocol stack virtualization. Two main aspects are discussed in detail: (1) designing a scalable and yet loyal network testbed; (2) develop test cases manually or automatically. Compared to existing solutions, VCSTC has a few advantages. Our novel design of hybrid network emulation provides both fidelity (by network host virtualization) and scalability (by lightweight protocol stack virtualization). We also develop an intermediate level test description language that is suitable for security tests at various network protocol layers. In the paper we discuss how test cases are executed automatically on the emulated network model. Extensive experiments have been conducted on our implementation platform, which justify the benefits of our proposed methodology.

On the other hand, we are still at the initial stage of applying network virtualization techniques to testing. Lots of issues remain to be explored in our current approach before its applicability could be further broadened. Our approach aims at security testing at IP layer and above. As a matter of fact, VCSTC does not support routing protocol emulation despite that it generates real IP packets. Consequently, routing related security features cannot be tested under our framework. For similar reason data link layer security features are not supported. Emulating routing in a scalable fashion is a challenging task and it may change the protocol stack virtualization scheme in a drastic way. A promising approach is to use one virtualizer for each routing domain or subnet, and connect them by FAT nodes where routing protocols are implemented. Also the test language is to be augmented to support routing operations at the PCOs.

Protocol synthesis from real network is another challenge where network traffic with high fidelity is desired. This is an issue for both network modeling and testbed design. Since running real user applications on top of all virtualized nodes is clearly not practical, we need to synthesize a model of the protocol from network traces [3,12] and emulate it on the testbed in order to generate (not simply replay) traffic patterns similar to those seen. In our ongoing work we use a state machine minimization approach [5] to obtain EFSM models from field-decoded protocols (e.g. by Ethereal), and implement a special program to simulate EFSM models that could be executed on top of both FAT and THIN nodes. We envision this and the enhancement for routing emulation will render our VCSTC a more powerful and useful tool for testing both hardware and software based network security systems.

Acknowledgement. The VCSTC project was supported by the U.S. Department of Defense under grant award N41756-06-C-5541. Work in this paper is also supported in part by the U.S. National Science Foundation (NSF) under grant awards CNS-0403342, CNS-0548403. We thank Steven Low from Caltech for kindly allowing us to use the WAN-in-Lab testbed.

References

1. Al-Shaer, E., Hamed, H.: Discovery of Policy Anomalies in Distributed Firewalls. In: Proceedings of IEEE INFOCOM (2004)
2. Benzel, T., Braden, R., Kim, D., Neuman, B., Joseph, A., Sklower, K., Ostrenga, R., Schwab, S.: Experience with DETER: A Testbed for Security Research. In: 2nd IEEE Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom) (2006)
3. Cui, W., Kannan, J., Wang, H.: Discoverer: Automatic Protocol Reverse Engineering from Network Traces. In: The 16th USENIX Security Symposium (2007)
4. El-Atawy, A., Ibrahim, K., Hamed, H., Al-Shaer, E.: Policy Segmentation for Intelligent Firewall Testing. In: 1st Workshop on Secure Network Protocols (NPSec) (2005)
5. Gören, S., Ferguson, F.J.: On state reduction of incompletely specified finite state machines. Computers and Electrical Engineering 33(1), 58–69 (2007)
6. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines - A survey. In: Proceedings of the IEEE, pp. 1090–1123 (1996)
7. Lee, G.S., Andrew, L.H., Tang, A., Low, S.H.: WAN-in-Lab: Motivation, Deployment and Experiments. Protocols for Fast, Long Distance Networks (PFLDnet), 85–90 (2007)
8. Liljenstam, M., Nicol, D.M., Berk, V., Gray, R.: Simulating Realistic Network Worm Traffic for Worm Warning System Design and Testing. In: Proceedings of the 2003 Workshop on Rapid Malcode (WORM) (2003)
9. Lyu, M., Lau, L.: Firewall security: policies, testing and performance evaluation. In: Proceedings of the COMSAC, pp. 116–121 (2000)
10. Maier, S., Herrscher, D., Rothermel, K.: Experiences with node virtualization for scalable network emulation. Computer Communication 30(5), 943–956 (2007)
11. Mayer, A., Wool, A., Ziskind, E.: Fang: A Firewall Analysis Engine. In: Proceedings of the IEEE Symposium on Security and Privacy (2000)
12. Orebaugh, A., Ramirez, G., Burke, J., Pesce, L.: Wireshark & Ethereal Network Protocol Analyzer Toolkit (Jay Beale's Open Source Security). Syngress Publishing (2007)
13. Pederson, P., Lee, D., Shu, G., Chen, D., Liu, Z., Li, N., Sang, L.: Virtual Cyber-Security Testing Capability for Large Scale Distributed Information Infrastructure Protection (submitted, 2008)
14. Provos, N., Holz, T.: Virtual Honeypots: From Botnet Tracking to Intrusion Detection, 1st edn. Addison-Wesley Professional, Reading (2007)
15. Provos, N.: A Virtual Honeypot Framework. In: Proceedings of the 13th USENIX. Security Symposium (2004)
16. Sabiguero, A., Baire, A., Boutet, A., Viho, C.: Virtualized Interoperability Testing: Application to IPv6 Network Mobility. In: 18th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, pp. 187–190 (2007)
17. Sherwood, J.: The Security Certification Criteria Project. In: The 3rd International Common Criteria Conference (2002)
18. Spitzner, L.: The Honeynet Project: Trapping the Hackers. IEEE Security and Privacy 1(2), 15–23 (2003)
19. VMware Inc, http://www.vmware.com
20. Wang, L., Ellis, C., Yin, W., Luong, D.D.: Hercules: An Environment for Large-Scale Enterprise Infrastructure Testing. In: Proceedings of the Workshop on Advances and Innovations in Systems Testing (2007)
21. Wool, A.: Architecting the Lumeta firewall analyzer. In: 10th USENIX Security Symposium, pp. 85–97 (2001)

Performance Test Design Process and Its Implementation Patterns for Multi-services Systems

George Din[1], Ina Schieferdecker[1,2], and Razvan Petre[1]

[1] Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, D-10589 Berlin
[2] Technical University Berlin, Franklinstr. 28/29, D-10623 Berlin

Abstract. Over the past years, the scope of telecommunication services has increased dramatically making network infrastructure-related services a very competitive market. Additionally, the traditional telecoms are combined with Internet technologies for providing a larger range of services. The obvious outcome is the increase of the number of subscribers and services demand. Due to this complexity, the performance testing of continuously evolving telecommunication services has become a real challenge and requires efficient and more powerful testing solutions. This ability highly depends on the performance test design and on the efficient use of hardware resources for test execution.

This paper proposes a performance testing methodology which copes with the characteristics mentioned above. The methodology consists of a set of methods and patterns, exemplified for the TTCN-3 language, to realize adequate performance tests for multi-service systems. The effectiveness of this methodology is demonstrated throughout a case study on IP Multimedia Subsystem (IMS) performance testing.

1 Introduction

The Service Providers (SP) are evolving their networks from legacy technologies to "fourth generation" technologies which involves: evolution of "traditional" wireline telecom standards to Voice over IP (VoIP) standards, evolution of GSM and CDMA networks to 3GPP/3GPP2 standards (e.g. UMTS), introduction of wireless LAN (WLAN) standards (e.g. IEEE 802.16), for both data and voice communications [17]. The current direction is to realize a convergence point of these trends into a set of technologies termed the IP Multimedia Subsystem (IMS). The concept behind IMS is to support a rich set of services available to end users on either wireless or wired User Endpoints (UE), provided via a uniform interface.

The performance testing of telecommunication services raises a challenging problem: *how to create efficient tests to evaluate the performance of such systems?*. A typical multi-service system offers a number of services which can be accessed through entry-points [13]. The system consists of many sub-systems (hardware and software) communicating usually through more than one protocol. The service consumers are the end users which access the services through compatible devices called User-Endpoints (UEs). The service consumption is realized through communication protocols involving different types of transactions (e.g. authentication, charging, etc.). With the specification of IMS, the current telecommunication infrastructure is moving rapidly to a generic approach [5] to create, deploy and manage services, therefore we expect a large variety of services to be available soon in many tested systems.

The services are becoming more and more complex, requiring more computing resources on the system side and more messages exchange between involved components. The expectation is that the more complex and demanded the service is, the more the system performance decreases. The service demand is usually unpredictable since it depends on user preferences for services. As a result, the overall system load is a composition of instances from different services where each type of service contributes in a small proportion.

This paper elaborates a set of methods and patterns to design and implement efficient performance tests for systems which offer a large number of services and typically have to handle a large number of requests in short periods of time. We selected the TTCN-3 [10] technology to exemplify some of the patterns presented in the paper.

This paper is structured as follows. The next section presents several related works and it is followed by the description of the performance test design process for multi-service systems. Section 4 introduces the concrete elements. Section 5 discusses various implementation patterns. In Section 6 a case study which applies the performance test design methods to IP Multimedia System is presented. The paper ends with the conclusions section.

2 Related Work

There are various commercial and open source tools which can be used for performance testing of multi-service systems. Although there are many documents put out by software vendors, very few research papers present performance test tools design issues.

In [2] and [12] the Tcl/Tk [24] language is used to develop performance test frameworks. These papers explain the complex requirements of load testing and give a detailed overview for the extensive use of Tcl/Tk within the system. The design and implementation illustrate the code re-usability inherent in using Tcl as an application glue language.

The Faban tool [21] provides a framework to automate running of server benchmarks as well as a container to host benchmarks allowing new benchmarks to be deployed. Similarly, Weevil tool [23] is a generic automation tool for the deployment and execution of distributed workloads.

Another automated load generator and performance measurement tool for multi-tier software systems is Autoperf [18]. The tool requires a minimal system description for running load testing experiments and it is capable of generating server resource usage profiles, which are required as an input to performance models.

In [19] the Hammer [7], LoadRunner [15] and eLoad [6] are compared. The paper also motivates the implementation of an inhause tool for performance testing.

The test tool is based on Visper middle-ware [20] which is written in Java and provides the primitives, components, and scalable generic services for direct implementation of architectural and domain specific decisions.

The work presented in this paper targets a missing gap in performance testing, namely the use of the TTCN-3 language for performance testing. Our platform enables TTCN-3 test engineers use the same language for both, functional and non-functional tests, but also benefit from the use of a dedicated testing language.

3 The Performance Test Design Process

Though performance testing is a common topic among people and organizations, the research does not address performance testing of multi-service systems at a general level but rather targets only for specific types of applications e.g. web applications [14]. The methodology presented in this paper is based on the performance test design process which is depicted in Figure 1. This process refines the general process described in [9] as applied to multi-service systems.

Performance Requirements. The process starts with the selection of *performance requirements*. In this step the test engineer has to identify which features characterize the performance of the tested system. High number of transactions per unit of time, fast response times, low error rate under load conditions are examples of common performance requirements. But there are also other specific requirements which might be considered for a system in particular: availability, scalability or utilization. A detailed view on performance requirements selection is provided in [3] where a performance requirements framework integrates and catalogues performance knowledge and the development process.

Workload. The next step is to define the *workload*. The workload comprehends the performance testing focus points, test interfaces and scenarios of interest. The

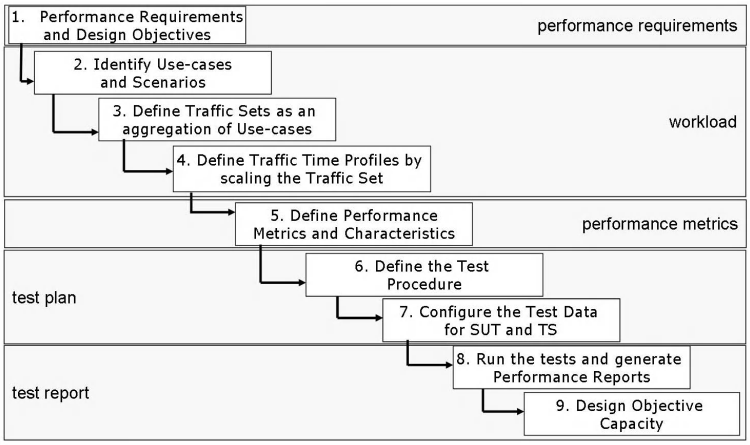

1. Performance Requirements and Design Objectives
performance requirements
2. Identify Use-cases and Scenarios
3. Define Traffic Sets as an aggregation of Use-cases
workload
4. Define Traffic Time Profiles by scaling the Traffic Set
5. Define Performance Metrics and Characteristics
performance metrics
6. Define the Test Procedure
test plan
7. Configure the Test Data for SUT and TS
8. Run the tests and generate Performance Reports
test report
9. Design Objective Capacity

Fig. 1. The Performance Test Design Process

test engineer has to identify for each scenario the sequence of actions the test system should interchange with the (System under Test) SUT, interaction protocols and input data sets. Since a multi-service system provides many services for each use-case, it is required that the workload is created as a composition of multiple test scenarios from each call model.

Performance Metrics. In a third step, the performance metrics have to be defined. Two types of metrics are regarded: global metrics (similar to other related works [11] e.g. CPU, MEM, fail rate, throughput etc) and scenario related metrics (defined for each scenario e.g. error rate, latency).

Performance Test Plan. The workloads are documented in a *performance test plan* which contains the technical documentation related to the execution of the performance test: which hardware is used to run the test, software versions, the test tools and the test schedule. The test cases and the test data are then implemented and executed in a selected performance test tool. Part of the test plan is also the *performance test procedure* which is specific to the selected type of performance test: volume, load, stress, benchmark etc.

Performance Test Report. A test report is a document, with accompanying data files, that provides a full description of an execution of a performance test. The test results should present the computed metrics and data sets in the form of charts, graph or other visual format.

4 Workload Elements

4.1 Use-Cases and Test Scenarios

The first step in the workload creation is to identify the *use-cases* and, for each use-case, select multiple *test scenarios*. A use-case is associated to one service and define interaction models between one or more users and the SUT (e.g. voice call, conference call).

An individual interaction path is called a *test scenario* and describes a possible interaction determined by the behaviour of the user and other system actors. The typical questions we have to answer at this point are: which services are going to be used most, which particular flows are characteristic to a given scenario, what is most likely to be an issue? Each test scenario is described by its message flow between the talking entities. A representative workload is realized when the selection of scenarios cover all possible situations: successful, fail, abandoned and rejected scenarios.

An example of a test scenario is depicted in Figure 2. This example presents the interaction between two User Endpoints (UE) and the SUT (all entities of the SUT are represented as a single box). The interaction is based on Request/Response transactions. Each response has to be received within a limit of time. This time is modeled as a timer which measures the time spend between the request and response. If the response is not received within the expected time, the transaction runs into a fail situation.

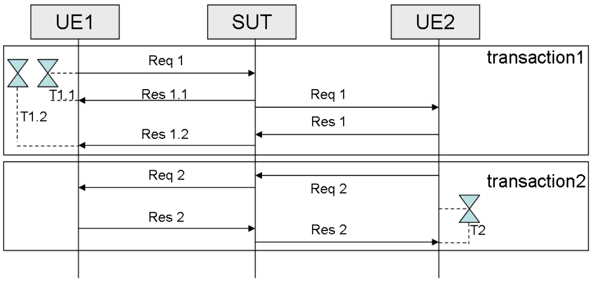


Fig. 2. Scenario Flow Example

4.2 Design Objectives

The performance requirements concern error-rates and delays. They are evaluated separately for each test scenario on top of the collected measurements. For each performance requirement a *design objective* (DO) is defined as a threshold value which is then used to compare the metrics. Examples of DOs are:

– *latency design objectives*: define the maximal time required to get a message through the SUT network from a caller UE1 to a callee UE2.
– *transaction round-trip time design objectives*: define the maximal time required to complete a transaction (including all messages).
– *error rates*: defines the threshold for allowed percentage of errors out of the number of instantiated scenarios.

4.3 Traffic Set Composition

The test system applies a workload to the SUT which consists of the traffic generated by a large number of individual simulated UEs. A conceptual question is how to allow testers define compositions of scenarios. The concept to cover this aspect is called *traffic set*.

Within the traffic set, each scenario has an associated relative occurrence frequency which is interpreted as its probability of occurring during the execution of the test. This frequency indicates how often a scenario should be instantiated during the complete execution of the performance test.

To avoid constant load intensities (in reality the load is random), the scenarios are instantiated according to an arrival distribution, which describes the arrival rate of occurrences of scenarios from the traffic set. The arrival rate characterizes the evolution of the average arrival rate as a function of time over the duration of the test procedure. An example of such an arrival process is the Poisson process [16] employed often in simulations of telecommunication traffic.

4.4 Traffic-Time Profile

A further concept is the *traffic-time profile* used to describe the workload intensity. The traffic-time profile defines the average *scenario attempt arrival rate* as

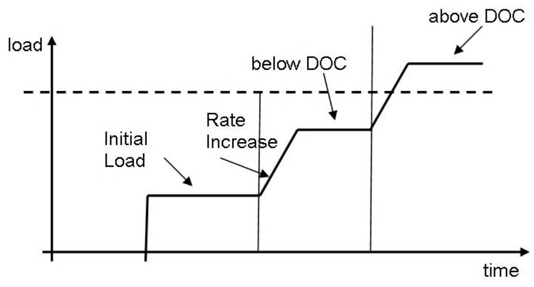


Fig. 3. Stair-Step Traffic-Time Profile

a function of elapsed time during a performance test. It should be defined in such a manner that, for a given scenario attempt arrival rate, sufficient samples are generated that metrics can be collected with an appropriate confidence bound.

A common traffic-time profile is the stair-step profile which is the one based on the *stairstep* shape (see Figure 3). The width of the stairstep is such chosen to collect sufficient samples at a constant average scenario arrival rate.

4.5 Scenario Based Performance Metrics

Based on the design objectives, performance metrics can be defined at scenario level.

Pass/Fail Metrics. The scenario attempts can be sorted into *passed* and *failed* scenarios. The number of failed versus passed scenarios out of the total attempts, is a metric which can be defined for each scenario. This metric is usually reported as a time based shape of fails or passes per second.

Transmission Latency. This metric is used to compute the average latency of the SUT at processing a certain event (*transmission latency*).

Round Trip Time. This metric is used to compute the average time needed to execute a transaction (*round-trip time*).

4.6 Global Performance Metrics

Besides the metrics computed per scenario, we also introduce global metrics to characterize the overall test execution. We classify them into: *resource usage metrics* and *troughput metrics*.

Resource usage metrics. These metrics capture the resource consumption of the SUT over time and may help identify *when* (i.e. at which load, after how much time) the SUT runs out of resources or starts failing.

Throughput metrics. The throughput metrics are the message rates and error rates. They are computed globally (or per use-case) for all scenarios. We propose a minimal set of metrics which any performance test should report.

- **SAPS.** Scenario Attempts Per Second (SAPS) metric represents the average rate at which scenarios are attempted.
- **SIMS.** SIMultaneous Scenarios (SIMS) counts the number of scenarios open in each second. The duration of one scenario varies from a few seconds to several minutes, therefore, SIMS metrics is a good indicator of how many open calls can handle the SUT.
- **%IHS.** The *Inadequately Handled Scenarios*(IHS) metric is the ratio of inadequately handled scenarios out of the total number of attempted scenarios. A scenario is considered inadequately handled when it does not respect the specified call flow.

4.7 Design Objective Capacity Definition

To be able to compare between different versions, hardware, products, etc., a *comparison criterion* is needed. In our methodology we define a global performance indicator called *Design Objective Capacity* (DOC) as the number which characterizes the overall performance of an SUT. This number is defined as the load rate for which the SUT still can handle the load for certain quality conditions. Increasing this load intensity would automatically affect the SUT to exceed the design objectives.

5 Performance Test Implementation Patterns

The test behavior is realized as a collection of parallel threads. Most operating systems provide features enabling a process to contain multiple threads of control [22]. The performance test systems use lots of threads at execution. There are threads specific to the testing purpose such as load generator, user state handlers but, within the execution platform, also threads dedicated to non-testing tasks (e.g. garbage collection thread in a java based platform) may coexist. A type of a thread may be instantiated for an arbitrary number of times. For each type of action, more than one threads can be instantiated in order to increase the parallelism.

The implementation patterns described in the following refer to the way how different types of threads are interacting with each other, how they share the data and how to manipulate the number of threads of different scopes for gaining the best performance.

5.1 Data Repository Representation

The interaction between a test behaviour thread and the user's data repository is depicted in Figure 4. The users' data repository consists of users' identity information and the list of active scenarios in which each user is involved. The thread simulates the behaviour of the user which consists, in that example, of the communication operations represented in the figure as *send* and *receive* operations. At any receive or send operation, the thread updates the state of the corresponding scenario. When a scenario finishes, the *scenario_id* is removed from the list of active scenarios and the user is made available for a new scenario.

A simple API to manage the user repository is presented in Listing 1. It consists of very simple operations to extract or update the information. Typical

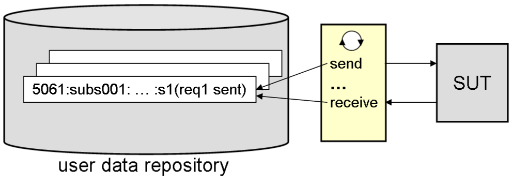


Fig. 4. User State Handling within a Thread

Listing 1. Repository Access API through external functions

```
external getAvailable() return userid;
external setState(userid u, state s);
external validateState(userid u, newstate ns) return boolean;
external makeAvailable(userid u);
```

operations are `getAvailable()` to get an available user to create a new call and `makeAvailable()` to release a user. For state management, `setState()` serves to set a new state while `validateState()` is meant to check whether a new state to get into is valid or not. However, the API can be extended to many needs but the principle remains the same.

5.2 User State Machine Design Patterns

The behaviour of a user is usually implemented as a state machine that stores the status of the user at a given time and can operate on input to change the status and/or cause an action or output to take place for any given change. From the implementation point of view, the user state machine can be realized either in a specific way or in a generic way. We discuss these two approaches.

Specific Handler (SM_SpecHdl). In this pattern the user state machine is processed entirely by one thread and all data associated to its state machine is stored locally. At receiving or sending of events, the user data is modified locally, the thread does not have to interact with an external repository. This avoids synchronization times with the rest of threads. As far as the implementation of this design is concerned, the whole logic concerns only one user. The behaviour does not have to perform complex checks to identify which user has to be updated for a given message. The timer events can be also simulated locally within the thread. Additionally, this pattern has the advantage that the state machine is implemented very efficiently since, at each state, only the valid choices are allowed. Everything unexpected is considered invalid.

Unfortunately, for large number of users, this pattern is not practical since many threads have to be created [4]. The most used operating systems (based on Unix and Windows) encounter all serious problems under conditions involving a tremendous number of threads.

In Listing 2, the specific handling mechanism is exemplified in the TTCN-3 language. The function gets a `userid` as start parameter to *personalize* it to a

Listing 2. Specific Event Handling

```
function user(id userid) runs on UserComponent {
 var stateType state := state1;
 var requestType req; var responseType resp;
 alt {
  [state == state1] p.receive(requestType1(userid))
        -> value req {
    if(match(req.field1, expectedValue) {
      state := state2;
      resp := responseTypeTemplate;
      resp.field2:=req.field2; resp.field3:=req.field3+1;
      p.send(resp);
      repeat;
    }
    else {state = failureState; makeAvailable(userid); stop;}
  }
  [state == state2] p.receive(requestType2(userid)) {
    state := finalState; makeAvailable(userid); stop; }
  [] p.receive {
    state = failureState; makeAvailable(userid); stop;}
 } //end alt
} // end function
```

particular user. As long as the function implements a specific user behaviour, the *state* variable (representing the state of the user) can be defined in the function as a local variable. The events are received by the port **p** and are handled by an **alt** block. Each event type is caught by a receive statement. The template must contain matching constraints for the `userid` given as parameter to the function. This is simple to achieve by parameterizing the template with the `userid`.

In the example, we define three possible branches. The first one defines the handle of an event of type `requestType1`. An event of this type can occur only when the user is in `state1`. This condition is specified in the guard of the branch. In the state handling, particular fields of the received message can be inspected by using the **match** function. If the received value matches a valid state for the user, the state is updated to the `state2` and a response message is prepared and sent back to the SUT. For a non-valid message, the test system has to set the state of the user to `failureState` and stop the component.

Generic Handler (SM_GenHdl). The *specific event handler* approach suffers of performance problems when too many threads are created. Therefore, a better solution is to create less threads by using one thread for more than one user. This way, the platform scales obviously better than in the previous approach.

This pattern requires a global data management and the state machine is rather a message processor. The functionality of this pattern is depicted in Figure 5. When new messages are received, the message processor identifies the user to which the new message belongs to and updates its status in the required way.

This pattern has the advantage that a handler can be used to process an arbitrary number of users in parallel. As long as the user data is stored outside the thread, the thread does not keep track of the flow of execution, instead, it executes its actions only when they are triggered by external events. Another advantage offered by this pattern is that the information of one user can be

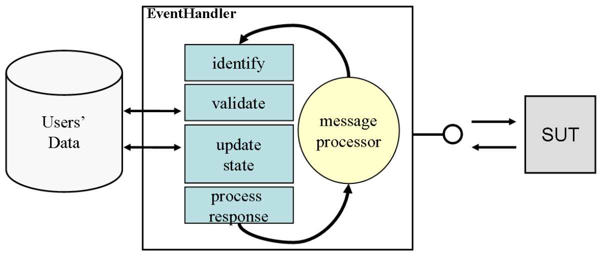


Fig. 5. Generic Event Handler

Listing 3. Generic Event Handling

```
function userHandler() runs on UserComponent {
 var requestType req; var responseType resp;
 alt {
  [] p.receive(requestType1) -> value req
  {
    if(not validState(req.userid, req)) {
      setState(req.userid, failureState);
      makeAvailable(req.userid); repeat;
    }
    if(match(req.field1, expectedValue)) {
      setState(req.userid, state2);
      resp := responseTypeTemplate;
      resp.field2 := req.field2; resp.field3 := req.field3 + 1;
      p.send(resp) to address addressOf(req.userid); repeat;
    }
    else { setState(req.userid, failureState);
      makeAvailable(userId); repeat;
    }
  }
  [] p.receive(requestType2) -> value req {
    if(not validState(req.userid, req)) {
      setState(req.userid, failureState);
      makeAvailable(req.userid); repeat;
    }
    setState(req.userid, finalState);
    makeAvailable(req.userid); repeat;
  }
  [] p.receive { setState(req.userid, failureState);
    makeAvailable(userid); repeat;
  }
 } // end alt
} // end function
```

managed by more than one thread. Since the thread does not manage the user information locally, it might be assigned to handle any arbitrary event for any user.

Listing 3 provides an example of a generic handler in TTCN-3. It basically rewrites the example from Listing 2 in a generic way. In the first example, the component has one port which is used for the communication with the SUT. In the second example, the same port is used for all users simulated by the component by using the **to address** statement. This concept optimizes the number of ports which require a lot of resources.

Next, instead of using the guard conditions, we introduce the **validState()** function to check the state of a user in a repository. The validation is realized upon the `userid` information extracted from the received message and the message itself. If the new state created by the received message is not a valid one, the test system considers the current call as *fail* and makes the user available for a new call. For a valid state, the behaviour looks the same as in the previous example. The state is updated by the **setState()** function.

Further on, the receive templates are now generic templates instead of parameterized templates per `userid`. These templates are not so restrictive as the ones used for the specific pattern but one can compensate by introducing more checks through **match** conditions.

5.3 Patterns for Thread Usage in User Handling

In the above section we discussed the two patterns to realize a user state machine. Both approaches have advantages and disadvantages. We present three more patterns regarding the thread design and usage. These patterns base on the fact that not all users have to be active at the same time. This avoids the existence of inactive threads, by instantiating new threads only when they are needed.

Single User Behaviour per Thread (UH_SingleUPT). The easiest way to implement a user is to create an instance of a thread simulating only that user, i.e. the *single user per thread* pattern. This pattern is suitable to the *state machine specific* handling approach (SM_SpecHdl) but can also be combined with the generic approach. However, the creation, the start and the termination of threads are very expensive operations with respect to CPU. Therefore, this pattern will is not suitable for an efficient design.

Sequential User Behaviours per Thread (UH_SeqUPT). This pattern considers the *reusability* of a thread for new users. This method requires that the user identities are stored outside the thread and are loaded into the thread only when the user has to become active. The threads are used like a pool of threads where each thread may take any user identity. During the test, the number of threads may be increased or decreased according to the load intensity. This way, the number of active users is maintained and controlled from within the running threads.

Interleaved User Behaviours per Thread (UH_InterleavedUPT). An even better approach to use threads is to interleave user behaviours at the same time on the same thread. This pattern may be seen as an extension of the previous thread with the addition that users are handled at the same time. In this way, the thread is able to simulate in parallel an arbitrary number of users.

5.4 Pattern for Timer Implementation

All protocols used nowadays in telecommunication include time constraint specifications for the maximal response times. Many protocols include also retransmissions specification at the protocol level. Additionally, the user interaction

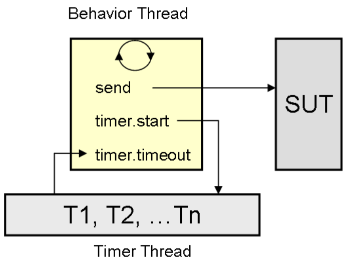


Fig. 6. Use of a Timer Thread for Timer realization

with the SUT involves various *user times* such as talking time or ringing time which in performance tests have to be simulated too. All these *timing specifications* have been regarded also in our test methodology. The test system has to validate if these time constraints are fulfilled.

In our approach, a timer thread manages all timers involved in test scenarios. When an event handler thread comes to the point that a timer has to be started, it asks the timer thread to create a new timer which will notice it back when it expires. The timer thread is provided with the user identity information and the expiration time. The timer thread manages a queue of timers and executes them in their temporal order. When a new timer is created, the timer thread schedules the new event in the right place. Since the timer events are ordered on time base, the timer thread has only to sleep until the next timeout.

When a new timeout occurs, the timer thread notices the events handler thread responsible for the user which created the timer. This can happen simply by sending a timeout event for that user. However, if the user receives in the meantime a valid response from the SUT, the timer thread has to remove the timeout event out of the scheduling queue.

This approach is compatible with all patterns previously described. It is very flexible since it allows the event handling threads to process events for an arbitrary number of users in parallel or even parallel calls of the same user. Moreover, the timeout events are handled as normal events, which makes the concept more generic. However, for even more flexibility, more than one timer thread can be used in parallel for different kinds of timers.

5.5 Patterns for Sending Messages

Sending and receiving operations are usually requiring long execution times due to data encoding, queuing and network communication. The threads which execute them block until the operation is completed. From this perspective, sometimes it is not convenient to let a *state machine handling thread* spend too much time for these operations. We propose a concept in which a separate thread is instantiated for the sending operations when these operations consume too much time. This pattern has the advantage that the main thread can work in parallel with the sending thread. We distinguish four different ways to realise this concept.

Thread per Request (S_SepThreadPerRequest). The *thread per request pattern* requires that each send request is handled by a separate thread of control. This pattern is useful for test systems that simulate multiple users that handle long-duration request/response (such as database queries). It is less useful for short-duration requests due to the overhead of creating a new thread for each request.

Thread per Session (S_SepThreadPerSession). This pattern is a variation of the *thread per request pattern* that compensates the cost of spawning the thread across multiple requests. This pattern handles each user simulated by the test system in a separate thread for the duration of the session. It is useful for test systems that simulate multiple users that carry on long-duration conversations.

Send Thread Pool Pattern (S_ThreadPool). A number of threads are created for all message sending tasks, usually organized in a queue. These threads are shared between multiple test behaviour threads. As soon as a thread completes its task, it will request the next task from the queue until all tasks have been completed. The thread can then terminate, or sleep until new tasks are available.

5.6 Patterns for Messages Receiving

Similar to sending of messages, there are several patterns to implement the receiving operations. For receiving of messages, the receive thread creates a message queue and reads from it whenever something is enqueued.

Thread per Session (R_SepThreadPerSession). In this pattern, a main thread deals with the main flow of actions (i.e. user behaviour) and another thread handles the events from SUT. The second thread only waits for SUT responses and notifies the main thread in case something is received.

Thread Pool for SUT Responses (R_ThreadPool). To simplify the thread management, test platforms organize their threads with thread pools. Since any

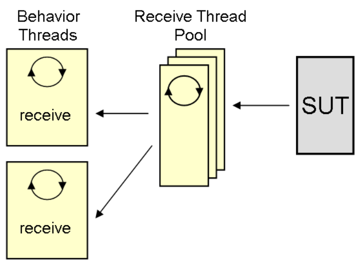


Fig. 7. Receive Thread Pool

main behaviour thread requires a second thread for checking SUT replies, we can share one such thread among several main threads. The shared thread listens to more sockets in parallel and whenever something is received it notifies the corresponding main thread.

This approach requires also an identification mechanism, usually called *event demultiplexor*, between the received messages and the main threads (see Figure 7). The event demultiplexor is an object that dispatches events from a limited number of sources to the appropriate event handlers. The developer registers interest in specific events and provides event handlers, or callbacks. The event demultiplexor delivers the requested events to the event handlers.

5.7 Data Encapsulation Patterns

The content of a message is usually not handled in its raw form but through a message structure which provides means to access its information. The mechanism is called *data encapsulation*. The content of a message is accessed via an interface which provide pointers to the smaller parts of a message. We discuss three strategies to encapsulate and access the content of a message.

String-Buffer Pattern (D_StringBuffer). The simplest method to encapsulate message data is to store it into a string buffer (therefore, this approach is limited to string based protocols e.g. SIP). The string buffer allows for easy search and modification operations through regular expressions. Due to its simplicity, this pattern is easy to integrate in any execution environment. Unfortunately, the string processing costs a lot of CPU time. Therefore, the pattern should not be used for big messages.

Structured Representation of Message Content (D_StrContent). This pattern requires that the content of the message is represented as a tree structure (based on the protocol message specification). The information is extracted by a decoder which traverses the whole message and constructs the tree representation. At *send* operation, the tree is transformed back into a raw-message by traversing each node of the tree. These operations may cost a lot of CPU time but the tree representation offers a lot of flexibility to easily access every piece of information.

Structured Representation of Pointers to Message Content (D_StrPointers). In this pattern, a tree structure contains pointers to the locations where the content can be accessed in the raw-message. This pattern also requires a codec but it can be implemented efficiently since the only task is to identify data at given locations in the raw-message. This pattern can be used even more efficiently when the locations are predefined e.g. by using fixed sizes for the fields' lenghts.

6 Case Study

The test patterns have been evaluated throughout a case study on IMS performance testing. IMS is a standardised architecture [1] to support a rich set of

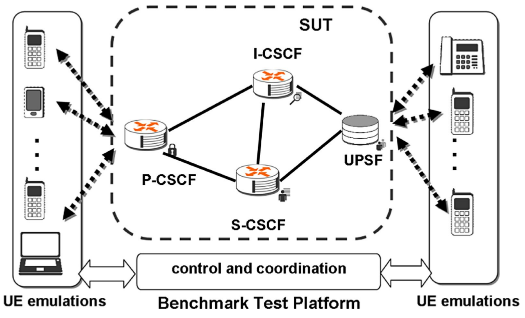


Fig. 8. Test System vs. System under Test

services and make them available to end users. The multitude of offered services and the complex call flows of the interactions between users and the IMS network, make IMS an excellent candidate to apply the methodology.

The SUT. The performance tests focus on the Session Control Subsytem (SCS) (see Figure 8). It consists of the three main (Call Session Control Function) CSCF elements: Proxy, Serving and Interrogating, and the (User Profile Server Functions) UPSF. The traffic set consists of scenarios which belong to the most common IMS use cases that are encountered the most in the real life deployments: *Registration and de-registration*, covering five scenarios, *Session set-up or tear-down*, covering twelve scenarios, *Page-mode messaging*, covering two scenarios.

The Test System Implementation. For the implementation of the test system we selected the TTCN-3 language. In TTCN-3, a thread corresponds to a *test component* and the thread functionality is implemented as a *function*. The load is generated by a specialised load generator component. Each call created by the load generator is associated to an `EventHandler` component, which handles all required transactions for that call. The number of `EventHandlers` is arbitrary and depends on the number of simulated users and on the performance of the hardware running the test system.

Along the case study development we improved step by step the performance of the test system by combining the patterns described in the paper. We illustrate these results in the following. Since the test system and the IMS SUT implementation were developed almost at the same time, the test system developers had to come with new solutions to increase the performance in order to compete the continuously improving performance of the SUT.

At the beginning, the test system was based on the *Specific Handler* design pattern (SM_SpecHdl) and on the Single User Behavior per Thread (UH_SingleUPT). The receiving of the messages was based on the *Thread per Session* (R_SepThreadPerSession) pattern while the data encapsulation was based on the structured representation of the message content (D_StrContent). This was the first draft of the implementation, the overall performance of

Table 1. Results comparison of different hardware configurations

Hardware Configuration	**DOC**
mem=4GB cpu=4x2.00GHz cache=512KB L2	180
mem=8GB cpu=4x2.00GHz cache=2MB L2	270
mem=8GB cpu=4x2.66GHz cache=4MB L2	450

the test system was not amazing, but it was a very easy adaptation from a functional test suite.

The first improvement was related to the mechanism used for receiving messages. We changed to a thread pool approach for handling the received messages (R_ThreadPool) and the performance of the test system increased up to 40%. The increase occurs in fact due to the reduction of the number of receiving threads.

Another milestone in the implementation was the switch from *Single User Behaviour per Thread* (UH_SingleUPT) to *Sequential User Behaviours per Thread* (UH_SeqUPT) and, later on, to the *Interleaved User Behaviours per Thread* pattern (UH_InterleavedUPT). That was necessary mainly because the SUT was capable of supporting more users and we needed to test the maximal capacity of the SUT. By using this approach the test system increased the number of emulated users from a couple of hundreds up to more than 10.000. Also the throughput performance increased again up to 30% since we create fewer threads then before.

In the early stages only 5 scenarios were supported, but in the end the traffic set consists of 20 different scenarios. This amount on different scenarios made that the Specific Handler pattern was not suitable anymore, thus we decided to adopt the Generic Handler pattern. The Generic Handler proved to be more flexible and easier to maintain and also more suitable to satisfy all the requirements of the workload (e.g. the mix of scenarios, the possibility of a user to handle more than one scenario at the same time).

A final improvement of the test solution is the switch from a structured representation of the message content (D_StrContent) to a structured representation of pointers to message content (D_StrPointers). From the preliminary experiments we expect a remarkable increase of the performance to more than 300% out of the current performance.

Experimental Work. The performance test has been applied to different hardware configurations in order to illustrate how the the test can be used for performance comparison. The implementation for the IMS architecture used as SUT within the case study is OpenIMSCore [8], the open source implementation of Fraunhofer FOKUS.

Table 1 presents a comparison of the DOC numbers for different servers. The SUT software has been installed with the same configuration on all servers. The DOC is obviously higher for the last two servers which have more memory. However, the cache seems to have the biggest impact since the last board (with the highest DOC) has similar configuration as the second server but more cache.

The experiments revealed that many parameters have a great impact on the SUT performance. Among these parameters are: traffic set composition, number of users, duration of the test. Obiviously, changing the traffic composition will automatically change the demand of resources on SUT side. The duration is actually influenced by the number of total calls created along the test. For different loads but same duration, the test system creates different numbers of calls. With respect to the number of users, another experiment comparing the SUT performance with 5000 users respectively 10000 users, shown that the SUT performance is 40% worse when running the test with 10000 users.

7 Conclusions

The performance testing of continuously evolving services is becoming a real challenge due to the estimated increase of the number of subscribers and services demand. This paper presented a methodology for performance testing of telecommunication systems. The topic embraces the challenges of the nowadays telecommunication technologies. The main outcome is the method to create workloads for performance testing of such systems. It takes into account many factors: use cases, call flows, test procedures, metrics and performance evaluation. The design method ensures that the workloads are realistic and simulate the conditions expected to occur in real-live usage.

Within the case study, we designed and developed a performance test suite capable of evaluating the performance of IMS networks for realistic workloads. The TTCN-3 language has been selected to allow a more concrete illustration of the presented concepts. Along implementation of the test framework we pursuit the efficiency related aspect which determined us to experiment with various patterns to design and implement performance test systems.

References

1. 3GPP. Technical Specification Group Services and System Aspects, IP Multimedia Subsystem (IMS), Stage 2, V5.15.0, TS 23.228 (2006)
2. Amaranth, P.: A Tcl-based multithreaded test harness. In: Proceedings of the 6th conference on Annual Tcl/Tk Workshop, San Diego, California, vol. 6, p. 8. USENIX Association (1998)
3. Nixon, B.A., Nixon, B.A.: Management of performance requirements for information systems. Transactions on Software Engineering 26, 1122–1146 (2000)
4. Born, M., Hoffmann, A., Schieferdecker, I., Vassiliou-Gioles, T., Winkler, M.: Performance testing of a TINA platform. In: Telecommunications Information Networking Architecture Conference Proceedings, TINA 1999, pp. 273–278 (1999)
5. Camarillo, G., Garcia-Martin, M.A.: The 3G IP Multimedia Subsystem (IMS). Merging the Internet and the Cellular Worlds, 2nd edn. Wiley & Sons, Chichester (2005)
6. Empirix. e-Load (2007), `http://www.scl.com/products/empirix/testing/e-load`
7. Empirix. Hammer (2007), `http://www.empirix.com/`

8. Fraunhofer FOKUS. FOKUS Open Source IMS Core (2006)
9. Gao, J.Z., Tsao, H.-S.J., Wu, Y.: Testing and Quality Assurance for Component-Based Software. Artech House Publishers (August 2003) ISBN 1580534805
10. Grabowski, J., Hogrefe, D., Rethy, G., Schieferdecker, I., Wiles, A., Willcock, C.: An introduction to the testing and test control notation (TTCN-3). Computer Networks 42, 375–403 (2003)
11. Jain, R.K.: The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. Wiley, Chichester (1991)
12. Keskin, A.C., Patzschke, T.I., von Voigt, E.: Tcl/Tk: a strong basis for complex load testing systems. In: Proceedings of the 7th conference on USENIX Tcl/Tk Conference, Austin, Texas, vol. 7, p. 6. USENIX Association (2000)
13. McDysan, D., Paw, D.: ATM & MPLS Theory & Applications: Foundations of Multi-service Networking. Mcgraw-Hill Professional, New York (2002)
14. Menascé, D.A., Almeida, V.A.F., Fonseca, R., Mendes, M.A.: A methodology for workload characterization of e-commerce sites. In: EC 1999. Proceedings of the 1st ACM conference on Electronic commerce, pp. 119–128. ACM, New York (1999)
15. Mercury. Mercury LoadRunner (2007), `http://www.mercury.com/`
16. NIST/SEMATECH. e-Handbook of Statistical Methods (2006)
17. Rupp, S., Banet, F.-J., Aladros, R.-L., Siegmund, G.: Flexible Universal Networks - A New Approach to Telecommunication Services? Wirel. Pers. Commun. 29(1-2), 47–61 (2004)
18. Shirodkar, S.S., Apte, V.: AutoPerf: an automated load generator and performance measurement tool for multi-tier software systems. In: Proceedings of the 16th international conference on World Wide Web, Banff, Alberta, Canada, pp. 1291–1292. ACM Press, New York (2007)
19. Stankovic, N.: Patterns and tools for performance testing. In: Electro/information Technology, 2006 IEEE International Conference, pp. 152–157 (2006)
20. Stankovic, N., Zhang, K.: A distributed parallel programming framework. IEEE Trans. Softw. Eng. 28, 478–493 (2002)
21. SUN. Faban (2007), `http://faban.sunsource.net/`
22. Tanenbaum, A.: Modern Operating Systems, 2nd edn. Prentice-Hall, Englewood Cliffs (2001)
23. Wang, Y., Rutherford, M.J., Carzaniga, A., Wolf, A.L.: Automating experimentation on distributed testbeds. In: ASE 2005. Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, pp. 164–173. ACM, New York (2005)
24. Welch, B., Jones, K.: Practical Programming in Tcl and Tk, 4th edn. Prentice Hall PTR, Englewood Cliffs (2003)

Generating Checking Sequences for Partial Reduced Finite State Machines

Adenilso Simão[1] and Alexandre Petrenko[2]

[1] Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo
São Carlos, São Paulo, Brazil
adenilso@icmc.usp.br
[2] Centre de recherche informatique de Montreal (CRIM)
Montreal, Quebec, Canada
petrenko@crim.ca

Abstract. The problem of generating checking sequences for FSMs with distinguishing sequence has been attracting interest of researchers for several decades. In this paper, a solution is proposed for partial reduced FSMs with distinguishing sets, and either with or without reset feature. Sufficient conditions for a sequence to be a checking sequence for such FSMs are formulated. Based on these conditions, a method to generate checking sequence is elaborated. The results of an experimental comparison indicate that the proposed method produces shorter checking sequences than existing methods in most cases. The impact of using the reset feature on the length of checking sequence is also experimentally evaluated.

1 Introduction

Test generation from a Finite State Machine (FSM) is an active research topic with numerous contributions over decades, starting with the seminal work of Moore [9] and Hennie [5]. Given a specification FSM *M* and a black box implementation *N*, the objective is to construct a test suite that can check whether *N* behaves correctly according to *M*. It is usual to assume that *N* behaves like an unknown FSM with the same input alphabet and has at most as many states as *M*.

A checking sequence is an input sequence that can be used to check the correctness of the implementation. Many methods for generating checking sequences have been proposed, e.g., [5], [8], [4], [2], [11], [6], [3], [10], [7], and [12]. The crucial issue for these methods is how to guarantee that a black box implementation is in a known state after the application of some input sequence. This problem is somewhat simplified if the specification FSM *M* has a distinguishing sequence, that is an input sequence for which different states of *M* produce different outputs. However, not every FSM has a distinguishing sequence. A distinguishing set is a set of input sequences, one for each state of *M*, such that for each pair of distinct states, the respective input sequences have a common prefix for which both states produce different outputs. A distinguishing set can be obtained from a distinguishing sequence. Notice, however, that there exist FSMs with a distinguishing set which do not have distinguishing sequence [2].

Several generation methods have been proposed for generating checking sequence when a distinguishing sequence is available, e.g., [5], [4], [11], [6], [3], [10], [7], and [12]. In [5], Hennie elaborates the basis for the existing generation methods. Hennie divides the checking sequence into two parts: in the first part, the distinguishing sequence and some (possibly empty) transfer sequence are applied to each state, verifying that the distinguishing sequence is also a distinguishing sequence for the implementation under test, while in the second part, each transition is verified certifying that the source and target states are correctly implemented. A graph-theoretical method for generation of checking sequence is presented in [4]. No attempt to optimize the checking sequence is made, though. Recently, interest in improving methods for generating checking sequence with distinguishing sequence has revived again. Ural *et al.* [11] state an important theorem with sufficient conditions for a sequence to be a checking sequence for a complete FSM with a distinguishing sequence. A graph-theoretical method is also suggested, which casts the problem of finding a checking sequence as a Rural Chinese Postman Problem, following Aho *et al.*'s work [1]. This work has later been improved in [6] and [7], which fine-tune the modeling of the problem; Chen *et al.* [3] demonstrate that it is sufficient to consider only a subset of the FSM transitions; and Ural and Zhang [10] use the overlapping of the distinguishing sequence with itself to shorten the checking sequence.

Boute [2] shows how to generate a checking sequence for FSMs which may not have distinguishing sequences, but have distinguishing sets. The method also determines when transitions are "automatically" verified (i.e., when the verification of a transition is a consequence of the verification of other transitions) similarly to a more recent work [3].

All the above methods deal only with complete FSMs. Moreover, when the implementation under test has the reset feature, these methods do not attempt to use the reset input to shorten the checking sequence.

The contributions of this paper are twofold. First, we claim a theorem that states sufficient conditions for a sequence to be checking sequence for a (possibly partial) FSM with a distinguishing set. Notice that this theorem generalizes Ural *et al.*'s theorem [11]. The second contribution is a constructive method that generates checking sequence from FSM with a distinguishing set. Differently from recent works (namely, [11], [6], [3], [7], and [10]), the proposed method does not attempt to make a global optimization. Instead, it makes a local best choice in each step. Although the global optimization is expected to lead to shorter checking sequence, the graph-theoretical methods require that some choices be made *a priori* which may reduce the effectiveness of global optimization to that of local optimization if not lower. For instance, the method of Hierons and Ural [7] requires that a set of transfer sequences to be defined, and, moreover, the so-called α-set is determined by a separate algorithm, which may influence the effectiveness of the method. We present experimental results which indicate that the proposed method based only on local optimization performs better than existing methods in most cases.

This paper is organized as follows. In Section 2, we provide the necessary basic definitions. In Section 3, we define the notion of confirmed sets and use it to state sufficient conditions for an input sequence to be a checking sequence. A method for generating checking sequences based on the proposed conditions is presented in

Sections 4. In Section 5, we present an experimental evaluation of the method. We discuss the related work in Section 6. Section 7 concludes the paper.

2 Definitions

A Finite State Machine is a deterministic Mealy machine, which is defined as follows.

Definition 1. *A* Finite State Machine (FSM) *M is a 6-tuple* $(S, s_0, I, O, D_M, \delta, \lambda)$, *where*

- *S is a finite set of states with the initial state s_0,*
- *I is a finite set of inputs,*
- *O is a finite set of outputs,*
- *$D_M \subseteq S \times I$ is a specification domain,*
- *$\delta : D_M \rightarrow S$ is a transition function, and*
- *$\lambda : D_M \rightarrow O$ is an output function.*

If $D_M = S \times I$, then M is a *complete* FSM; otherwise, it is a *partial* FSM. A tuple $(s, x) \in D_M$ is a (*defined*) *transition* of M. A string $\alpha = x_1 \ldots x_k$, $\alpha \in I^*$, is said to be a *defined* input sequence at state $s \in S$, if there exist $s_1, \ldots, s_{k+1}$, where $s_1 = s$, such that $(s_i, x_i) \in D_M$ and $\delta(s_i, x_i) = s_{i+1}$, for all $1 \leq i \leq k$. We use $\Omega(s)$ to denote the set of all defined input sequences for state s and Ω_M as a shorthand for $\Omega(s_0)$, i.e., for the input sequences defined for the initial state of M and, hence, for M itself.

Given sequences $\alpha, \varphi, \beta \in I^*$, if $\beta = \alpha\varphi$, then α is a *prefix* of β, denoted by $\alpha \leq \beta$, and φ is *suffix* of β. For a sequence $\beta \in I^*$, *pref*(β) is the set of prefixes of β, i.e., *pref*$(\beta) = \{\alpha \mid \alpha \leq \beta\}$. For a set of sequences T, *pref*(T) is the union of *pref*(β), for all $\beta \in T$.

We extend the transition and output functions from input symbols to defined input sequences, including the empty sequence ε, as usual, assuming $\delta(s, \varepsilon) = s$ and $\lambda(s, \varepsilon) = \varepsilon$, for $s \in S$, and for each $\alpha x \in \Omega(s)$, $\delta(s, \alpha x) = \delta(\delta(s, \alpha), x)$ and $\lambda(s, \alpha x) = \lambda(s, \alpha)\lambda(\delta(s, \alpha), x)$. Moreover, we extend the transition function to sets of defined input sequences. Given an FSM M, a set of input sequences $C \subseteq \Omega(s)$, $s \in S$, we define $\delta(s, C)$ to be the set of states reached by the sequences in C, i.e., $\delta(s, C) = \{\delta(s, \alpha) \mid \alpha \in C\}$. For simplicity, we slightly abuse the notation and write $\delta(s, C) = s'$, whenever $\delta(s, C) = \{s'\}$. Let also $\Phi(C, s) = \{\alpha \in C \mid \delta(s_0, \alpha) = s\}$, i.e., $\Phi(C, s)$ is the subset of sequences of C which lead M from the initial state to s, if any, thus containing the sequences of A converging on state s.

An FSM M is said to be *strongly connected*, if for each two states $s, s' \in S$, there exists an input sequence $\alpha \in \Omega(s)$, called a *transfer* sequence from state s to state s', such that $\delta(s, \alpha) = s'$.

Two states $s, s' \in S$ are *distinguishable*, denoted $s \nsim s'$, if there exists $\gamma \in \Omega(s) \cap \Omega(s')$, such that $\lambda(s, \gamma) \neq \lambda(s', \gamma)$. We also use the notation $s \nsim_\gamma s'$ when we need to refer to a sequence distinguishing states. If a sequence γ distinguishes each pair of distinct states, then γ is a *distinguishing* sequence. If γ distinguishes a state s from every other state, then γ is an *identification* sequence for state s. A distinguishing sequence is an identification sequence for each state, however, the converse does not hold. A *distinguishing set* Ξ is a set of $|S|$ identification sequences, such that for each pair of distinct states $s, s' \in S$, there exists a sequence distinguishing s and s' which is a common prefix of the respective identification sequences. Notice that, given a distinguishing sequence E, the set of the shortest prefixes of E, each of which is an identification sequence of a state, is a distinguishing set. Moreover, for a given FSM, there may exist a distinguishing set even if no distinguishing sequence exists [2]. Given a set $C \subseteq \Omega(s) \cap \Omega(s')$, states s and s' are *C-equivalent*, denoted $s \sim_C s'$, if $\lambda(s, \gamma) = \lambda(s', \gamma)$ for all $\gamma \in C$. We define distinguishability and C-equivalence of machines as a corresponding relation between their initial states. An FSM M is said to be *reduced*, if all states are pairwise *distinguishable*, i.e., for all $s, s' \in S$, $s \neq s'$ implies $s \nsim s'$. An FSM N is *quasi-equivalent* to M, if $\Omega_M \subseteq \Omega_N$ and N is Ω_M-equivalent to M.

Given a reduced FSM M, let $\Im(M)$ be the set of all reduced complete FSMs with the input alphabet of M and at most n states, where n is the number of states of M.

Definition 2. *A finite input sequence $\omega \in \Omega_M$ of FSM M is a* checking sequence (for M), *if for each FSM $N \in \Im(M)$, such that $N \nsim M$, it holds that $N \nsim_\omega M$.*

Let $N = (Q, q_0, I, O', D_N, \Delta, \Lambda)$ be an arbitrary element of $\Im(M)$. Given an input sequence α, let $\Im_\alpha(M)$ be the set of all $N \in \Im(M)$, such that N and M are α-equivalent. Thus, ω is a checking sequence for M if every $N \in \Im_\omega(M)$ is quasi-equivalent to M.

A finite set $K \subseteq \Omega_M$ is a *state cover* for M if $\delta(s_0, K) = S$. A state cover K is *minimal* if $|K| = |S|$.

3 Generalizing Sufficient Conditions for Checking Sequences

Constructing checking sequence, a crucial issue is how to guarantee that the black box implementation is in a known state after the application of some input sequence. We propose a new way of addressing this issue by introducing the notion of confirmed sets of defined input sequences. In particular, a set of prefixes of an input sequence is confirmed if and only if it has transfer sequences for each state of the specification FSM M and any sequences that converge, i.e., lead to a same state (diverge, lead to different states) in any FSM that has the same output response to the given input sequence and has as many states as M if and only if they converge (diverge) in M.

Definition 3. *Let ω be a defined input sequence of an initially connected reduced FSM $M = (S, s_0, I, O, D_M, \delta, \lambda)$ and $K \subseteq pref(\omega)$. The set K is $\Im_\omega(M)$*-confirmed (*or simply* confirmed) *if K is a state cover and, for each $N \in \Im_\omega(M)$, it holds that for all $\alpha, \beta \in K$, $\Delta(q_0, \alpha) = \Delta(q_0, \beta)$ if and only if $\delta(s_0, \alpha) = \delta(s_0, \beta)$. An input sequence is* confirmed *if there exists a confirmed set that contains it.*

In this paper, we assume that the FSM M is strongly connected, reduced, and has a distinguishing set Ξ. Given Ξ, $\omega \in \Omega_M$ and $\alpha \leq \omega$, α is said to be Ξ-*recognized* in ω, if $\alpha E_s \leq \omega$,where $E_s \in \Xi$ and $\delta(s_0, \alpha) = s$.

Lemma 1. *Let $\omega \in \Omega_M$ and K be a minimal state cover. If each $\alpha \in K$ is Ξ-recognized in ω, then K is $\Im_\omega(M)$-confirmed.*

Proof. Let $N \in \Im_\omega(M)$ and $\alpha, \beta \in K$. We demonstrate that $\delta(s_0, \alpha) \neq \delta(s_0, \beta)$ implies $\Delta(q_0, \alpha) \neq \Delta(q_0, \beta)$. Suppose that $s = \delta(s_0, \alpha) \neq \delta(s_0, \beta) = s'$. Notice that α and β are followed in ω by E_s and $E_{s'}$, respectively. Thus, there exists a sequence $\gamma \in pref(E_s) \cap pref(E_{s'})$, such that $\lambda(s, \gamma) \neq \lambda(s', \gamma)$. As N is ω-equivalent to M, it follows that $\Lambda(\Delta(q_0, \alpha), \gamma) = \lambda(s, \gamma) \neq \lambda(s', \gamma) = \Lambda(\Delta(q_0, \beta), \gamma)$. Therefore, $\Delta(q_0, \alpha) \neq \Delta(q_0, \beta)$. ♦

From Lemma 1, we state the following corollary.

Corollary 1. *Let $\omega \in \Omega_M$ and K be a minimal state cover and $\Im_\omega(M)$-confirmed. Then K is a minimal state cover for any $N \in \Im_\omega(M)$.*

The next lemma indicates when a state cover K is confirmed, even if it is not minimal.

Lemma 2. *Let $\omega \in \Omega_M$ and K be a state cover. If each $\alpha \in K$ is Ξ-recognized in ω, then K is $\Im_\omega(M)$-confirmed.*

Proof. Let $N \in \Im_\omega(M)$. Let $\alpha, \beta \in K$. If $\delta(s_0, \alpha) \neq \delta(s_0, \beta)$, we can use the same argument used in the proof of Lemma 1 to prove that $\Delta(q_0, \alpha) \neq \Delta(q_0, \beta)$. Suppose then that $\delta(s_0, \alpha) = \delta(s_0, \beta)$. Let $K' \subseteq K$ be such that both $K_\alpha = K' \cup \{\alpha\}$ and $K_\beta = K' \cup \{\beta\}$ are minimal state covers for M. By Corollary 1, we have that K_α and K_β are also minimal state covers for N. Thus, $\Delta(q_0, \alpha)$ is distinct from each of the $n - 1$ states in $\Delta(q_0, K')$. As $\Delta(q_0, \beta)$ is also distinct from each of the $n - 1$ states in $\Delta(q_0, K')$ and N has n states, it follows that $\Delta(q_0, \alpha) = \Delta(q_0, \beta)$. ♦

The next statement relies on the fact that if proper prefixes of some transfer sequences converge in a deterministic FSM, then the sequences converge as well. We use the following definitions. If α and $\alpha\varphi$ are confirmed sequences (in a confirmed set K), then φ is *verified* in $\delta(s_0, \alpha)$ (w.r.t. to K). If $x \in I$ is verified in s, then the transition (s, x) is *verified.*

Lemma 3. *Let K be a $\Im_\omega(M)$-confirmed set. If $\alpha, \beta \in K$, $\delta(s_0, \alpha) = \delta(s_0, \beta)$, and φ is verified in $\delta(s_0, \alpha)$, then the set $K \cup \{\beta\varphi\}$ is also $\Im_\omega(M)$-confirmed.*

Proof. As α and β are confirmed in K and φ is verified in $\delta(s_0, \alpha)$, $\alpha\varphi$ is also confirmed. Thus, we have that $\Delta(q_0, \alpha) = \Delta(q_0, \beta)$ and, therefore, it follows that $\Delta(q_0, \beta\varphi) = \Delta(\Delta(q_0, \beta), \varphi) = \Delta(\Delta(q_0, \alpha), \varphi) = \Delta(q_0, \alpha\varphi)$. ♦

Thus, each sequence that is Ξ-recognized or is concatenation of Ξ-recognized and verified sequences can be included into a confirmed set. This key property of confirmed sets suggests a method for constructing a checking sequence which we elaborate later in the paper.

Theorem 1. *Let ω be an input sequence of a reduced FSM $M = (S, s_0, I, O, D_M, \delta, \lambda)$ with n states. ω is a checking sequence for M, if there exists a confirmed set K with the following properties*:

1. $\varepsilon \in K$.
2. *Each defined transition is verified.*

Proof. Let $N \in \Im_\omega(M)$. As M is strongly connected, for each $s \in S$, there exists $\alpha \in K$, such that $\delta(s_0, \alpha) = s$. For each $\beta \in K$, if $\delta(s_0, \beta) \neq \delta(s_0, \alpha)$, then $\Delta(q_0, \beta) \neq \Delta(q_0, \alpha)$.

Thus, $|Q| = n$. Consequently, there exists a bijection $f : S \to Q$, such that for each $\alpha \in K$, $f(\delta(s_0, \alpha)) = \Delta(q_0, \alpha)$. As $\varepsilon \in K$, $f(s_0) = q_0$.

We prove that, for each $\nu \in \Omega_M$, $f(\delta(s_0, \nu)) = \Delta(q_0, \nu)$ using induction on ν, and, moreover, $\lambda(s, x) = \Lambda(f(s), x)$ for each $(s, x) \in D_M$.

If $\nu = \varepsilon$, we have $\nu \in K$, and, by definition, $f(\delta(s_0, \nu)) = \Delta(q_0, \nu)$. Let $\nu = \varphi x$ and assume that $f(\delta(s_0, \varphi)) = \Delta(q_0, \varphi)$. As the transition $(\delta(s_0, \varphi), x)$ is verified, there exist α, $\alpha x \in K$, such that $\delta(s_0, \alpha) = \delta(s_0, \varphi)$. Thus, we have that $\Delta(q_0, \alpha) = f(\delta(s_0, \alpha)) = f(\delta(s_0, \varphi)) = \Delta(q_0, \varphi)$ and $f(\delta(s_0, \alpha x)) = \Delta(q_0, \alpha x)$. It follows that $f(\delta(s_0, \varphi x)) = f(\delta(\delta(s_0, \varphi), x)) = f(\delta(\delta(s_0, \alpha), x)) = f(\delta(s_0, \alpha x)) = \Delta(q_0, \alpha x) = \Delta(\Delta(q_0, \alpha), x) = \Delta(\Delta(q_0, \varphi), x) = \Delta(q_0, \varphi x)$. Therefore, $f(\delta(s_0, \varphi x)) = \Delta(q_0, \varphi x)$ and, by induction, for any $\nu \in \Omega_M$, $f(\delta(s_0, \nu)) = \Delta(q_0, \nu)$.

For each transition $(s, x) \in D_M$, there exists $\alpha x \in pref(\omega)$, $\delta(s_0, \alpha) = s$, $\alpha \in K$. Therefore, $\lambda(\delta(s_0, \alpha), x) = \Lambda(\Delta(q_0, \alpha), x)$. As $\alpha \in K$, we have that $\Delta(q_0, \alpha) = f(s)$ and, as N is ω-equivalent to M, it follows that $\lambda(s, x) = \Lambda(f(s), x)$.

Suppose finally that N can be distinguished from M. Therefore, there exists a sequence $\nu x \in \Omega_M$, such that $\lambda(s_0, \nu) = \Lambda(q_0, \nu)$ and $\lambda(s_0, \nu x) \neq \Delta(q_0, \nu x)$. There exist $\alpha \in K$, such that $\delta(s_0, \alpha) = \delta(s_0, \nu)$, and $\alpha x \in pref(T)$, such that $\lambda(\delta(s_0, \alpha), x) = \Lambda(f(\delta(s_0, \alpha)), x)$. $\delta(s_0, \alpha) = \delta(s_0, \nu)$ implies that $f(\delta(s_0, \alpha)) = f(\delta(s_0, \nu))$. Thus, $\lambda(\delta(s_0, \nu), x) = \Lambda(f(\delta(s_0, \nu)), x)$; and from $\lambda(s_0, \nu) = \Lambda(q_0, \nu)$, it follows that $\lambda(s_0, \nu x) = \Lambda(q_0, \nu x)$. The resulting contradiction concludes the proof. ♦

The theorem presented in [11] is a special case of Theorem 1. While Ural *et al.*'s theorem is applicable to complete FSMs with distinguishing sequence, Theorem 1 is applicable to partial FSMs as well as to FSMs with distinguishing sets. In the following section, we discuss how the sufficient conditions can be used to elaborate a method that generates checking sequence for FSMs with distinguishing sets. In Section 5, we present experimental results which demonstrate that the proposed method produces shorter checking sequences than known methods in most cases, even for complete FSMs with distinguishing sequences.

4 Checking Sequence Generation Method Based on Distinguishing Sets

In this section, we present a method based on distinguishing sets which exploits overlapping between identification sequences shortening the length of a checking sequence. The basic idea of the method is to consecutively append identification and transfer sequences to a current sequence ω, until a confirmed set $K \subseteq pref(\omega)$ is obtained and each transition of M is verified in K. We use $R(\omega)$ to denote the maximal subset of prefixes of ω, such that each $\alpha \in R(\omega)$ is either Ξ-recognized or there exist β, $\beta\varphi$, $\chi \in R(\omega)$, such that $\delta(s_0, \beta) = \delta(s_0, \chi)$ and $\alpha = \chi\varphi$ (Lemma 3). Notice that, if $R(\omega)$ is a state cover, by Lemma 1 and 2, $R(\omega)$ is a confirmed set. Notice also that if $\alpha \leq \omega$, then $R(\alpha) \subseteq R(\omega)$. Thus, the method obtains a checking sequence by guaranteeing that $R(\omega)$ is a confirmed state cover and that each transition is verified in $R(\omega)$. By $V(\omega)$ we denote the set of transitions verified in $R(\omega)$. Let also $U(\omega) = D_M \setminus V(\omega)$ be the set of unverified transitions.

The method is described in Algorithms 1 and 2 presented below. Let ω_i be a sequence obtained in the i-th iteration of Algorithm 1. There exist two cases that are dealt with by the algorithm. The first case occurs when $\omega_i \notin R(\omega_i)$. Then, we identify the longest suffix χ of ω_i, such that $\alpha_i\chi = \omega_i$ and χ is also a prefix of the identification sequence of $s = \delta(s_0, \alpha_i)$, and append E_s to α_i. Actually, as α_i is already followed by χ in ω_i, the suffix φ of E_s, with $\chi\varphi = E_s$, is appended to ω_i to obtain ω_{i+1}. Notice that $\alpha_i \in R(\omega_{i+1})$, since α_i is Ξ-recognized. After doing this a certain number of times, we have that $\omega_i \in R(\omega_i)$ (see Lemma 4 below), which is the second case. In this case, we verify a yet unverified transition (w.r.t. to $R(\omega_i)$). Notice that, as $\omega_i \in R(\omega_i)$, if α is a verified transfer sequence from $\delta(s_0, \omega_i)$ to some state s, we have that $\omega_i\alpha \in R(\omega_i\alpha)$. We then append $xE_{s'}$, for $s' = \delta(s, x)$, to $\omega_i\alpha$, so that $\omega_i\alpha x \in R(\omega_i\alpha xE_{s'})$, i.e., the transition (s, x) is verified w.r.t. $R(\omega_i\alpha xE_{s'})$. Algorithm 1 terminates when no transition remains unverified. Notice that, the determination of $R(\omega_{i+1})$ and $U(\omega_{i+1})$ from a given intermediate sequence ω_i is a key feature of the algorithm. We provide an efficient method for doing this in Algorithm 2.

Algorithm 1.

```
Input: A distinguishing set $\Xi$ for a reduced FSM $M = (S, s_0, I, O, D_M, \delta, \lambda)$.
Output: A checking sequence $\omega$
$i \leftarrow 0$
$\omega_0 \leftarrow \varepsilon$
$U(\omega_0) \leftarrow D_M$
$R(\omega_0) \leftarrow \varnothing$
while $U(\omega_i) \neq \varnothing$ do
    Step 1. if $\omega_i \notin R(\omega_i)$, then
        o let $\alpha_i \in pref(\omega_i)$ be the shortest prefix of $\omega_i$, such that $\alpha_i \notin R(\omega_i)$, $\omega_i = \alpha_i\chi$, $s = \delta(s_0, \alpha_i)$, $E_s = \chi\varphi$.
        o Update $\omega_{i+1} \leftarrow \omega_i\varphi$.
        o Determine $R(\omega_{i+1})$ and $U(\omega_{i+1})$ using Algorithm 2 with the input $\omega_{i+1}$ and $\alpha_i$.
    Step 2. else,
        o determine a shortest verified transfer sequence $\beta_i$ from state $\delta(s_0, \omega_i)$ to some state $s$, such that there exists $x \in I$ and $(s, x) \in U(\omega_i)$.
        o Let $\alpha_i = \omega_i\beta_i$ and $s' = \delta(s_0, \alpha_i x)$.
        o Update $\omega_{i+1} \leftarrow \alpha_i xE_{s'}$
        o Determine $R(\omega_{i+1})$ and $U(\omega_{i+1})$ using Algorithm 2 with the input $\omega_{i+1}$ and $\alpha_i x$.
    end if
    $i \leftarrow i + 1$
end while
Return $\omega \leftarrow \omega_i$
```

Now, we present an algorithm to calculate $R(\omega_{i+1})$ and $U(\omega_{i+1})$. Actually, these sets can be determined directly from their definitions. A straightforward method to find $R(\omega_{i+1})$ would require the inspection of all subsequences of ω_{i+1}. Notice, however, that it is sufficient to determine the set of verified sequences, since $R(\omega_{i+1})$ and $U(\omega_{i+1})$ can be derived from them. Suppose that the sequences β, $\beta\varphi$ and $\beta\varphi\chi$ are in $R(\omega_{i+1})$. Then, the sequences $\varphi\chi$ and φ are verified in $\delta(s_0, \beta)$ and χ is verified in $\delta(s_0, \beta\varphi)$. The fact that $\varphi\chi$ is verified in $\delta(s_0, \beta)$ is not used to determine $R(\omega_{i+1})$ and $U(\omega_{i+1})$, since the same result is obtained from the fact that the other two sequences are verified. This observation suggests that only shortest verified sequences have to be considered for a given state. We denote by $P(\omega_{i+1})$ the maximal subset of $S \times (I^*\backslash\{\varepsilon\})$, such that $(s, \alpha) \in P(\omega_{i+1})$ iff α is the shortest sequence verified in s. Thus, to determine $R(\omega_{i+1})$ and $U(\omega_{i+1})$, we first determine $P(\omega_{i+1})$ as follows. Notice that $P(\omega_0) = \emptyset$. We identify the longest $\beta \in R(\omega_{i+1})$, such that $\alpha = \beta\varphi$, for some non-empty φ, and include the pair $(\delta(s_0, \beta), \varphi)$ in $P(\omega_{i+1})$. Notice that if α is the only recognized sequence, i.e., if $R(\omega_i) = \emptyset$, such a sequence β does not exist, in which case $P(\omega_{i+1})$ is empty. After the inclusion of a new pair into $P(\omega_{i+1})$, we check whether some sequences can be removed from $P(\omega_{i+1})$, so that it contains only the shortest verified sequences.

Once $P(\omega_{i+1})$ is determined, we can obtain $R(\omega_{i+1})$ and $U(\omega_{i+1})$ as follows. If α is recognized, for each $(\delta(s_0, \alpha), \varphi) \in P(\omega_{i+1})$, we include $\alpha\varphi \in pref(T)$ in $R(\omega_{i+1})$. The set of verified transitions $V(\omega_{i+1})$ is now $D_M \cap P(\omega_{i+1})$ and, consequently, $U(\omega_{i+1})$ becomes $D_M \setminus V(\omega_{i+1})$. The following algorithm shows how $P(\omega_{i+1})$ is determined after the Ξ-recognition of a sequence α and how $R(\omega_{i+1})$ and $U(\omega_{i+1})$ are obtained from $P(\omega_{i+1})$.

Algorithm 2.

```
Input: Sequence $\omega_{i+1}$ and $\Xi$-recognized sequence $\alpha$; the sets $R(\omega_i)$ and $U(\omega_i)$.
Output: $R(\omega_{i+1})$ and $U(\omega_{i+1})$
  •  $R(\omega_{i+1}) \leftarrow R(\omega_i) \cup \{\alpha\}$
  •  $P(\omega_{i+1}) \leftarrow P(\omega_i)$
  •  if $R(\omega_i) \neq \emptyset$ then
        o  Let $\beta$ be the longest sequence in $R(\omega_{i+1})$. Let $s = \delta(s_0, \beta)$ and $\varphi$ be
           the such that $\beta\varphi = \alpha$.
        o  $P(\omega_{i+1}) \leftarrow P(\omega_{i+1}) \cup \{(s, \varphi)\}$
        o  while there exist $(s, \tau), (s, \chi) \in P(\omega)$, such that $\tau = \chi\gamma$ and $\gamma \neq \varepsilon$ do
              $P(\omega_{i+1}) \leftarrow P(\omega_{i+1}) \setminus \{(s, \tau)\} \cup \{(\delta(s, \chi), \gamma)\}$
        o  end while
        o  $V(\omega_{i+1}) \leftarrow D_M \cap P(\omega_{i+1})$
        o  $U(\omega_{i+1}) \leftarrow D_M \setminus V(\omega_{i+1})$
        o  for each $\chi \in pref(\omega_{i+1}) \setminus R(\omega_{i+1})$, $s = \delta(s_0, \chi)$ do
              for each $(s, \varphi) \in P(\omega_{i+1})$ do
                    $R(\omega_{i+1}) \leftarrow R(\omega_{i+1}) \cup (\{\chi\varphi\} \cap pref(\omega_{i+1}))$
              end for
        o  end for
  •  end if
Return $R(\omega_{i+1})$ and $U(\omega_{i+1})$
```

The next lemma states that Step 1 of Algorithm 1 cannot be executed infinitely many times. This lemma is important to prove that Algorithm 1 terminates and that the obtained sequence is actually a checking sequence.

Lemma 4. *In Algorithm 1, Step 1 can be executed at most* $\sum_{s \in S} |E_s|$ *times without executing Step 2.*

Proof. We show that, for a given $s \in S$, the number of executions of Step 1 when α_i is such that $\delta(s_0, \alpha_i) = s$ is at most $|E_s|$. Let α_i and α_j be the shortest Ξ-recognized sequences (obtained in the i-th and the j-th iterations of Algorithm 1, respectively), such that $i < j$ and $\delta(s_0, \alpha_i) = \delta(s_0, \alpha_j) = s$. Notice that $\alpha_i \notin R(\omega_i)$, but $\alpha_i \in R(\omega_{i+1})$. If $\omega_{i+1} \in R(\omega_{i+1})$, then Step 2 must be executed. Therefore, suppose that $\omega_{i+1} \notin R(\omega_{i+1})$. In the next iteration of the algorithm, we have that $\omega_{i+1} = \alpha_i E_s$ and, thus, $\alpha_{i+1} < \alpha_i E_s$. Let β_i be the non-empty prefix of E_s, such that $\alpha_{i+1} = \alpha_i\beta_i$. Consider now α_j, i.e., the sequence which is Ξ-recognized in the j-th iteration of the algorithm. It follows from the definition of R that $\alpha_j\beta_i \in R(\omega_{j+1})$, since $\delta(s_0, \alpha_i) = \delta(s_0, \alpha_j)$ and α_i, α_j, $\alpha_i\beta_i \in R(\omega_{j+1})$. In the next iteration, we have that $\omega_{j+1} = \alpha_j E_s$ and, thus, $\alpha_{j+1} \leq \alpha_j E_s$. Then, there must exist a non-empty sequence $\beta_j \leq E_s$, such that $\alpha_{j+1} = \alpha_j\beta_j$ and $\beta_i < \beta_j \leq E_s$. As $|\beta_i| < |\beta_j| \leq |E_s|$, it follows that there exist at most $|E_s|$ executions of Step 1, such that $\delta(s_0, \alpha) = s$. ♦

Theorem 3. *Let ω be a sequence obtained by Algorithm 1. Then, ω is a checking sequence.*

Proof. When the algorithm terminates, $U(\omega) = \varnothing$, which implies that each transition is verified in $R(\omega)$. The algorithm indeed terminates because after each execution of Step 2, the number of unverified transitions is decreased by at least one. Therefore, Step 2 can be executed at most $|D_M|$ times. As, by Lemma 4, after a finite number of executions of Step 1, Step 2 must be executed, the number of iterations of the algorithm is finite.

In the first iteration of the algorithm, $\omega_0 = \varepsilon$ and $R(\omega_0) = \varnothing$. Then, Step 1 is executed and yields $\omega_1 = E_{s0}$. Thus, $\varepsilon \in R(\omega_1)$ and, consequently, $\varepsilon \in R(\omega)$.

We now show that $R(\omega)$ is a $\Im_\omega(M)$-confirmed set. By the definition of $R(\omega)$, we have that each $\alpha \in R(\omega)$ is either (i) Ξ-recognized or (ii) there exist β, $\beta\varphi$, $\chi \in R(\omega)$, such that $\delta(s_0, \beta) = \delta(s_0, \chi)$ and $\alpha = \chi\varphi$. Let $K \subseteq R(\omega)$ be the set of Ξ-recognized sequences. Observe first that, in the case (ii), we have that $\delta(s_0, \beta\varphi) = \delta(s_0, \alpha)$, which implies that if $\alpha \in R(\omega)$ is not Ξ-recognized, then another sequence that takes M to the same state as α should be. Thus, if there exists a sequence $\alpha \in R(\omega)$, $\delta(s_0, \alpha) = s$, there must exist at least one sequence $\beta \in \Phi(K, s)$. As each transition is verified and M is strongly connected, for each state s, there exists a sequence $\alpha \in \Phi(R(\omega), s)$. Thus, there exists $\beta \in \Phi(K, s)$ for each state s. Consequently, K is a state cover. By Lemma 2, K is a confirmed set. By the definition of $R(\omega)$ and Lemma 3, it follows that $R(\omega)$ is a confirmed set and satisfies the conditions of Theorem 1. Thus, ω is a checking sequence. ♦

4.1 An Example

We now illustrate the application of the method to the FSM M_1 in Figure 1. This machine has a distinguishing sequence $E = aa$. The distinguishing set contains three identification sequences $E_1 = E_2 = aa$, and $E_3 = a$. The intermediate values of ω_i, $R(\omega_i)$, and $U(\omega_i)$ are presented in Table 1. The obtained checking sequence $\omega = aaaaababaabaa$ has length of 13. For the same FSM, the method in [6] finds a checking sequence of length 32. An improved version of the method generates a checking sequence of length 15 [10].

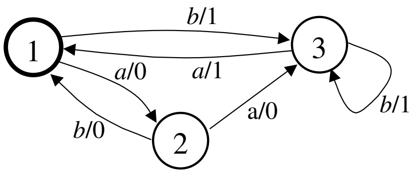


Fig. 1. Complete FSM M_1 from [11]

Table 1. Execution of Algorithm 1

i	**Step Executed**	ω_i	**Recognized Prefixes $R(\omega_i)$**	**Unverified Transitions $U(\omega_i)$**
0		ε	∅	{(1, *a*), (1, *b*), (2, *a*), (2, *b*), (3, *a*), (3, *b*)}
1	Step 1	*aa*	{ε}	{(1, *a*), (1, *b*), (2, *a*), (2, *b*), (3, *a*), (3, *b*)}
2	Step 1	*aaa*	{ε, *a*, *aa*}	{(1, *b*), (2, *b*), (3, *a*), (3, *b*)}
3	Step 1	*aaaaa*	{ε, *a*, *aa*, *aaa*, *aaaa*, *aaaaa*}	{(1, *b*), (2, *b*), (3, *b*)}
4	Step 2	*aaaaaba*	{ε, *a*, *aa*, *aaa*, *aaaa*, *aaaaa*, *aaaaab*, *aaaaaba*}	{(1, *b*), (2, *b*)}
5	Step 2	*aaaaababaa*	{ε, *a*, *aa*, *aaa*, *aaaa*, *aaaaa*, *aaaaab*, *aaaaaba*, *aaaaabab*, *aaaaababaa*}	{(2, *b*)}
6	Step 2	*aaaaababaabaa*	{ε, *a*, *aa*, *aaa*, *aaaa*, *aaaaa*, *aaaaab*, *aaaaaba*, *aaaaabab*, *aaaaababaa*, *aaaaababaab*, *aaaaababaaba*, *aaaaababaabaa*}	∅

4.2 Reset Feature

An FSM M has a reset feature if it has a special input (denoted r), which transfers it, as well as all its possible implementation machines, from any state to the initial state producing a null output, usually represented by the empty sequence ε. The reset transitions are assumed to be correct, i.e., verified; so Theorem 1 directly applies to FSMs with reset feature. Moreover, Algorithms 1 and 2 can be extended to deal with cases where the reset input is available. By default, the reset transitions are verified in each state and can be used to construct verified transfer sequences, which may result in shorter checking sequences. In Step 2 of Algorithm 1, when searching for a shortest verified transfer sequence β_i, the reset input may be used. In Algorithm 2, we have that $(s, r) \in P(\omega)$, for each state s and any input sequence ω.

Consider the partial FSM M_2 in Figure 2 (dashed lines represent the reset transitions) and the distinguishing set Ξ with $E_1 = E_2 = E_3 = E_4 = aa$. If Algorithm 1 is applied to this FSM, it generates the checking sequence $\omega = aaaaabaaabaabbabaa$ with length 18. However, if M_2 has the reset feature, the algorithm generates the checking sequence $\omega_r = aaaaabaaabaarabaa$ with length 17. The execution of the algorithm either with or without reset is the same up to the point where $\omega_i = aaaaabaaabaa$. At that point, the only unverified transition remains $(2, b)$. Notice that $\delta(s_0, \omega_i) = 3$. The shortest verified transfer sequence from state 3 to state 2 is ra, if the reset input is available, or bba, otherwise. In this case, the reset input contributes to shortening checking sequence.

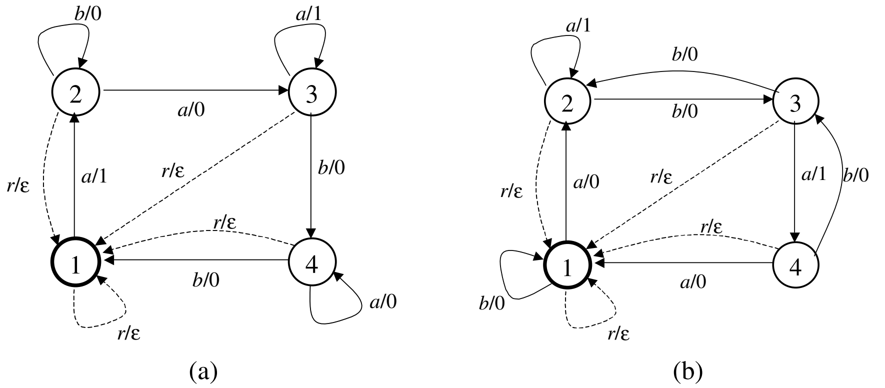


Fig. 2. (a) Partial FSM M_2 with reset feature; (b) Complete FSM M_3 with reset feature

There are cases, however, where the reset feature increases the length of a checking sequence generated by Algorithm 1, since the best local choices (e.g., shortest paths) made do not guarantee to lead to globally optimized checking sequences. Consider, for instance, M_3 in Figure 2 with the distinguishing sequence aa. The checking sequence without reset is $\omega = aaaabaaaabbaababaabaa$ of length 21, while the checking sequence with reset is $\omega_r = aaaabaaaabbaarbaababaa$ of length 22. The execution of the algorithm in both cases is the same, up to the point where $\omega_i = aaaabaaaabbaa$, $\delta(s_0, \omega_i) = 2$. At this point, $U(\omega_i) = \{(1, b), (4, b)\}$. If reset is not available, the shortest path chosen in Step 2 of Algorithm 1 is $\beta_i = ba$, which transfers to state 4. After this step, we have that $\omega_{i+1} = aaaabaaaabbaababaa$, $\delta(s_0, \omega_{i+1}) = 1$, and $U(\omega_{i+1}) = \{(1, b)\}$. Then, Step 2 is executed again, choosing $\beta_{i+1} = \varepsilon$. On the other hand, if reset is used, the shortest path chosen in Step 2 is $\beta_i = r$, which transfers to state 1. After this step, we have that $\omega_{i+1} = aaaabaaaabbaarbaa$, $\delta(s_0, \omega_{i+1}) = 2$, and $U(\omega_{i+1}) = \{(4, b)\}$. Then, Step 2 is executed, choosing $\beta_{i+1} = ba$. An interesting question investigated in Section 5.1 is the impact of using the reliable reset feature on the length of checking sequence for randomly generated FSMs.

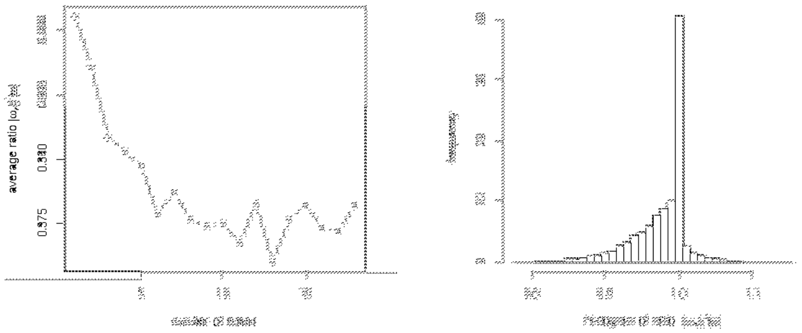
5 Experimental Results

This section describes an experimental evaluation of the method for constructing checking sequence proposed in this paper and some existing methods. We also investigate the reduction provided by the use of the reset feature. The experiments involve randomly generated FSMs. Since the existing methods treat only complete FSMs with distinguishing sequences, only such machines are considered to have a fair comparison.

We generate complete strongly connected reduced FSMs with a distinguishing sequence in the following way. Sets of states, inputs, and outputs with the required number of elements are first created. The generation proceeds then in three phases. In the first phase, a state is selected as the initial state and marked as "reached". Then, for each state *s* not marked as "reached", the generator randomly selects a reached state *s'*, an input *x*, and an output *y* and adds a transition from *s'* to *s* with input *x* and output *y*, and mark *s* as "reached". When this phase is completed, an initially connected FSM is obtained. In the second phase, the generator adds transitions (by randomly selecting two states, an input, and an output) to the machine until a complete FSM is obtained. If the FSM is not strongly connected, it is discarded and another FSM is generated. In the third phase, a distinguishing sequence is searched. If the FSM does not have a distinguishing sequence, it is discarded and another FSM is generated.

5.1 Reset Feature

As discussed in Section 4, our method can be applied to FSMs with the reset feature. The use of the reset input can result in shorter transfer sequences, which may shorten the resulting checking sequence. In this experiment, we evaluate the reduction obtained by the usage of the reset input. We randomly generated FSMs which have distinguishing sequence. Each FSM has two inputs, two outputs, and the number of states *n* ranging from three to 20. For each value of *n*, 1000 FSMs are generated.



(a) (b)

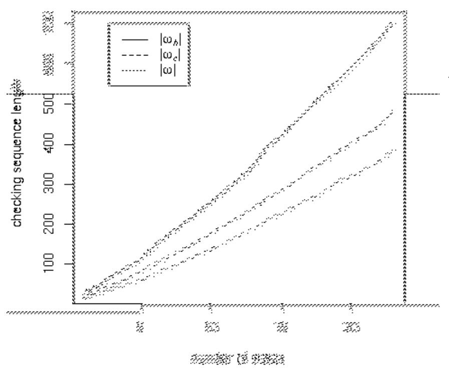
Fig. 3. Reduction ratio of the length of checking sequences with and without the reset input: (a) average ratio variation with respect to the number of states; (b) histogram of the ratio

For each FSM, a checking sequence ω is obtained using the proposed method. Then, we augmented the machine with the reset input and executed the method on the resulting FSM, obtaining a checking sequence ω_r. Figure 3(a) characterizes the variation of the average ratio $|\omega_r|/|\omega|$ with respect to the number of states. We observe that on average ω_r is about 2.5% shorter than ω. Figure 3(b) shows the frequency of the obtained reduction. Notice that in about 40% of the cases, the ratio is between 0.995 and 1.005, indicating that the reset feature has a little impact on the length of the checking sequence (at least for the chosen FSM parameters). However, in some cases, the checking sequence for the FSM augmented with the reset input is 20% shorter. On the other hand, in some cases it may be 10% longer. Thus, our experiments indicate that the reset feature does not significantly influence the length of checking sequences. However, more large scale experiments may be needed to confirm this conclusion.

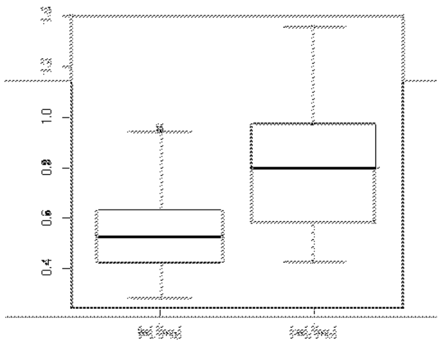
5.2 Comparison with Existing Methods

In this experiment, we compare the length of checking sequences generated by the proposed method and by the methods presented in [7] and [3]. We chose to consider machines with two inputs, two outputs, and number of states *n* ranging from three to 25. For each value of *n*, we randomly generated 1000 FSMs which have distinguishing sequence. For each FSM, we executed Algorithm 1, generating checking sequence ω. No execution took more than one second. Then, we executed Hierons and Ural's method [7], obtaining checking sequence ω_h, and Chen *et al.*'s method [3], which results in sequence ω_c. Figure 4(a) shows the average length of ω, ω_h and ω_c. The experimental data indicate that ω is, on average, 45% shorter than ω_h, while ω is, on average, 20% shorter than ω_c.

Figure 4(b) shows the boxplots of the ratios $|\omega|/|\omega_h|$ and $|\omega|/|\omega_c|$. Notice that the maximum value of $|\omega|/|\omega_h|$ is smaller than 1.0, thus, the proposed method generated shorter checking sequences than the Hierons and Ural's method in all experiments. Notice also that the reduction may be as high as 70%. On the other hand, compared with the Chen *et al.*'s method, method proposed in this paper generated shorter checking sequences in 75% of the cases, since the 3rd quartile of the boxplot is below 1.0.



(a)



(b)

Fig. 4. (a) Average lengths of ω, ω_h and ω_c; (b) Boxplots of the ratios $|\omega|/|\omega_h|$ and $|\omega|/|\omega_c|$

However, the maximum value of $|\omega|/|\omega_h|$ is 1.35, as in some experiments, the proposed method generated sequences 35% longer than Chen *et al.*'s method. On the other hand, in some cases our method generated sequences 60% shorter than Chen *et al.*'s method.

6 Related Work

There has been much interest in the generation of checking sequence, pioneered by Hennie's work [5]. Hennie discusses an approach for designing a checking sequence based on a distinguishing sequence. The method consists of two parts. The states are assumed to be ordered. In the first part, the distinguishing sequence is applied to each state, starting from the initial state, and transfer sequences are used to lead FSM to the second state. In the second part, each transition (s_i, x) is checked. To do this, the FSM is brought to a known state. This is done by transferring FSM to s_{i-1}, applying the distinguishing sequence, transferring to s_i (the same sequence used in the first part must be used), and applying x followed by the distinguishing sequence. Notice that Hennie does not assume that the FSM is initialized, thus checking sequence generated by his method should be prepended by a synchronizing or homing sequence, in order to bring both the specification and the implementation to a known state. Differently, we assume that the specification and the implementation are initialized and, thus, we do not use a synchronizing or homing sequence.

Gonenc [4] presents an algorithmic approach to generate checking sequence for FSMs with distinguishing sequences. The method, known as the method D, is divided into two parts, similarly to Hennie's work. In the first part, an α-sequence is generated, such that the distinguishing sequence is guaranteed to distinguish the states in the implementation. In an α-sequence, the distinguishing sequence is applied to each state of the FSM, using transfer sequences, if necessary. In the second part, a β-sequence is generated, such that each transition is checked. A β-sequence is a concatenation of transitions, followed by the distinguishing sequence, using transfer sequences, if necessary. Notice that α- and β-sequences correspond to the sequences generated in the first and the second part of Hennie's method, respectively. The α- and β-sequences are, then, concatenated to form a checking sequence. The number of transitions that are checked is reduced, using the fact that the last transition of the distinguishing sequence when applied to a state will be checked when all the other transitions of the distinguishing sequence were checked.

Kohavi and Kohavi [8] show that, instead of whole distinguishing sequence, suitable prefixes of it can be used to shorten the checking sequence. Boute [2] further shows that shorter sequences can be obtained if, instead of distinguishing sequences, distinguishing sets are used, and if the overlapping among the identification sequences is exploited.

Ural *et al.* [11] propose a method that attempts to minimize the length of checking sequence generated using distinguishing sequence. Sufficient conditions for a sequence to be a checking sequence are formulated there and used in several work, e.g., [6], [3], [10], [7], and [12]. The conditions are now further relaxed in Theorem 1 of this paper. The α- and β-sequences of Gonenc's method are divided in [11] into smaller pieces (the α-set and the set of transition tests) that are combined with appropriate transfer sequences to form a checking sequence. The problem of finding a minimal length checking sequence is then cast as a Rural Chinese Postman Problem

(RCPP), as previously proposed by Aho *et al.* [1]. The RCPP is an NP-complete problem of finding a minimal tour which traverses some required edges in an appropriate graph. Ural *et al.* show how a graph can be defined, such that the RCPP tour satisfies the stated sufficient conditions and, thus, it is a checking sequence. An α-set and a set of transfer sequences must be provided as input parameters of the method. Improvements to this method are proposed by Hierons and Ural [6] [7].

Chen *et al.* [3] demonstrate that the verification of last transition traversed by a distinguishing sequence applied to a particular state is implied by the verification of the other transitions. The authors present an (NP-complete) algorithm that identifies transitions to remove sequences ensuring their verification from the RCPP graph. The possibility of overlapping the distinguishing sequence, as proposed by Boute [2], is exploited by Ural and Zhang [10]. The RCPP graph modelling of Hierons and Ural [6] is modified, so that edges with negative cost are added to represent the possible overlapping. However, incorporating sequence overlapping comes with the price, since the size of the RCPP grows significantly.

All the above methods only deal with complete FSMs (however, partial machines can also be allowed, provided that definitions are adjusted as in this paper), and do not attempt to use the reset input to shorten the checking sequence. On the other hand, the method proposed in this paper can be used for generating checking sequence for a possibly partial FSM with a distinguishing set and, possibly, with the reset feature. The proposed method makes a local best choice in each step. This approach diverges from methods proposed in recent work (namely, [11], [6], [3], [7], [10]), which use graph-theoretical modeling for minimizing the length of checking sequence. We consider that our local optimization based approach has at least two advantages over the graph-theoretical ones, discussed below.

Firstly, the graph-theoretical methods attempt to globally optimize the length of checking sequence, but only after some input parameters are set, e.g., the α-sequences and transfer sequence set used by Ural *et al.* [11]. However, the length of checking sequence is influenced by these parameters and, thus, a sub-optimized sequence may be generated anyway. On the other hand, the method proposed in this paper does not require any input, besides the distinguishing set. Instead of assuming that suitable parameters are furnished, the algorithm makes choices based on the information available up to a certain execution point. The results of an experimental comparison indicate that the proposed method produces shorter checking sequences than existing methods in most cases. However, more experiments are needed to order to find a proper compromise between global and local optimization in generating checking sequence.

The second advantage of our approach is in its extensibility. For instance, it is not immediately clear how to the ideas of [10], [3] and [12] can be integrated in a same method, since each requires adaptations of the graph model which are not straightforward to merge. Our approach is based on a new problem casting, using the notion of “confirmed sequences”. This formulation allows us to relax the existing sufficient conditions and generalize them to partial reduced FSM with distinguishing sets. Further generalizations constitute our current work.

7 Conclusion

In this paper, we stated sufficient conditions for an input sequence to be a checking sequence for possibly partial FSMs with distinguishing sets. Based on these conditions, we proposed a method for generating checking sequence. The method can be used either with or without the reset feature. Moreover, the method allows the use of distinguishing sets, while recent methods deal only with FSMs with distinguishing sequences.

We experimentally compared the proposed method with existing generation methods, using randomly generated complete FSMs with distinguishing sequences. The results indicate that the proposed method generates shorter checking sequence in most cases. We also presented an experimental evaluation of the impact of the reset feature on the length of checking sequence. We noticed that, although shorter sequences may sometimes be obtained, our preliminary experiments indicate that the reset feature does not significantly influence the length of checking sequences.

As future work, we can mention several possible extensions of the presented results. For instance, the improvement suggested by Chen *et al.* [3] can be incorporated into the method proposed in this paper. The set of unverified transitions may be initialized with a subset of the defined transitions, following the algorithm of Chen *et al.* Finally, our experiments show that some checking sequences do not satisfy the suggested sufficient conditions, so there is still room for improvements on the conditions, which may lead to shorter checking sequences.

References

1. Aho, A.V., Dahbura, A.T., Lee, D., Uyar, M.U.: An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours. IEEE Transactions on Communications 39(11), 1604–1615 (1991)
2. Boute, R.T.: Distinguishing sets for optimal state identification in checking experiments. 23(8), 874–877 (1974)
3. Chen, J., Hierons, R.M., Ural, H., Yenigun, H.: Eliminating redundant tests in a checking sequence. In: Khendek, F., Dssouli, R. (eds.) TestCom 2005. LNCS, vol. 3502, pp. 146–158. Springer, Heidelberg (2005)
4. Gonenc, G.: A method for the design of fault detection experiments. IEEE Transactions on Computers 19(6), 551–558 (1970)
5. Hennie, F.C.: Fault-detecting experiments for sequential circuits. In: Proceedings of Fifth Annual Symposium on Circuit Theory and Logical Design, pp. 95–110 (1965)
6. Hierons, R.M., Ural, H.: Reduced length checking sequences. IEEE Transactions on Computers 51(9), 1111–1117 (2002)
7. Hierons, R.M., Ural, H.: Optimizing the length of checking sequences. IEEE Transactions on Computers 55(5), 618–629 (2006)
8. Kohavi, I., Kohavi, Z.: Variable-length distinguishing sequences and their application to the design of fault-detection experiments. IEEE Transactions on Computers 17(8), 792–795 (1968)
9. Moore, E.F.: Gedanken-experiments on sequential machines. Automata Studies, Annals of Mathematical Studies (34), 129–153 (1956)
10. Ural, H., Zhang, F.: Reducing the lengths of checking sequences by overlapping. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, pp. 274–288. Springer, Heidelberg (2006)
11. Ural, H., Wu, X., Zhang, F.: On minimizing the lengths of checking sequences. IEEE Transactions on Computers 46(1), 93–99 (1997)
12. Yalcin, M.C., Yenigun, H.: Using distinguishing and uio sequences together in a checking sequence. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, pp. 274–288. Springer, Heidelberg (2006)

Testing Systems Specified as Partial Order Input/Output Automata

Gregor v. Bochmann[1], Stefan Haar[2], Claude Jard[3], and Guy-Vincent Jourdan[1]

[1] School of Information Technology and Engineering (SITE)
University of Ottawa
800 King Edward Avenue, Ottawa, Ontario, Canada, K1N 6N5
{bochmann,gvj}@site.uottawa.ca
[2] IRISA/INRIA
Rennes, France
Stefan.Haar@irisa.fr
[3] Université Européenne de Bretagne
ENS Cachan, IRISA, Campus de Ker-Lann, 35170 Bruz, France
Claude.Jard@bretagne.ens-cachan.fr

Abstract. An Input/Output Automaton is an automaton with a finite number of states where each transition is associated with a single inpu**f** or output interaction. In [1], we introduced a new formalism, in which each transition is associated with a bipartite partially ordered set made of concurrent inputs followed by concurrent outputs. In this paper, we generalize this model to Partial Order Input/Output Automata (POIOA), in which each transition is associated with an almost arbitrary partially ordered set of inputs and outputs. This formalism can be seen as High-Level Messages Sequence Charts with inputs and outputs and allows for the specification of concurrency between inputs and outputs in a very general, direct and concise way. We give a formal definition of this framework, and define several conformance relations for comparing system specifications expressed in this formalism. Then we show how to derive a test suite that guarantees to detect faults defined by a POIOA-specific fault model: missing output faults, unspecified output faults, weaker precondition faults, stronger precondition faults and transfer faults.

Keywords: Testing distributed systems, partial order, finite state automata, conformance relations, partial order automata, HMSC.

1 Introduction

Finite State Machines (FSM) are commonly used to model sequential systems. When modeling concurrent systems, other models have been used, such as multi-port automata [13], where several distributed ports are considered and an input at a given port can generate concurrent outputs at different ports. The multi-port automata model is however not really adapted for truly distributed systems, since input concurrency is not taken into account. In [1], a new model of automata is introduced, where each transition is equipped with a *bipartite partially ordered set*, consisting of a set of concurrent inputs and a set of causally related concurrent outputs. This new model provides the ability to directly and explicitly specify concurrency between inputs, and causal relationships between inputs and outputs. A testing method for this new model was also proposed. Even though the model is up to exponentially smaller than the equivalent multi-port model, in a case of an automaton having an adaptive distinguishing sequence, the testing method proposed is able to generate a checking sequence which is polynomial in the number of inputs, and thus up to exponentially shorter than a checking sequence generated for an equivalent specification written as a multi-port automaton.

This model still has the limitation that no order constraint can be defined for the concurrent inputs of a given transition. In this paper we present a more general model where order constraints can be defined for inputs as well as outputs for a given transition. This provides a more symmetrical framework which simplifies the composition of several automata. The order constraints for inputs defined for a given automaton can then be interpreted as assumptions that are made about the behavior of the automaton's environment. A transition is therefore characterized by a multi-set of input/output events, where certain input or output interactions may occur several times, and a partial order between these events. We assume, however, that a transition starts with inputs and that there is no conflict between the initial inputs of different transitions starting from the same automaton state. This model is very close to High-level Messages Charts (HMSC), a standard already used to specify in a global manner dynamic behaviors of distributed systems [3].

We explain in this paper how the testing method that was defined for the previous model can be extended to our general case in an efficient manner. The basic idea is as follows: In order to test the order constraints imposed by a given input on the outputs of a given transition, first all inputs that may be applied (according to the partial order of the transition) before the given input, are applied and the resulting outputs are recorded. Then the given input is applied and the resulting outputs are recorded. A given output will occur in the first set of observed outputs if and only if it has no order constraint depending on the given input. The tests concerning the different inputs of a given transition can be combined into several test cases. However, several test cases are in general required to completely test the implemented partial order between inputs and outputs. Finally, the well-known methods for testing finite state machines can be used in our context for identifying the states of the automaton and to bring the implementation into the starting state from where a given transition can be tested.

In Section 2 of this paper, we first give an intuitive explanation of the model of Partial-Order Input/Output Automata (POIOA), and then give a formal definition. We also discuss different criteria for comparing the partial orders of different transitions, and based on this, how the behavior of different POIOA can be compared. In particular, we consider that the specification of a system component is given in the form of a POIOA M1, as well as an implementation of this component in the form of a POIOA M2. We define a conformance relation, called quasi-equivalence, which states that, if satisfied between M2 and M1, the implementation provides the outputs in an order satisfying the specification, provided that the environment of the component presents the inputs in an order satisfying the specification.

In Section 3, we present the testing methodology in detail and show that any deviation from quasi-equivalence will be detected by the derived test sequence. We also indicate how the results observed during the tests can be used to diagnose specific faults within the implementation. Then, in Section 4, we provide some discussion of the assumptions we have to make about the implementation in order to assure the detection of all faults by our testing method. We also discuss the assumptions we have made for our POIOA specification model. We give some justification for these assumptions and discuss why it may be interesting to remove some of these assumptions in future work.

2 Partial Order Input/Output Automata

2.1 Basic Concepts

An **Input/Output Automaton (IOA)** is an automaton with a finite number of states where each transition is associated with a single input or output interaction [2].

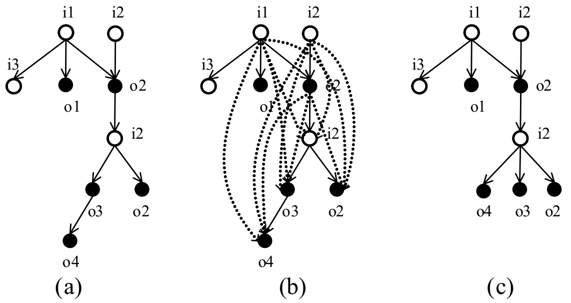


Fig. 1. Partially ordered multisets of input and output events

In this paper we consider a more general form of automata where each transition is associated with a partially ordered set of input and output events. For instance as shown in Figure 1(a), a given transition t may be associated with the following set of events: inputs i_1, i_2 (occurring twice), i_3, and outputs o_1, o_2 (occurring twice), o_3 and o_4 for which the following ordering relations must hold: $i_1 < o_1$; i_2 (first occurrence) $< o_2$ (first occurrence); $i_1 < o_2$ (first occurrence); o_2 (first occurrence) $< i_2$ (second occurrence); $i_1 < i_3$; i_2 (second occurrence) $< o_3$ and i_2 (second occurrence) $< o_2$ (second occurrence), and $o_3 < o_4$.

Here the notation *event-1* < *event-2* means that there is an order constraint between the occurrence of *event-1* and *event-2*, namely, that *event-2* occurs after *event-1*. The order between the events of a transition represent two aspects: (a) a safety property, and (b) a liveness property, sometimes called "progress property". The order *event-1* < *event-2* implies the safety property stating that event-2 will only happen after event-1 has already happened. If *Earlier-2* is the set of events *e* of the transitions for which *e* < *event-2* holds, then the order of events implies that *event-2* will eventually occur when all events in *Earlier-2* have occurred. We note that often, as shown in Figure 1(a), we are only interested in the basic order constraints from which the complete order relationship can be constructed by transitivity. For instance, Figure 1(b) shows the complete order relationship generated by the constraints shown in Figure 1(a).

It is clear that such specifications of partial order input/output automata (POIOA) allow for much concurrency between the inputs and outputs belonging to the same transition. We believe that this is an important feature for describing distributed systems. In fact, it is often difficult to determine, in a distributed system, in which order in time two particular events occur if they occur in different points in space. Therefore one may ask the question how it would be possible to check whether two interactions, for instance i_1 and i_3, occur in the order specified (for instance in the order $i_1 < i_3$ as specified above). One way to bring some rigor into this question is to introduce ports, sometimes called interaction points or service access points, and to associate each input/output event with a particular port of the distributed system. Then one may assume that the order of events at each port can be determined, while the order of events occurring at different ports can not be determined. The situation is similar in HMSC or in UML sequence diagrams, where vertical lines represent different system components and events belonging to the same component are normally executed in sequential order while the order of events at different components is only constrained by message transmissions. – In this paper we do not introduce ports nor system components. We simply assume that the order of execution of two events can be determined if a particular order of execution is specified for them.

For a **reactive** automaton, in the following called **input-guarded**, we assume that all initial events of each transition are inputs. We call an event of a transition initial if there is no other event that must precede it. In order to allow for a straightforward determination of the next transition of a POIOA, we assume that the following condition is satisfied concerning the initial events of different transitions starting from the same state: the set of initial events of two transitions starting from the same state must be disjoint. We say that the automaton has **exclusive transitions**.

In addition, we assume that each state of the automaton is a "strong synchronization point", that is, the initial input for the next transition will only become available after all events of the previous transition have occurred. We note, however, that this assumption may not always be realistic in a distributed system; and one may consider a distributed model with several local components where the sequential order between transitions is weak sequencing, that is, events pertaining to the next transition may occur at a given component after all **local** events of the previous transitions have (locally) occurred. This weak sequencing semantics has been adopted for HMSC [3], where the "local" events are those pertaining to a given system component, as for UML sequence diagrams. In fact, the model of HMSC is similar to our model of POIOA: A HMSC is a kind of state diagram where each state represents the execution of a sequence diagram (MSC) and the transition from one state to another represents the sequential execution of the two associated MSCs with weak sequencing semantics. In the POIOA model a transition can be equated to the partial order defined by a sequence diagram. It is to be noted that weak sequencing leads to many difficulties for the implementation of the specified ordering in a distributed environment, as discussed in many research papers [4,5,6,7].

As explained by Adabi and Lamport [8], the specification of the requirements that a system component must satisfy in the context of a larger system normally consists of two parts: (a) the assumptions that can be made about the behavior of the environment in which the component will operate, and (b) the guarantees about the behavior of the component that the implementation must satisfy. In the context of IOA (see for instance [9]), the guarantees are related to the output produced by the specified component (in relation to the inputs received), while the environment should satisfy certain assumptions about the order in which input to the component will be provided (in relation with the produced output earlier). In the case of a partially defined, state-deterministic IOA (where the state is determined by the past sequence of input/output events), the fact that in a given state some given input is not specified is then interpreted as the assumption that the environment will not produce this input when the component is in that given state.

During the testing of an implementation for conformance to an IOA specification, two types of problems may occur: After a given execution trace, that is, a given sequence of input and output events, the specification will be in a particular state. If the next event that occurs does not corresponds to a transition of the IOA specification then we have encountered a problem: If the event is an output, an implementation fault has been detected; if it is an input, this is an unexpected input, also called "unspecified reception", which represents a wrong behavior of the environment.

A specification of a system component C in the form of a POIOA S_C, similarly, can be interpreted as defining assumptions about the environment of C and guarantees that the implementation of C must satisfy. The difference between an IOA and a POIOA is that a transition of the latter is characterized by a set of input/output events with a defined partial order instead of a single input or output event. Similarly as for an IOA, one can define a dual specification for a given POIOA which represents the most general behavior of the environment and can be used for testing an implementation of the given specification.

It is clear that the behavior of a *POIOA* S can be modeled by an *IOA* S' as follows: The states of S' include the states of S and a large number of intermediate states that correspond to the partial execution of a transition. For instance, the POIOA transition t shown in Figure 2(a) can be modeled by the IOA transitions shown in Figure 2(c). The conformance testing of an implementation in respect to a specification S may therefore be performed by a test suite that checks the performance in respect to S' and that is obtained by one of the known test development methods for finite state machines or IOA. However, this approach is not very efficient since the equivalent IOA specification S' is in general much more complex than the original POIOA specification S.

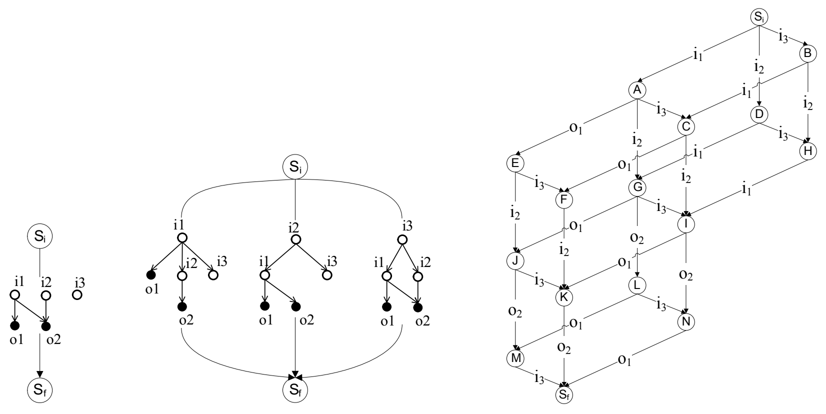


Fig. 2. (a) specification of transition. (b) implementation of the specified transition in terms of three separate transitions. (c) An Input/Output Automata model equivalent to the POIOA transition shown in (a).

We propose in this paper a method for deriving a test suite that guarantees to detect all faults defined by a POIOA-specific fault model. Specifically, we propose to test an implementation for the following faults:

- *Unspecified output:* An output produced during a given transition is not in the set of outputs specified; or the number of occurrences of an output is larger than allowed by the specification.
- *Missing output*: An output foreseen by the transition is not produced after all possible inputs (those that could be applied before that output according to the specification) have been applied.
- *Unsupported input*: The implementation does not support the reception of all input events foreseen in the specification.
- *Missing output constraint:* An output foreseen by a transition is produced before all the events that are specified as precondition have occurred.
- *Additional output constraint depending on input:* An output foreseen by a transition is not produced after all its precondition events have occurred, but it is produced after certain other expected input events have been applied.
- *Additional input constraints:* The implementation does not support the reception of certain inputs in some order allowed by the specification.
- *Transfer fault:* A transition of the implementation leads to a state different from what is specified.

Transfer faults are tested by simply adapting the classical state recognition and transition verification methods from IOAs to POIOAs [1]. Testing for the other faults require new techniques, especially the missing and additional ordering constraints. Our approach to catch these types of faults is to test each input event of a given transition separately, as explained in Section 3. When combined with transfer fault detection, our new tests might have to be applied a number of times per transition according to the classical fault detection methods.

2.2 Formalization of the Automata Model

In the following, we suppose that two disjoint nonempty sets I and O (representing inputs and output) are given; set $V= I \cup O$. Recall that a *partial order* is a pair (E, $\leq$) where E is a set and $\leq$ *is* a transitive, antisymmetric and reflexive binary relation on E, and $<$ is the irreflexive part of $\leq$. The set of *minimal* elements of E is the set $min(E)=\{x\in E,$ for all $z\in E,\ z \leq x$ implies $z=x\}$. For two elements x and y of E, we note $x||y$ if neither $x \leq y$, nor $y \leq x$.

Definition 1. *An **V-pomset** (partial order multi-set) is a tuple $\omega= (E, \leq, \mu)$ such that*

1. *(E, $\leq$) is a partial order, and*
2. *μ: $E \rightarrow V$ a total mapping called the **labeling.***

A POIOA is a special kind of finite state transition machine. A POIOA has a finite number of states and transitions, and each transition is associated with a partial order of input and output events, as follows:

Definition 2. *A **Partial Order Input/Output Automaton** (or **POIO Automaton, POIOA**) is a tuple $M= (S, s^{in}, I, O,T)$, where*

1. *S is a finite set of **states** and $s^{in}\in S$ is the **initial state**; the number of states of M is denoted $n = |S|$;*
2. *I and O are disjoint and nonempty **input** and **output** sets, respectively;*
3. *$T \subseteq S\times\Omega(I\cup O) \times S$ is the finite set of **transitions**, where $\Omega(I\cup O)$ is the set of all $(I\cup O)$-pomsets.*

*We say that an POIOA is **input-guarded** if for each transition the minimal events are all inputs.*

*We say that an POIOA has **exclusive transitions** if for any two transitions $t1=(s, \omega_1,s_1)$ and $t_2=(s,\omega_2,s_2)$ starting from the same state s, we have $min(\omega_1) \cap min(\omega_2)= \emptyset$. Note: This implies that the next transition from a given state is determined by the first input that occurs.*

*We say that a POIOA is **strongly synchronized** if all input/output events of one transition are performed before any event of the next transition of the automata occurs.*

We consider in this paper strongly synchronized, input-guarded POIOA with exclusive transitions. The implications of these restrictions are discussed in Section 4.

In this paper we assume that a POIOA is the specification of a distributed system where the inputs and outputs may occur at different system interfaces. For the purpose of testing an implementation for conformance with a given POIOA specification, we also assume that we can model the implementation by a POIOA and that the implementation satisfies certain assumptions that can be modeled by restrictions on the form of the POIOA implementation model.

2.3 Comparing the Behavior of Different POIOA

In this section we first define several basic conformance relations that can be used for comparing the behavior of different POIOA, for instance the specification of a system and its implementation. It turns out that these basic relations correspond to the different types of faults that an implementation may have. We then discuss the nature of these relations and define the quasi-equivalence relation that can be easily tested by the method described in Section 3.

Definition 3. *We say that a $(I \cup O)$-pomset $\omega_I = (E_I, \leq_I, \mu_I)$ is quasi-equivalent to a $(I \cup O)$-pomset $\omega_S = (E_S, \leq_S, \mu_S)$, written $\omega_I =>_{qe} \omega_S$, if and only if:*

1. *The input and output events are the same, that is, $E_I = E_S = E$, and, $\mu_I = \mu_S = \mu$*
2. *The following conditions are satisfied concerning the order relations between events: for all e, o, i in E such that $\mu(o) \in O$ and $\mu(i) \in I$*
   a. *$e <_S o$ implies $e <_I o$*
   b. *$e <_S i$ implies $e <_I i$ or $e \,||_I\, i$*
   c. *$e \,||_S\, i$ implies $e \,||_I\, i$*

The means that ω_I (corresponding to an implementation) is quasi-equivalent to ω_S (corresponding to a specification) if the sets of input events and output events are the same and certain conditions are satisfied for the order relations of the two *$(I \cup O)$*-pomsets. We talk about a **missing output fault** if some output event of ω_S is not included in ω_I. We talk about an **unexpected output fault** if, on the contrary, some output event of ω_I is not included in ω_S. We say that the implementation has an **unsupported input fault** if input event of ω_S is not included in ω_I. If, on the contrary, there is an input event in ω_I that is not included in ω_S, then this means that the implementation is ready to accept additional input which is not a fault; this condition would normally not be tested.

The conditions concerning the order relations have the following significance. Condition 2(a) states that if an output event has an order constraint in ω_S then it must have the same constraint in ω_I. If that is not the case, we say that the implementation has a **missing output constraint fault**. Condition 2(c) implies (when e is an output) that an output event that has no order constraint depending on input in the specification should not have such a constraint in the implementation. If this is not true, we talk about an **additional output constraint depending on input fault**. We note that if two output events have no ordering relation in the specification, there is the possibility that a quasi-equivalent implementation introduces such a relation.

The ordering constraints on inputs are described by Conditions 2(b) and 2(c). They imply that the implementation may not introduce any **additional input constraints**. However, an implementation may not rely on the assumption that an ordering constraint for input defined in the specification is actually observed by the environment; such a more powerful implementation is allowed.

Quasi-equivalence captures the notion of “compatible implementation”, an implementation that accepts any inputs compatible with the specification (and may accept more) and which will produce outputs defined by the specification in an order compatible with the specification.

Definition 4. *Let $M= (S, s^{in}, I, O, T)$ be a POIOA. A (finite)* ***transition trace*** *of M is a word $w = \omega_1\omega_2\omega_3...\omega_n$ such that there exist $t_1t_2t_3... t_n \in T^*$ such that the $t_i=(s_i, \omega_i, s_i^+)$ satisfy*

1. *$s_1= s^{in}$*
2. *$s^+_i=s_{i+1}$ for all i.*

We denote the set of transition traces of M as Tr(M).

Definition 5. *Let $M= (S, s^{in}, I, O, T)$ and $M'= (S', s^{in'}, I, O, T')$ be two POIOA, and let $w = \omega_1\omega_2\omega_3...\omega_n$ be a transition trace of M and $w' = \omega'_1\omega'_2\omega'_3...\omega'_n$ be a transition trace of M'. We say that w is quasi-equivalent to w' (written $w \Rightarrow_{qe} w'$) iff (by induction)*

- *if $n=1$ ($w = \omega_1$ and $w' = \omega'_1$) $\omega_1 \Rightarrow_{qe} \omega'_1$*
- *else ($w = w_1\ \omega_2$ and $w' = w_1'\ \omega'_2$.) $w_1 \Rightarrow_{qe} w_1$ and $\omega_2 \Rightarrow_{qe} \omega'_2$.*

We now define the notion of *trace quasi-equivalence* between two POIOA as being the fact that the traces of one POIOA are by quasi-equivalence included in the traces of the other one. Note: This notion has some similarity with the notion of quasi-equivalence as defined for partially defined finite state machines.

Definition 6. *Let $M= (S, s^{in}, I, O, T)$ and $M'= (S', s^{in'}, I', O', T')$ be two POIOA. M is* ***trace quasi-equivalent to*** *M' if*

1. *$I' \subseteq I$ and $O' \subseteq O$*
2. *For each t' in Tr(M'), there is a t in Tr(M) such that $t \Rightarrow_{qe} t'$*

In summary, the trace quasi-equivalence of a POIOA M with a POIOA M' (where M may be the implementation of M') means that (a) M may have a different number of states than M', (b) M may have additional transitions (for which there are no corresponding transitions in M') which must have exclusive initial inputs with the transitions defined in M', and these additional transitions may involve inputs and outputs that are not defined for M'. However, the transitions of M that correspond to transitions in M' must be quite similar: (c) they must involve the same input and output events, and (d) they must have very similar order relations, as defined by points 2(a), (b), and (c) in Definition 3.

The interesting property of trace quasi-equivalence is the following: If an implementation M is trace quasi-equivalent to the specification M', then this implementation will exhibit the (safeness and liveness) properties to be guaranteed by the outputs according to the specification, if the assumptions concerning the applied inputs, as specified by the specification, are satisfied by the real environment in which the implementation evolves.

3 Transition Testing

In this section, we explain a method for generating test cases for POIOA and outline the diagnostics for each of the possible implementation faults outlines in Section 2.1. We then show how to combine the elementary test cases for specific input events into longer sequential test cases, and conclude with a method for testing full conformance of implementations with reliable resets. We concentrate on the testing of individual transitions and their input/output behavior, related to unspecified and missing outputs, unsupported inputs, missing output constraints, additional output constraints on input and additional input constraints. For the testing of transfer faults, the known methods developed for finite state machines can be directly applied to POIOA [11].

3.1 Single Input Event Testing

Let $M= (S, s^{in}, I, O, T)$ be a POIOA, and let $t=(s,\omega,s') \in T$ be a transition of T. Our goal is to generate a set of test cases, in the following called test suite, to verify that an implementation of M has *correctly implemented* a corresponding transition. By *correctly implemented* we mean that the corresponding transition is quasi-equivalent to the transition of the specification (according to Definition 3). We therefore have to test that the implemented transition has the same input and output events as specified for ω and that the constraints between events are compatible with the definition of quasi equivalence.

We make the following assumptions regarding the testing environment:

- Assumption 1: we can detect when the implementation receives "unspecified input" in the case that the implementation has additional input constraints and receives an input event that it did not expect. In such a situation, the implementation may for instance return some error message to the tester.
- Assumption 2: we can observe the order between two outputs if an order relation is defined in the specification.

Notation: Given a $(I\cup O)$-pomset $\omega = (E_\omega, \leq_\omega, \mu)$ and an element $x \in E_\omega$, we write $\mathcal{P}(x) = \{y \in E_\omega, y <_\omega x)\}$ for the set of elements that are before x (the "Past" of x). We write $\mathcal{NF}(x) = \mathcal{P}(x) \cup \{y \in E_\omega, x \parallel_\omega y\}$ for the set of elements of E_ω that are neither greater than nor equal to x (element Not in the "Future" of x).

3.1.1 Basic Test Suite

We say that a set of inputs is serialized if it respects the ordering constraint of the specification and the inputs are sent one at a time, waiting for all possible output to be produced before the next input is sent.

For each input event i of the transition $t=(s,\omega,s')$, the test suite should include the following test case, where we assume that the implementation is already in the starting state of the corresponding transition:

1. Enter all the input events in $\mathcal{NF}(i)$ in a serialized way, and observe the multiset $S1$ of produced output events.
2. Enter input event i and observe the multiset $S2$ of produced output events.

3. Enter the input events of ω that have not been input yet in a serialized way, and observe the multiset *S3* of produced output events.

We say that an implementation *I* is *almost quasi equivalent* to a specification *S* if it is quasi equivalent except for possible missing output constraints on outputs, and possible added output constraints on input.

Proposition: The tested transition of the implementation is almost quasi-equivalent to the corresponding transition in the specification if and only if, for all inputs of the transition, the observed output multisets S1, S2 and S3 have the values predicted by the specification.

The following diagnostics can be issued depending on the outputs observed within the three multisets S1, S2 and S3:

- *Unspecified output fault:* the number of occurrences of an output *o* in $S1 \cup S2 \cup S3$ is larger than the number of occurrence of that output as specified in ω. In this case, at least one occurrence of *o* is produced by the implementation while not being specified by the POIOA specification.
- *Missing output*: the number of occurrence of an output *o* in $S1 \cup S2 \cup S3$ is smaller than the number of occurrence of that output as specified in ω. In this case, at least one occurrence of *o* is not produced by the implementation while being specified by the POIOA specification.
- *Unsupported input*: Based on Assumption 1, this fault will be detected through an error message received from the implementation.

If there is a single fault for a given output event, we can diagnose the following faults:

- *Missing output constraint depending on input:* the number of occurrences of the output *o* in *S1* is larger than the number predicted by the specification. In this case, at least one occurrence of *o* is produced by the implementation prior to the input *i* while *i* is specified by the POIOA specification as a precondition of *o*.
- *Additional output constraint depending on input:* the number of occurrences of the output *o* in *S2* is larger than the predicted by the specification. In this case, at least one occurrence of *o* is produced by the implementation only after the input *i* (that is, *i* is a precondition for *o* in the implementation) while *i* is not specified by the POIOA specification as a precondition of *o*.
- *Additional input constraint on input:* Since inputs will be applied in all allowed relative orders, such faults will be detected through an error message received from the implementation, based on Assumption 1.

3.1.2 Testing Output-Output Ordering Constraints

The basic test suite detailed above will not necessarily detect a fault of missing output constraint depending on another output. For instance, the order between o_3 and o_4 in Figure 1(a) is not realized in the implementation shown in Figure 1(c). In the case of such a fault, two outputs that are ordered in the specification could be concurrent in the implementation, or implemented in the opposite order. The case of concurrent implementation would allow for nondeterminism, which means that the implementation may produce these outputs sometimes in the order defined by the specification and sometimes in the opposite order. If we want to detect these faults, we have to observe not only the sets S1, S2 and S3, but also the order in which the outputs in each set occur. And in the case that the behavior of the implementation may be non-deterministic, we have to execute an appropriate test case several times: if in one of the observed execution scenarios, the ordering of outputs is reversed, then we know that the implementation does not realize the specification. For a positive verdict, we need to execute the test a large number of times in order to obtain a sufficiently high statistical assurance that the right order was not obtained by chance.

3.1.3 Added Input Constraints on Output

The test cases presented so far will not necessarily catch the case of an additional input constraint depending on output. In this case, an output and an input are specified as concurrent but the implementation expects the output to be produced before accepting the input. This is an error since it is legitimate, according to the specification, to enter the input before the output is produced, but the implementation would not accept this behavior.

We note that such an additional input constraint may introduce, by transitivity, an additional constraint between inputs. If this is the case, the latter constraint will be detected by the basic test suite. Otherwise, specific test, as described below, must be applied in order to detect the additional input constraints depending on output.

For each input event i of the transition $t=(s,\omega,s')$, let $\omega' = (E_\omega \leq_\omega, \mu) \backslash \{ P(i) \cup i \}$ be the sub-pomset of ω restricted to the events that do not belong to $P(i) \cup i$, and let $O_{min} = \{x \in \min(E_{\omega'}), x \in \mu(E) \cap O\}$ be the set of minimal elements of ω' that are output events.

If $O_{min} \neq \emptyset$, then the suite should include the following test case, where we assume that the implementation is already in the starting state of the corresponding transition:

1. Enter all the input events in $P(i) \cup i$, respecting the ordering constraints, and (if possible) before any element of O_{min} is produced by the implementation.
2. Observe that the implementation outputs all elements of O_{min}.

Note that the tester may fail to perform the first step, in that elements of O_{min} are produced before all of $P(i) \cup i$ can be input. In this case, the test is *INCONCLUSIVE* and should be done again.

Proposition: *If a transition successfully passes the basic test suite and the above test case for each input that has a non-empty O_{min}, then the implementation has no additional input constraints depending on outputs.*

Proof: Assume that a constraint o > i has been added by the implementation. By definition, it means that in the specification, $i \parallel_\omega o$. When testing i, if $o \in O_{min}$ then the test above will exhibit the problem (input i will put the implementation in unspecified behavior mode, which will be detected based on Assumption 1). If $o \notin O_{min}$ then either there exists $o' \in O_{min}$ such that $o' >_\omega o$, or there exists $i' \in \omega'$ such that $i' >_\omega o$ and $i' \parallel_\omega i$. In the former case, the constraint o' > i has been added (by transitivity) and we are back to the first case. In the second case, the constraint i' > i has been added (by transitivity) and this will be seen when applying the basic test suite to i'.

3.2 Testing Several Input Events

The tests described in the Section 3.1 are designed to test the behavior of the implementation in respect to the behavior in relation with a particular input event of a particular transition. We show in the following how several of these elementary test cases can be combined into a larger test case that covers several input events of a given transition. This can be done as long as the corresponding input events are all ordered in the partial order of the transition $(I \cup O)$-pomset.

Let $i_1, i_2, \ldots, i_k$ be a set of input events of the $(I \cup O)$-pomset $\omega = (E_\omega, \leq_\omega, \mu)$ of a transition of the IOPOA going from state s to state s' such that $i_1 \leq_\omega i_2 \leq_\omega \ldots \leq_\omega i_k$. Then the basic test cases for these inputs, as described in Section 3.1.1, can be combined into a single test case that tests all k inputs sequentially as follows, assuming that the implementation is in the starting state of the transition:

1. For $m=1$ to k
   a. Enter all the input events in $\mathcal{NF}(i_m)$ that have not been input yet, respecting the ordering constraints, and observe the multiset $S1_m$ of produced output events.
   b. Enter input event i_m and observe the multiset $S2_m$ of produced output events.
2. Enter the input events of ω that have not yet been input, respecting the ordering constraints, and observe the multiset $S3$ of produced output events.

The diagnostics described in Section 3.1.1 can be adapted in the following way:

- For missing and unspecified outputs faults, use the multiset $S1_1 \cup S1_2 \cup \ldots \cup S1_k \cup S2_1 \cup S2_2 \cup \ldots \cup S2_k \cup S3$, in place of $S1 \cup S2 \cup S3$.
- For missing and additional output constraints, apply the diagnostics described in Section 3.1.1 for each input i_m separately ($m=1$ to k), using the multiset $S1_1 \cup S1_2 \cup \ldots \cup S1_m \cup S2_1 \cup S2_2 \cup \ldots \cup S2_{m-1}$ in place of S_1, and $S2_m$ in place of S2.

Moreover, it can be easily shown that the test cases of two concurrent inputs (as defined in Section 3.1) cannot be combined into a single pass through the transition. Indeed, if i_1 and i_2 are two concurrent inputs, then by definition $i_2 \in \mathcal{NF}(i_1)$ and $i_1 \in \mathcal{NF}(i_2)$, so whichever input is tested first, the other one will have to be entered during Step 2 of the test case and thus will not be testable anymore during the same pass through the transition.

Since we have shown that we can test any number of input events sequentially in the same pass over a given transition if and only if these events are all mutually ordered in the $(I \cup O)$-pomset of that transition (they define a *chain* in the order), we can deduce that an optimal strategy (in terms of number of passes) for testing a given transition for all inputs consists of finding the minimum number of chains of the order that would include all input events. This is a classical property of ordered set theory called the *minimal chain decomposition* of an order [10] (and the number of chains in the minimal chain decomposition is equal to the largest number of mutually incomparable elements of the order). Thus, in order to test a transition associated with the $(I \cup O)$-pomset $\omega = (E_\omega, \leq_\omega, \mu)$ in an optimal way, we must create $\omega_I = (E_I, \leq_I, \mu_I)$, the projection of ω onto the input events, then create a minimal chain decomposition of $(E_I, \leq_I)$, and finally, for each chain of the decomposition, bring the implementation to the starting state of the transition and apply the above combined test case for the input events of that chain.

3.3 Discussion

We now come back to the assumptions we have made about the POIOA that represent the system specification and about the implementation that are also modeled as POIOAs. When we apply our testing method, we make the following assumptions about the specification and the implementation:

1. **Assumption 1 – unspecified input are detectable:** We think that this assumption (see Section 3.1) is reasonable, since the behavior of an implementation in response to unexpected input is not well defined. Therefore it can be assumed that the behavior observed in such a situation would be different from what the specification prescribes.
2. **Assumption 2 – specified ordering of outputs can be observed:** As explained in Section 2, we assume that the ordering of events can be observed if the specification prescribes a specific order. Therefore this assumption (see Section 3.1) is reasonable.
3. **Bounded response time:** This means that when all precondition events of a given output event have occurred, then the implementation will produce the output within a bounded time delay. Therefore, if the output did not occur within this delay, it can be concluded that it will not be produced without any additional input being applied. This appears to be a reasonable assumption. It is also made for testing finite state machines, where after each input one has to wait for the resulting outputs before the next input is applied.
4. **No transition splitting in the implementation:** We have implicitly assumed that the implementation of a given transition of the specification can be modeled as a single transition in the implementation POIOA. As an example, we consider the case of the specification transition shown in Figure 2(a). It has three concurrent initial inputs. A valid implementation may foresee three different transitions, depending on which of the three inputs occurs first, as shown in Figure 2(b). Assume now that the implementation of the third transition has a missing output (o_1 does not occur). Then this fault will not be detected by the test cases derived from the specification according to our method. In order to detect all such faults, one may adapt our test generation procedure by applying it not to the specification, but to a refined specification model that has split transitions, as shown in Figure 2(b). However, this leads to very long test suites, since the number of the split interleavings could be exponentially larger than the number of order constraints.

4 Conclusion

In this paper, we present a generalization of a previous test method where only bipartite orders where permitted. Relaxing the bipartite constraint allows to specify any combination of inputs and outputs, with any concurrency or ordering constraints between these events. So, we get closer to the standard modeling languages like HMSC or UML sequence diagrams. We provide a formal model for these automata, and give the formal definition of the quasi-equivalence relationship which can be used for the comparison between the behaviors of two automata. We then provide a testing methodology which can be used to determine whether any implementation is quasi-equivalent to a given specification. Finally, we explain the assumptions we have made about the implementations under test and about the POIOA model, provide some justifications and discuss why it would be interesting to remove some of these assumptions in future work.

Acknowledgments. This work has been supported in part by grants from the Natural Sciences and Engineering Research Council of Canada, as well as by the INRIA sabbatical support for the second author.

References

[1] Haar, S., Jard, C., Jourdan, G.V.: Testing Input/Output Partial Order Automata. In: Petrenko, A., Veanes, M., Tretmans, J., Grieskamp, W. (eds.) TestCom/FATES 2007. LNCS, vol. 4581, pp. 171–185. Springer, Heidelberg (2007)

[2] Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Quarterly 2(3), 219–246 (1989)

[3] Mauw, S., Reniers, M.: High-level Message Sequence Charts. In: Cavalli, A., Sarma, A. (eds.) Proceedings of the Eight SDL Forum, SDL 1997. Time for Testing - SDL MSC and Trends, Evry, France, September 23-26, 1997, pp. 291–306 (1997)

[4] Alur, R., Etessami, K., Yannakakis, M.: Realizability and verification of MSC graphs. Theor. Comput. Sci. 331(1), 97–114 (2005)

[5] Mooij, A., Romijn, J., Wesselink, W.: Realizability criteria for compositional MSC. In: Johnson, M., Vene, V. (eds.) AMAST 2006. LNCS, vol. 4019, Springer, Heidelberg (2006)

[6] Mooij, A.J., Goga, N., Romijn, J.: Non-local choice and beyond: Intricacies of MSC choice nodes. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, Springer, Heidelberg (2005)

[7] Castejón, H.N., Bræk, R., Bochmann, G.V.: Realizability of Collaboration-based Service Specification. In: APSEC conference (November 2007)

[8] Abadi, M., Lamport, L.: Conjoining specifications. ACM Transactions on Programming Languages & Systems 17(3), 507–534 (1995)

[9] Bochmann, G.v.: Submodule construction for specifications with input assumptions and output guarantees. In: FORTE 2002, Chapman & Hall, Boca Raton (2002)

[10] Dilworth, R.P.: A decomposition theorem for partially ordered sets. Annals of Mathematics (51), 161–166 (1950)

[11] Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines – a survey. Proceedings of the IEEE 84(8), 1089–1123 (1996)

[12] Gouda, M.G., Yu, Y.-T.: Synthesis of communicating Finite State Machines with guaranteed progress. IEEE Trans on Communications Com-32(7), 779–788 (1984)

[13] Luo, G., Dssouli, R., Bochmann, G.v., Ventakaram, P., Ghedamsi, A.: Generating synchronizable test sequences based on finite state machines with distributed ports. In: Proceedings of the IFIP Sixth International Workshop on Protocol Test Systems, Pau, France, September 1993, pp. 53–68 (1993)

Test Purpose Concretization through Symbolic Action Refinement

Alain Faivre[1], Christophe Gaston[1], Pascale Le Gall[2], and Assia Touil[3,⋆]

[1] CEA LIST Saclay
F-91191 Gif sur Yvette Cedex
[2] Ecole Centrale Paris - Laboratoire MAS
F-92295 Chatenay Malabry
[3] Supelec - Computer Science Department
F-91192 Gif sur Yvette Cedex

Abstract. In a Model Driven Design process, model refinement methodologies allow one to denote system behaviors at several levels of abstraction. In the frame of a model-based testing process, benefits can be taken from such refinement processes by extracting test cases from the different intermediate models. As a consequence, test cases extracted from abstract models often have to be concretized in order to be executable on the System Under Test. In order to properly define a test concretization process, a notion of conformance relating SUTs and abstract models has to be defined. We define such a relation for models described in a symbolic manner as so-called IOSTSs (Input Output Symbolic Transition Systems) and for a particular kind of refinement, namely action refinement, which consists in replacing communication actions of abstract models with sets of sequences of more concrete communication actions. Our relation is defined as an extension of the *ioco*-conformance relation which relates SUTs and models whose communication actions are defined at the same level of abstraction. Finally we show from an example how a test purpose resulting from an abstract IOSTS-model can be concretized in a test purpose defined at the abstraction level of the SUT.

Keywords: Model Based Testing, Action Refinement, Symbolic Conformance Testing, Test Purpose, Test Purpose Concretisation.

1 Introduction

Model Driven Design approaches allow one to describe systems at several abstraction levels. In an usual software development top-down approach, requirements are expressed at a very high abstraction level in a given model and refined into several more concrete models which successively detail more an more the implementation choices: this process is also called the *refinement process*. When dealing with reactive systems, automata based languages including input and

⋆ This work was partially supported by the RNTL French project EDEN2 and by the Action Marie Curie TAROT.

output mechanisms can be used as modeling languages: models denote behaviors in term of so-called *traces* which are sequences of inputs and outputs exchanged with the environment through interfaces. In the context of automata-based languages, such interfaces are often denoted by symbols representing communication channels. The ability to describe interfaces at different levels of abstraction is important in a refinement process: it allows one to denote behaviors while abstracting from implementation details concerning interfaces. Let us illustrate on a simple example of an ATM application: in an abstract model, expressing that a PIN-code is entered and compared to the actual PIN-code does not require to detail the concrete mechanisms to enter it: the corresponding interface can be denoted by an abstract channel PIN through which the number representing the PIN-code transits. At a more concrete level, one may specify that the abstract input sent through the channel PIN is in fact refined into a sequence of four inputs through a channel $DIGIT$ corresponding to the four digits composing the PIN-code. Traces from the abstract model are concretized into new ones, built over concrete interface elements which are directly compatible with the real system. Defining how an abstract interface is mapped to a concrete one and how abstract inputs and outputs are refined is a key point in a refinement process and is usually called *action refinement* [3].

The paper presents an approach to take benefits from action refinement in a model-based testing process. In a previous contribution [2], we have proposed a method to extract test cases from models given in the form of Input-Output Symbolic Transition Systems (IOSTS). IOSTSs are automata based models involving data and communication actions (input and output actions) denoted in a symbolic manner: those input (resp. output) actions denote sets of actual input (resp. output) values received (resp. emitted) through channels. Test cases are traces of an IOSTS-model, characterizing sequences of stimulations (*i.e.* input values) to be sent to the *System Under Test* (SUT) and of intended reactions (*i.e.* outputs values) of the SUT. In [2], we have built test cases from *test purposes* described as tree-like structures obtained by *symbolically executing* IOSTS-models. Defining a test purpose amounts to choosing a finite number of finite paths in the symbolic execution tree as behaviors to be tested. Symbolic execution has been first defined for programs ([4]) and mainly consists in replacing concrete input values of variables by symbolic ones in order to compute constraints induced on these variables by the execution of the program. Symbolic execution applied to IOSTS follows the same intuition considering guards of transitions as conditions and assignment together with communication actions as instructions. Symbolic execution of an IOSTS results in a so-called *symbolic execution tree* in which each path characterizes a set of behaviors constituted by all traces obtained by solving the constraints associated to the path. A test case associated to the test purpose will interact with the SUT in order to make it perform at least one trace per chosen finite path. Moreover, such test purposes may be automatically defined [2]: this is useful to generate test purposes with no human intervention while ensuring a coverage of the IOSTS-model behaviors. Applying such coverage criteria on the different models of a refinement process strengthens even more the coverage of all specified behaviors. However, when dealing with test purposes extracted from abstract models, due to the action refinement steps, it is not possible to directly define test cases interacting with the SUT: they have to be concretized at the description level of the SUT interface.

In this paper we extend the symbolic model-based testing framework defined in [2] to deal with action refinement, following the approach proposed in [9]. We adapt the notion of *refinement pairs* as defined in [9] in a non-symbolic context. Intuitively, refinement pairs associate abstract communication actions to some concrete communication action sequences. Concrete IOSTS can be derived from an abstract one by replacing abstract communication actions by their associated concrete communication action sequences. As in [9] we focus on input action refinement, but since in our framework, a symbolic input action is an abstraction of a set of input values, a symbolic refinement pair denotes a (possibly infinite) set of refinement pairs as defined in [9]. Moreover, in [9], the authors consider linear atomic input-inputs refinement: an abstract input is refined as one sequence of concrete inputs (regardless of particular intermediate action sequences denoting the quiescence of the SUT). We also consider atomic input-inputs refinement, but we do not require linearity: an abstract symbolic input can be refined by a set of possible input sequences. This is useful to define complex refinements in which an abstract input is refined in a non deterministic manner (*e.g.* an amount of money to be sent to an ATM may be concretely entered in different ways depending on the chosen coins). As in [9], conformance between an abstract model and a SUT is defined by means of a dedicated conformance relation derived from the ioco-conformance relation [7]. The usual ioco-conformance is adapted to cope with sets of symbolic input action refinements. We then establish a result which can be seen as an extension of the one given in [9], stating that *ioco*-conformance to a concrete model is equivalent to the extended notion of ioco-conformance to an abstract model provided that concrete and abstract considered models are related by the refinement pair set used to define the extended ioco-conformance relation. Finally, we illustrate with the help of an example how an abstract test purpose can be concretized through refinement pairs.

Paper organization. In Section 2, we introduce IOSTS and define symbolic action refinement pairs. In Section 3, we extend symbolic model based testing to take into account refinement of actions. Section 4 contains our example of test purpose concretization.

2 Symbolic Action Refinement for IOSTS

2.1 Input/Output Symbolic Transition Systems(IOSTS)

We assume the reader familiar with basic notions of many sorted first-order equational logic [5]. We recall notations about IOSTS as given in [2], [6] and [1]. Let us first introduce the data part specified with a many sorted first-order equational logic. A *data signature* is a couple $\Omega = (S, Op)$ where S is a set of types, Op is a set of operations, each one being provided with a profile $s_1 \cdots s_{n-1} \rightarrow s_n$ (for $i \leq n$,

$s_i \in S$). The set $T_\Omega(V) = \bigcup_{s \in S} T_\Omega(V)_s$ of *terms* with typed variables in $V = \bigcup_{s \in S} V_s$ is inductively defined as usual over Op and V. For any term t in $T_\Omega(V)$, $Var(t)$ denotes the set of all variables of V occurring in t. A *variable renaming* is any injective mapping $\mu_V : V \to V'$ and can be canonically extended to the set of terms $T_\Omega(V)$. A *substitution* is a function $\sigma : V \to T_\Omega(V)$ preserving types which can also be canonically extended to $T_\Omega(V)$. $T_\Omega(V)^V$ denotes the set of all substitutions defined on V. The definition domain $Dom(\sigma)$ of a substitution σ is the set $\{x | x \in V, x \neq \sigma(x)\}$. The set $Sen_\Omega(V)$ of all typed equational *formulas* contains the truth values *true* and *false* and all formulas built using the equality predicates $t = t'$ for $t, t' \in T_\Omega(V)_s$, and the usual connectives $\neg$, $\vee$ and $\wedge$.

A *model* is a family $M = \{M_s\}_{s \in S}$ with, for each $f : s_1 \cdots s_n \to s \in Op$, a function $f_M : M_{s_1} \times \cdots \times M_{s_n} \to M_s$. *Interpretations* are applications ν from V to M preserving types, extended to terms in $T_\Omega(V)$. A model M satisfies a formula φ, denoted by $M \models \varphi$, iff, for all interpretations ν, $M \models_\nu \varphi$, where $M \models_\nu t = t'$ is defined by $\nu(t) = \nu(t')$, and where the truth values and the connectives are handled as usual. M^V is the set of all interpretations from V to M. In the sequel, we only use integers, booleans, enumerated types and character strings as data types. Thus, data types are interpreted in a fixed model denoted M and defined for a given data signature $\Omega = (S, Op)$ dedicated to specify those data types.

IOSTS-signatures are couples (A, C) where $A = \bigcup_{s \in S} A_s$ is a set of *attribute variables* and where C is a set of *communication channels*. *Sig* is the set of all IOSTS-signatures. For two signatures $\Sigma_1 = (A_1, C_1)$ and $\Sigma_2 = (A_2, C_2)$, usual set operators can be extended: $\Sigma_1 \subseteq \Sigma_2$, iff $C_1 \subseteq C_2$ and $A_1 \subseteq A_2$; $\Sigma_1 \cup \Sigma_2 = (A_1 \cup A_2, C_1 \cup C_2)$; $\Sigma_1 \cap \Sigma_2 = (A_1 \cap A_2, C_1 \cap C_2)$; finally $\Sigma_1 \backslash \Sigma_2 = (A_1 \backslash A_2, C_1 \backslash C_2)$.

The *set* $Act(\Sigma)$ *of communication actions* over an IOSTS-signature Σ contains the unobservable action τ, the set $Input(\Sigma) = \{c?y \mid c \in C, y \in A\}$ whose elements are called *receptions* or *input actions* and the set $Output(\Sigma) = \{c!t \mid c \in C, t \in T_\Sigma(A)\}$ whose elements are called *emissions* or *output actions*.

Definition 1 (IOSTS). *An* IOSTS *over a signature* $\Sigma = (A, C)$ *is a tuple* $G = (Q, q_0, Trans)$ *where* Q *is a set of* states*,* $q_0 \in Q$ *is the* initial state *and* $Trans \subseteq Q \times Act(\Sigma) \times Sen_\Omega(A) \times T_\Omega(A)^A \times Q$ *is a set of* transitions*.*

A transition $tr = (q, act, \varphi, \rho, q')$ *of* $Trans$ *is composed of a source state* $source(tr) = q$*, an action* $act(tr) = act$*, a guard* $guard(tr) = \varphi$*, a substitution of variables* $subs(tr) = \rho$ *and a target state* $target(tr) = q'$*.*

$STS(\Sigma)$ *denotes the set of all IOSTS over the signature* Σ*.*

For a transition tr with $act(tr) = c?x$, $rec(tr)$ denotes the variable x and for an IOSTS $G = (Q, q_0, Trans)$, $\mathcal{T}_{c?}(G)$ is the subset of $Trans$ of all transitions tr such that $act(tr)$ is of the form of $c?x$. We also denote $Att(G)$ for A, $init(G)$ for q_0, $Trans(G)$ for $Trans$, $Interface(G)$ for C and $Sig(G)$ for Σ.

Example 1. Fig. 1 depicts a model of a very simple drink vending machine, called Abstract Vending Machine, or AVM for short. This machine allows the user to

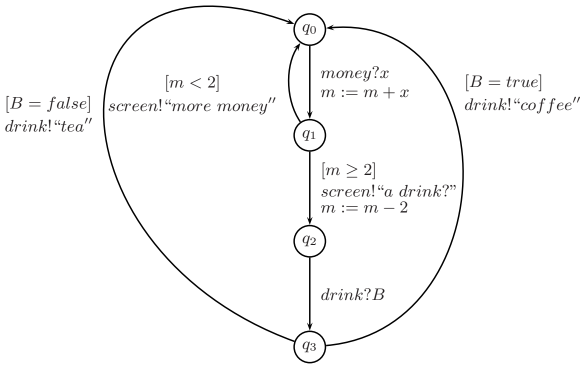


Fig. 1. AVM: an example of IOSTS

order a coffee or a tea. First, the user introduces some money. If the amount is greater than or equal to the cost of drinks (here 2) the user can choose a tea or a coffee, else he has to introduce more money. At last, the machine serves the user the asked drink.

The set $Obs(\Sigma)$ of *observations over* Σ is $(C \times \{?, !\} \times M)$. An observation of the form $c!m$ (resp. $c?m$) is called an output (resp. input) value. The set $Run(tr) \subseteq M^A \times (Obs(\Sigma) \cup \{\tau\}) \times M^A$ of *runs* of $tr = (q, act, \varphi, \rho, q') \in Trans$ is s.t. $(\nu^i, act_M, \nu^f) \in Run(tr)$ iff: (1) if act is of the form $c!t$ (resp. τ) then $M \models_{\nu^i} \varphi$, $\nu^f = \nu^i \circ \rho$ and $act_M = c!\nu^i(t)$ (resp. $act_M = \tau$), (2) if act is of the form $c?y$ then $M \models_{\nu^i} \varphi$, there exists ν^a such that $\nu^a(z) = \nu^i(z)$ for all $z \neq y$, $\nu^f = \nu^a \circ \rho$ and $act_M = c?\nu^a(y)$.

For a run $r = (\nu^i, act_M, \nu^f)$, we denote $source(r)$, $act(r)$ and $target(r)$ respectively ν^i, act_M and ν^f.

As in [7,8], we will use $\delta!$ to denote *quiescence*: quiescence refers to situations for which it is not possible to execute an output action.

For an IOSTS G, the set of its finite paths, denoted $FP(G)$ contains all finite sequences $p = tr_1 \ldots tr_n$ of transitions in $Trans(G)$ such that $source(tr_1) = init(G)$ and for all $i < n$, $target(tr_i) = source(tr_{i+1})$. The set of *runs* of p denoted $Run(p)$ is the set of sequences $r = r_1 \ldots r_n$ such that for all $i \leq n$, $r_i \in Run(tr_i)$ and for all $i < n$, $target(r_i) = source(r_{i+1})$. Following the approach of [8] and with the notation $Tr(r) = act(r_1) \ldots act(r_n)$ for $r \in Run(p)$, the set $STr(p, r)$ of *suspension traces of a run* r *of* p is the least set s. t.:

- If p can be decomposed as $p'.tr$ with $tr \in Trans(G)$ and with r of the form $r'.r_{tr}$ with $r_{tr} \in Run(tr)$, then $\{m.act(r_{tr}) | m \in STr(p', r')\} \subseteq STr(p, r)$ with the convention that τ is the neutral element for action concatenation.
- If there exists no finite path $p.p'$ for which there exists $r.r_1 \cdots r_k \in Run(p.p')$ with for all $i \leq k-1$, $act(r_i) = \tau$ and $act(r_k) = c!m$ for some c and m, then for any[1] $\delta_m \in \{\delta!\}^*$, $Tr(r).\delta_m \in STr(p, r)$.

[1] A^* denotes the set of finite sequences of elements of A.

The set of *suspension traces of a path* p is $STr(p) = \bigcup_{r \in Run(p)} STr(p, r)$ and *semantics* of G are $STr(G) = \bigcup_{p \in FP(G)} STr(p)$.

We note $STr(\Sigma) = \bigcup_{G \in STS(\Sigma)} STr(G)$ and $STr = \bigcup_{\Sigma \in Sig} STr(\Sigma)$. Moreover, for any observation trace $st \in STr$, $Interface(st)$ is the set of channel names occurring in at least one observation of st.

For a finite path p, we define the set $Def(p)$ of its defined variables. It contains all attribute variables which are defined along p. Intuitively, a variable z is defined either if there is a reception on the variable z for one of the transitions occuring in p or if there exists a variable substitution ρ associated to a transition of p such that $z \in Dom(\rho)$ and all variables in $\rho(z)$ are already defined in the sub-path preceding the considered transition. On the contrary, when a variable z is assigned by a transition substitution to a term containing undefined variables, then z becomes undefined and should be removed from the set of defined variables. More formally, the set of defined variables for a path may be characterized inductively as follows:

Definition 2 (Set of defined variables). *Let* $G = (Q, q_0, Trans)$ *be an IOSTS over* Σ. *Let* p *be a finite path of* $FP(G)$. *The* set $Def(p)$ of defined variables *of* p *is defined as follows:*

(1) if p *is of the form* ε, $Def(p) = \emptyset$
(2) if p *is of the form* $p'.tr$ *with* $tr = (q, act, \varphi, \rho, q')$

$$Def(p) = (Def^{tr}(p') \cup \{z | z \in Dom(\rho) \wedge Var(\rho(z)) \subseteq Def^{tr}(p')\}) \setminus \{z | z \in Dom(\rho) \wedge Var(\rho(z)) \not\subseteq Def^{tr}(p')\}$$

with $Def^{tr}(p') = \begin{cases} Def(p') \cup \{x\} & \textit{if act is of the form } c?x \\ Def(p') & \textit{otherwise} \end{cases}$

Finally, a *System Under Test* SUT is defined by a set $STr(SUT) \subseteq STr$. We note $Interface(SUT) = \bigcup_{st \in STr(SUT)} Interface(st)$. $STr(SUT)$ is required to be stable by prefix[2], and to be *input-complete*, that is: $\forall st \in STr(SUT), \forall c \in Interface(SUT), \forall m \in M, st.c?m \in STr(SUT)$.

2.2 Symbolic Action Refinement for IOSTS

An action refinement indicates which concrete action sequences implement an abstract action and possibly concretize the data handled in the abstract action by decomposing it into several concrete data. For instance, one can refine the abstract input action $money?x$ as many successive concrete input actions $coin?y_i$ as necessary to cope with the required money amount. Intuitively the abstract variable x is related to the concrete variables y_i by the equality $x = y_1 + \ldots + y_n$ with n denoting the number of successive input actions $coin?y_i$. In the sequel, we focus on the refinement of symbolic input actions as in [9]. In fact, the case of output actions is simpler than the one of input actions[3] and it can be treated in

[2] $STr(SUT)$ is stable by prefix if any prefix of traces in $STr(SUT)$ belongs to $STr(SUT)$.

[3] Indeed, contrarily to input actions, output actions do not impact values assigned to attribute variables.

a similar way. We simply do not treat it because we want to limit the complexity of definitions.

We characterize a refinement pair R as a couple associating a channel name c which defines the class of abstract input actions to be refined to an input refinement describing all the associated intended behaviors: an input refinement is a tuple composed of an IOSTS $G = (Q, q_0, Trans)$, an exit (or final) state s belonging to Q and a variable χ. Intuitively, an input action $c?x$ can be refined in any of the finite paths starting from the entry state q_0 and ending at the exit state s. We now discuss about the usefulness of χ. As we are in a symbolic framework, the input action to be refined can be any input action through the channel c on any attribute variable of the abstract IOSTS. For generality sake, our refinement mechanism is independent of the attribute variable introduced in the input action to be refined. The variable χ allows us to link the symbolic input action and the refining IOSTS G by applying some renaming mechanisms on G substituting χ by the targeted abstract attribute variable (this is done in Definition 4).

Definition 3 (Input Refinement/Refinement pair). *An* input refinement *is a triple* (G, s, χ) *with* $G = (Q, q_0, Trans)$ *an IOSTS,* $s \in Q$ *with* $s \neq q_0$ *called the* final state *and denoted by* $final(G)$*, and* χ *a variable satisfying* $\chi \notin Att(G)$ *such that:*

- $\forall p \in FP(G)$*, there exists* p' *with* $p.p' \in FP(G)$ *and* $target(p.p') = s$.
- $\forall tr \in Trans$*,* $act(tr) \notin Output(\Sigma)$.
- $\forall p \in FP(G), target(p) = s \Rightarrow \chi \in Def(p)$.

For any $c \in C$*, the couple* $(c, (G, s, \chi))$ *is called* a refinement pair*.*

We note $RP(\Sigma)$ *the set of all refinement pairs* $(c, (G, s, \chi))$ *with* $Sig(G) = \Sigma$*.*

By abuse, G will be called the input refinement or the refining IOSTS.

Intuitively, a refinement pair $(c, (G, s, \chi))$ denotes the capacity of refining any action of the form $c?x$ by all the paths of G from q_0 to s, provided that within the IOSTS G, the variable χ has been first substituted by x.

Let us point out that actions used in refining IOSTS are required to be either input actions or internal actions. This restriction has already been made in [9] and corresponds to a simplification motivated by testing issues: imposing that restriction ensures the ability to define the variable χ only in relation to the refining input actions. If refinement of input actions would authorize output actions, the SUT might have different ways to answer and this would make the testing process more difficult. By requiring that refinement actions of abstract input actions are only made of input actions, we can compute *a priori* a sequence of refining input actions matching the targeted abstract one. Such a restriction, also known as input-inputs refinement, is thus of particular interest for testing since the level of controllability is maintained by action refinement.

Example 2. In Fig. 2, we show an IOSTS, denoted G in the sequel, which is used to build an input refinement $R = (G, s, \chi)$. A refinement pair $(money, (G, s, \chi))$

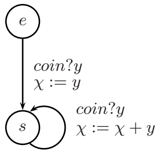


Fig. 2. Refining IOSTS G

is then constituted of an abstract channel *money* and the input refinement $R = (G, s, \chi)$ built over G. In order to put money in the drink vending machine (the AVM of Example 1), we have to put coins whose sum corresponds to the total amount sent on the channel *money*. The refinement consists in decomposing that amount into repetitive receptions of coins whose accumulated sum corresponds to the abstract amount. For example, if the price of a drink is 2, the customer can put either a coin of 2 or two coins of 1. Let us remark that all paths in G clearly define the variable χ, whose value exactly corresponds to the sum of introduced coins. Indeed, let us point out that for any path p of G starting from e, $Def(p) = \{\chi, y\}$.

We now define the function refining an abstract model G_a into a concrete model w.r.t. a refinement pair $(c, (G, s, \chi))$. Intuitively, all input actions of the form $c?x$ of G_a are replaced by the IOSTS G in which χ is renamed by x. All possible paths of G starting at q_0 and ending at s concretize $c?x$.

Definition 4 (Refinement Function). *Let $\Sigma = (A, C)$ and $\Sigma_R = (A_R, C_R)$ be two IOSTS-signatures, such that $A \cap A_R = \emptyset$.*

We define the refinement function

$$\begin{array}{llll} ref : RP(\Sigma_R) & \times\, STS(\Sigma) & \rightarrow STS(\Sigma \cup \Sigma_R) \\ \quad R = (c, ((Q, q_0, Trans), s, \chi)) & G_a{=}(Q_a, q_{a0}, Trans_a) & \mapsto (Q', q_{a0}, Trans') \end{array}$$

and the family of attribute renaming $\mu_{A,rec(tr)}$ indexed by the variable $rec(tr)$ such that $\mu_{A,rec(tr)}(\chi) = rec(tr)$ and for all variables y of Σ_R, verifying $y \neq \chi$, $\mu_{A,rec(tr)}(y) = y$.

$ref(R, G_a) = (Q', q_{a0}, Trans')$ is the IOSTS s.t.[4]*:*

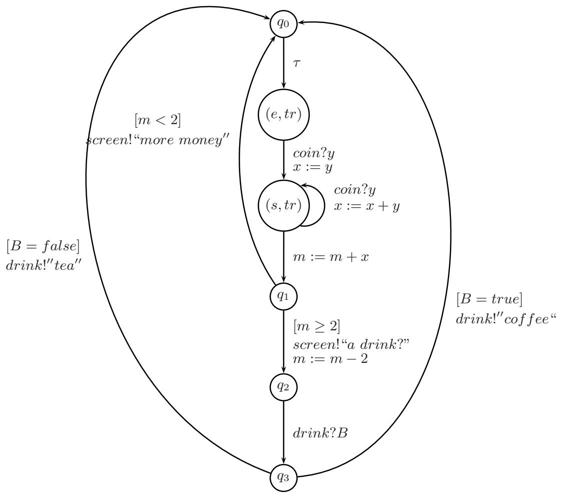
$$Q' = Q_a \cup \{(q, tr) \mid q \in Q, tr \in \mathcal{T}_{c?}(G_a)\}$$

$$\begin{array}{ll} Trans' = & (Trans_a \setminus \mathcal{T}_{c?}(G_a)) \\ & \bigcup_{tr \in \mathcal{T}_{c?}(G_a)} (\{((q, tr), \mu_{A,rec(tr)}(act), \mu_{A,rec(tr)}(\varphi), \mu_{A,rec(tr)}(\rho), (q', tr)) \\ & \quad \mid (q, act, \varphi, \rho, q') \in Trans\} \\ & \quad \cup \{(source(tr), \tau, guard(tr), id_A, (q_0, tr)), \\ & \quad ((s, tr), \tau, true, subs(tr), target(tr))\}) \end{array}$$

$ref(R, G_a)$ is the concretization *of G_a w.r.t. the refinement pair R.*

[4] id_A is the identity function on A and $\mu_{A,rec(tr)}$ is extended to communication channels and formulae in a canonical way.

The refinement function has the following two interesting properties. (1) Since the refining IOSTS are such that the variable under refinement, denoted χ in Definition 3, is defined, the refinement of a symbolic input action $c?x$ ensures that in $ref(R, G_a)$, the attribute variable x receives a value controlled by the input refinement (this value is the result of a function only depending on the concrete sequence of input values). Such variables are sometimes referred as *controllable*. (2) The refinement of a symbolic input action can restrict the set of reachable values for the targeted attribute variable x since concrete input actions can be specialized as much as wished according to the designer choices.



q_0
τ
(e, tr)
$coin?y$ $x := y$
(s, tr)
$coin?y$ $x := x + y$
$m := m + x$
q_1
$[m < 2]$ $screen!$"more money"
$[B = false]$ $drink!$"tea"
$[B = true]$ $drink!$"coffee"
$[m \geq 2]$ $screen!$"a drink?" $m := m - 2$
q_2
$drink?B$
q_3

Fig. 3. CVM IOSTS

Example 3. We refine the AVM of Example 1 using the refinement pair (*money*, (G, s, χ)) of Example 2. The transition tr of source q_0 and target q_1, carrying the abstract action $money?x$ is replaced by concrete behaviors. For this purpose, we rename each state st of G by (st, tr). Here we obtain two new states (e, tr) and (s, tr). The variable χ is replaced by the variable x that corresponds to the reception of the amount in the abstract specification. The refinement function connects then the states (e, tr) and (s, tr) respectively to q_0 and q_1 with τ transitions and adds the substitution $m := m + x$ on an exit transition. Finally, we obtain a concrete model *Concrete Vending Machine*, or CVM for short, that specifies how to introduce coins to order some drink.

We may use several refinement pairs to refine a given abstract model. It suffices to iteratively apply the refinement function ref with the considered refinement pairs. In order to ensure that refining an abstract IOSTS G_a according to a refinement pair RP_1 defined for a channel $c1$ then refining according to another refinement pair RP_2 for a channel $c2$ (with $c1 \neq c2$) leads to the same model than refining G_a according to RP_2 and then to RP_1, it suffices to require that for any considered refinement pair $(c, (G, s, \chi))$, c is an abstract channel of G_a and the refining IOSTS G does not share a channel with the abstract specification G_a under consideration. Thus, the channels occurring in the refining IOSTS which appear in the intermediate refined specifications from G_a cannot be refined later in the next refinement steps. Under this condition, the order of refinement steps does not matter. Indeed, let us recall that a refinement pair ensures that the abstract reception variable occurring in the abstract action to be refined is defined within the input refinement (up to the variable renaming). This means that for each path of the input refinement, the resulting value associated to the abstract reception variable is uniquely defined in function of the concrete input actions. In other words, the value of the abstract reception variable cannot depend either on the attribute variables of the abstract IOSTS to be refined or on the attribute variables of the input refinements. So, it does not depend on the context of use of the refinement pairs. Let us also remark that different input refinements may share some concrete channels or attributes.

In the sequel, for any abstract specification G_a and for any family of refinement pairs $\mathcal{R}$ verifying the above hypotheses (not the same abstract channel for two refinement pairs, no abstract channel in refining IOSTS), then we note $ref(\mathcal{R}, G_a)$ the resulting specification obtained by applying the refinement function on G_a for all refinement pairs in $\mathcal{R}$. We note $\Sigma_{\mathcal{R}}$ (resp. $C_{\mathcal{R}}$) the union of all signatures Σ_R (interfaces $Interface(G_R)$) for the refinement pairs $(c_R, (G_R, s_R, \chi_R))$ belonging to $\mathcal{R}$ with $Sig(G_R) = \Sigma_R$.

3 IOCO Conformance up to Refinement

Testing a system w.r.t. a specification requires the definition of a conformance relation. Our approach is based on the *ioco*-conformance relation [7].

Definition 5 (*ioco*). *Let G be an IOSTS and SUT be a system under test*[5] *such that $Interface(G) = Interface(SUT)$. SUT is* ioco-conform *to G, iff for any $str \in STr(G) \cap STr(SUT)$, if there exists act of the form $c!v$ or $\delta!$ such that $str.act \in STr(SUT)$, then $str.act \in STr(G)$. In such a case, we also say that str is* ioco-conform *to G.*

We extend Definition 5 in order to reason about conformance of a SUT to an IOSTS-model G_a up to a set of refinement pairs. This extension requires to define abstractions of traces of the SUT. Those abstractions are traces only involving channels of $Interface(G_a)$. For a refinement pair RP and a trace *str*, we define the set of inputs that can be concretized in the form of *str* through RP.

Definition 6. *Let $RP = (c, (G, s, \chi))$ be a refinement pair. Let str be a suspension trace of STr. Let p be a finite path of G with $target(p) = s$ such that $str \in STr(p)$. Such a path p is called a* complete path *of str.*

[5] Let us recall that by hypothesis, a system under test is such that its set of traces is stable by prefix and input-complete.

We note $Run(str,p) \subseteq \{r \mid r \in Run(p), str \in STr(p,r)\}$*, and* $CP(str)$ *the set of all complete paths of* str*.*

For any run $r = r_1 \cdots r_n$ *of* $Run(str,p)$*,* $target(r_n)(\chi)$ *is called the* value assigned by str to χ through p and r*. One notes* $(str,p)(\chi)$ *the set of all values assigned by* str *to* χ *through* p *and* r *for any* $r \in Run(str,p)$*.*

The set $Abs(str, RP)$ of abstractions of str associated to RP *is the set:*

$$\{c?v \mid \exists p \in CP(str), v \in (str,p)(\chi)\}$$

Note that $Abs(str, RP)$ is necessarily empty if str is not a trace only made up of observations which are either inputs whose associated channels are in $Interface(G)$ or $\delta!$.

A refinement process generally involves a family of refinement pairs. Therefore we generalize Definition 6 to define the set of inputs that can be concretized in the form of str through at least one refinement pair of the family.

Definition 7. *Let* $\mathcal{R}$ *be a family of refinement pairs and* $str \in STr$*. The* set of abstractions of str associated to $\mathcal{R}$ *is the set* $Abs(str,\mathcal{R}) = \bigcup_{RP \in \mathcal{R}} Abs(str, RP)$*.*

Note that we do not require that several IOSTSs defined in different refinement pairs of $\mathcal{R}$ do not share channels (*i.e.* have disjoint interfaces). Therefore, there may exist $RP_1 \neq RP_2 \in \mathcal{R}$ s.t. $Abs(str, RP_1) \neq \emptyset$ and $Abs(str, RP_2) \neq \emptyset$: a concrete trace may be abstracted through several refinement pairs.

Suspension traces of $STr(SUT)$ can only be composed of: inputs and outputs defined over channels of $Interface(G_a)$, the action $\delta!$, and inputs defined over channels introduced by refinement pairs of $\mathcal{R}$. Thus we generalize Definition 7 to define abstractions of any trace of $STr(SUT)$.

Definition 8. *With notations of Definition 7, the* set of abstract suspension traces of str associated to $\mathcal{R}$ *is the set* $AbsC(str,\mathcal{R})$ *defined as follows:*

– *if* str *is of the form* ε *then*

$$AbsC(str,\mathcal{R}) = \{\varepsilon\}$$

– *if* str *is of the form* $a.str'$ *where* a *is an observation s.t.* $a \notin Obs(\Sigma_{\mathcal{R}})$ *then*

$$AbsC(str,\mathcal{R}) = \{a.str'' \mid str'' \in AbsC(str',\mathcal{R})\}$$

– *if* str *is of the form* $str_r.str'$ *where* $str_r \in STr(\Sigma_{\mathcal{R}})$*,* $str_r \neq \emptyset$*, and* str' *is the empty trace or a trace beginning by an observation not in* $Obs(\Sigma_{\mathcal{R}})$*, let us consider* $Dec(str_r)$ *the set of all decompositions* (pr, sf) *of* str_r *(i.e* $str_r = pr.sf$ *and* $pr \neq \varepsilon$*), then* $AbsC(str,\mathcal{R})$ *is the set*[6]*:*

$$\bigcup_{(pr,sf)\in Dec(str_r)} Abs(pr,\mathcal{R}).AbsC(sf.str',\mathcal{R})$$

[6] For two sets E and F, $E.F = \{e.f \mid e \in E, f \in F\}$. If $E = \emptyset$ or $F = \emptyset$ then $E.F = \emptyset$.

Roughly speaking, $AbsC(str, \mathcal{R})$ denotes the set of all suspension traces composed of abstract observations that are abstractions of *str* w.r.t. $\mathcal{R}$. Let us observe that if the trace *str* cannot be correctly abstracted then its associated set of abstract suspension traces is empty.

Let us point out that for any G_c obtained by applying iteratively Definition 4 for all refinement pairs of $\mathcal{R}$ on an IOSTS G_a, the two following lemmas hold:

Lemma 1. *With $G_c = ref(\mathcal{R}, G_a)$, for all $str_a \in STr(G_a)$ s.t. there exists str_c with $str_a \in AbsC(str_c, \mathcal{R})$ then $str_c \in STr(G_c)$.*

Intuitively, Lemma 1 holds because by construction, Definition 4 ensures that for all $str_a \in STr(G_a)$, $STr(G_c)$ contains all suspension traces str_c such that $str_a \in AbsC(str_c, \mathcal{R})$.

Lemma 2. *With $G_c = ref(\mathcal{R}, G_a)$, for any $str_c \in STr(G_c)$ such that $AbsC(str_c, \mathcal{R}) \neq \emptyset$, there exists $str_a \in STr(G_a)$ s.t. $str_a \in AbsC(str_c, \mathcal{R})$.*

By Definition 4 any suspension trace str_c of G_c such that $AbsC(str_c, \mathcal{R}) \neq \emptyset$ is either also a suspension trace of G_a, and in that case $str_c \in AbsC(str_c, \mathcal{R})$, or a suspension trace of a path p_c of G_c which can be built from a path p_a of G_a by replacing in p_a transitions involving input actions to be refined by a complete path (of some suspension trace) of the IOSTS defined in the corresponding refinement pairs RP. For each such complete path p appearing in the definition of p_c and associated suspension trace σ, one can build a suspension trace str_a of p_a by replacing σ by any input action $c?v$ where c is the refined channel of RP and v belongs to $(\sigma, p)(\chi)$. This ensures that $str_a \in AbsC(str_c, \mathcal{R})$. Lemmas 1 and 2 are useful to prove Theorem 1.

We now define conformance up to a family of refinement pairs.

Definition 9 (Conformance up to refinement). *Let SUT be a system under test s. t. $Interface(SUT) \subseteq Interface(G_a) \cup C_{\mathcal{R}}$. SUT is $ioco_{\mathcal{R}}$-conform to G_a iff for any $str \in STr(SUT)$:*

- *if $AbsC(str, \mathcal{R}) \cap STr(G_a) \neq \emptyset$ then there exists $str_a \in AbsC(str, \mathcal{R}) \cap STr(G_a)$ s.t. if there exists act of the form $c!v$ or $\delta!$ with $str.act \in STr(SUT)$, then $str_a.act \in STr(G_a)$,*
- *if $AbsC(str, \mathcal{R}) = \emptyset$ and there exists $str.str' \in STr(SUT)$ s.t. $AbsC(str.str', \mathcal{R}) \cap STr(G_a) \neq \emptyset$ then for any suspension trace $str.str'' \in STr(SUT)$, str'' has no prefix of the form $\delta_m.act$ where $\delta_m \in \{\delta!\}^*$ and act is of the form $c!v$.*

The first item corresponds to situations in which *str* can be abstracted in the form of some suspension traces of G_a. In such a case at least one of them can be extended in G_a by any output that extends *str* in SUT. The second item corresponds intuitively to situations in which *str* ends by a sequence of concrete inputs which does not concretize an abstract one but it is possible to extend *str* in SUT by concrete inputs in order to form a trace that can be abstracted. In such a case, it is required that no output can occur until *str* is completed.

We now state the following theorem which relates *ioco*-conformance and $ioco_{\mathcal{R}}$-conformance.

Theorem 1. *SUT is ioco-conform to $ref(\mathcal{R}, G_a)$ iff SUT is ioco$_{\mathcal{R}}$-conform to G_a.*

4 Test Purpose Concretization

We illustrate by means of an example how to adapt our testing framework ([2], [1]) to take into account concretization through action refinement. In [2], test cases are extracted from so-called *test purposes* which are tree-like structures from the so-called *symbolic execution tree* of the IOSTS-model. The symbolic execution tree is composed of *symbolic extended states* and transitions between symbolic extended states. Paths in the symbolic execution tree denote executions of the model. A symbolic extended state is a triple (q, π, σ) structuring three pieces of information relatively to the path (*i.e.* execution) leading to it: (1) the reached state q of the model; (2) values assigned to attributes at this step of the execution (in the form of a substitution $\sigma : A \rightarrow T_{\Omega}(V_{froz})$ [7]); (3) constraints over symbolic terms assigned to attributes in the form of a formula π called the *path condition* [8]. Each transition *st* of the symbolic execution tree (*symbolic transition* for short) corresponds to the execution of a transition *tr* of the model whose source (*resp.* target) is the state introduced in the symbolic extended state at the source (*resp.* target) of the symbolic transition. Moreover, *st* is labeled by a symbolic action obtained by replacing terms occurring in $act(tr)$ by their symbolic values. Characterizing a test purpose simply consists in choosing behaviors to be tested (*i.e.* paths) in the symbolic execution tree by labeling their final symbolic extended states with the '*accept*' flag.

Example 4. Fig. 4 contains a test purpose TP_a extracted from the AVM of Example 1. Symbolic extended states labeled by $\odot$ are targets of paths which are outside of the behavior to be tested. The path to be tested denotes the following behavior:

pay at least 2 *units*[9]*; the AVM requires a choice for the drink to be served; ask for a coffee; receive the coffee.*

In [2], we have proposed an algorithm to extract test cases from such test purposes. The appliance of that algorithm requires that the IOSTS-model and the SUT share a common interface. If we consider as SUT an actual vending machine whose associated Interface is the one of the CVM of Example 2, the abstract test purpose TP_a cannot obviously be used directly: it has to be refined.

Intuitively, in order to define a test purpose at the good level of abstraction from TP_a, our goal is to characterize a test purpose in which paths to be tested concretize those characterized in TP_a. In TP_a there is only one path to be tested. We extract from TP_a the sequence of consecutive transitions of AVM of Fig. 1

[7] Values assigned to variables by σ are denoted by terms over frozen variables (chosen in a set of frozen variables V_{froz}) whose assignments result either from inputs or from execution of substitutions introduced in transitions of the IOSTS-model.

[8] Those constraints are deduced from guards of transitions executed to reach the symbolic extended state.

[9] Provided that m is initialized to 0.

$$\begin{tikzcd}
& \mathit{init} = (q_0, \mathit{true}, \sigma_0) \arrow[d, "\mathit{money}?x_1"] & & \\
& \eta_0 = (q_1, \mathit{true}, \sigma_1) \arrow[dl, "\mathit{screen}!\text{“}\mathit{more\ money}\text{“}"'] \arrow[dr, "\mathit{screen}!\text{“}\mathit{a\ drink}?\text{“}"] & & \\
\eta_1' = (q_0, \pi_1, \sigma_1) \odot & & \eta_1 = (q_2, \pi_0, \sigma_2) \arrow[d, "\mathit{drink}?B_1"] & \\
& & \eta_2 = (q_3, \pi_0, \sigma_3) \arrow[dl, "\mathit{drink}!''\mathit{tea}''"'] \arrow[dr, "\mathit{drink}!\text{“}\mathit{coffee}''"] & \\
& \eta_3 = (q_0, \pi_2, \sigma_3) \odot & & \eta_4 = (q_0, \pi_3, \sigma_3)\ \texttt{accept}
\end{tikzcd}$$

$$\begin{aligned}
\sigma_0 &= \{m \to m_0, x \to x_0, B \to B_0\} \\
\sigma_1 &= \{m \to m_0 + x_1, x \to x_1, B \to B_0\} \\
\sigma_2 &= \{m \to m_0 + x_1 - 2, x \to x_1, B \to B_0\} \\
\sigma_3 &= \{m \to m_0 + x_1 - 2, x \to x_1, B \to B_1\}
\end{aligned}$$

$$\begin{aligned}
\pi_0 &= (m_0 + x_1) \geq 2 \\
\pi_1 &= (m_0 + x_1) < 2 \\
\pi_2 &= \pi_0 \wedge (B_1 = \mathit{false}) \\
\pi_3 &= \pi_0 \wedge (B_1 = \mathit{true})
\end{aligned}$$

Fig. 4. Abstract test purpose TP_a

which have to be executed in order to reach the symbolic extended state labeled by *accept*. This sequence is a path p of AVM which relates states q_0, q_1, q_2, q_3, q_0 and whose transition from q_3 to q_0 is the one associated to the output action $\mathit{drink}!''\mathit{coffee}''$. In the following we need to differentiate several occurrences of a state or of a transition in p. To reach this purpose we construct a set of transitions T_p containing possibly several copies of transitions of AVM used to define p (one copy per occurrence). Practically, the source state and target state of a transition tr of p are named in a different manner than the one used in AVM of Fig. 1: this is done by using symbolic extended states as source and target states. For example to represent the occurrence of the transition $(q_0, \mathit{money}?x, \mathit{true}, [m := m+x], q_1)$ in p, the transition $(\mathit{init}, \mathit{money}?x, \mathit{true}, [m := m + x], \eta_0)$ belongs to T_p.

The following phase consists in defining an IOSTS whose associated set of transitions is T_p. We define the IOSTS $G_{TP_a} = (Q_p, q_0, T_p)$ where Q_p is the set of all source and target states of all transitions of T_p.

Example 5. The IOSTS G_{TP_a} associated to TP_a is depicted in Fig. 5.

$$\begin{tikzcd}
\mathit{init} \arrow[r, "\mathit{money}?x_1", "m := m + x"'] & \eta_0 \arrow[r, "{[m \geq 2]\ \mathit{screen}!\text{“}\mathit{a\ drink}?''}", "m := m - 2"'] & \eta_1 \arrow[r, "\mathit{drink}?B"] & \eta_2 \arrow[r, "{[B = \mathit{true}]\ \mathit{drink}!\text{“}\mathit{coffee}''}"] & \eta_4
\end{tikzcd}$$

Fig. 5. G_{TP_a} for the test purpose of Fig. 4

A set of distinguished symbolic extended states are those labeled by *accept* in the test purpose. In Fig. 4 there is only one such state: η_4. We note $\mathit{final}(TP_a) = \{\eta_4\}$ this set. Now let us note RP the refinement pair described in Example 2 and used to define the $CVM = (AVM, \{RP\})$. We note G_{SUT} the IOSTS $(G_{TP_a}, \{RP\})$. G_{SUT} characterizes all finite paths that concretize p through refinement pairs of RPs. The interface of G_{SUT} is the one of the actual vending

$$\begin{tikzcd}[column sep=small]
& (init, true, \sigma_0) \arrow[d, "\tau"'] & \\
& ((e,tr), true, \sigma_0) \arrow[d, "coin?y_1"'] & \\
& ((s,tr), y_1 \in \{1,2\}, \sigma_1) \arrow[dl, "m := m_0 + y_1"'] \arrow[dr, "coin?y_2"] & \\
(\eta_0, y_1 \in \{1,2\}, \sigma_2) \arrow[d, "screen!''a\ drink?``"'] & & ((s,tr), y_1 \in \{1,2\} \wedge y_2 \in \{1,2\}, \sigma_3) \arrow[d, "m := m_0 + y_1 + y_2"] \\
(\eta_1, y_1 \in \{1,2\} \wedge m_0 + y_1 \geq 2, \sigma_5) \arrow[d, "drink?B_1"'] & & (\eta_0, y_1 \in \{1,2\} \wedge y_2 \in \{1,2\}, \sigma_4) \arrow[d, "screen!''a\ drink?``"] \\
(\eta_2, y_1 \in \{1,2\} \wedge m_0 + y_1 \geq 2, \sigma_7) \arrow[d, "drink!``coffee''"'] & & (\eta_1, y_1 \in \{1,2\} \wedge y_2 \in \{1,2\} \wedge m_0 + y_1 + y_2 \geq 2, \sigma_6) \arrow[d, "drink?B_1"] \\
\begin{array}{c}(\eta_4, y_1 \in \{1,2\} \wedge m_0 + y_1 \geq 2 \wedge B_1 = true, \sigma_7)\\ \texttt{accept}\end{array} & & (\eta_2, y_1 \in \{1,2\} \wedge y_2 \in \{1,2\} \wedge m_0 + y_1 + y_2 \geq 2, \sigma_8) \arrow[d, "drink!''coffee''"] \\
& & \begin{array}{c}(\eta_4, y_1 \in \{1,2\} \wedge y_2 \in \{1,2\} \wedge m_0 + y_1 + y_2 \geq 2 \wedge B_1 = true, \sigma_8)\\ \texttt{accept}\end{array}
\end{tikzcd}$$

$\sigma_0 = \{m \to m_0, x \to x_0, y \to y_0, B \to B_0\}$ $\sigma_1 = \{m \to m_0, x \to y_1, y \to y_1, B \to B_0\}$
$\sigma_2 = \{m \to m_0 + y_1, x \to y_1, y \to y_1, B \to B_0\}$ $\sigma_3 = \{m \to m_0 + y_1, x \to y_2, y \to y_2, B \to B_0\}$
$\sigma_4 = \{m \to m_0 + y_1 + y_2, x \to y_2, y \to y_2, B \to B_0\}$ $\sigma_5 = \{m \to m_0 + y_1 - 2, x \to y_1, y \to y_1, B \to B_0\}$
$\sigma_6 = \{m \to m_0 + y_1 + y_2 - 2, x \to y_2, y \to y_2, B \to B_0\}$ $\sigma_7 = \{m \to m_0 + y_1 - 2, x \to y_1, y \to y_1, B \to B_1\}$
$\sigma_8 = \{m \to m_0 + y_1 + y_2 - 2, x \to y_2, y \to y_2, B \to B_1\}$

Fig. 6. Concrete test purpose

machine. Therefore it can be used to define test purposes for testing the actual vending machine by means of our algorithm. The set of concrete test purposes associated to TP_a contains all test purposes defined by symbolically executing G_c and labeling only symbolic extended states whose associated state is in $final(TP_a)$.

Example 6. In Fig. 6 we show a concrete test purpose associated to the abstract test purpose of Example 4. The variable y can take two values to denote the two possible coins that can be entered in the CVM: 1 for coins of 1 unit and 2 for coins of 2 units. It refines the behavior characterized in the test purpose of Fig. 4 by two concrete behaviors which correspond both to enter 2 units in the form of either one coin of two units (left) or two coins of one unit (right).

5 Conclusion

We have defined a model-based testing framework incorporating symbolic action refinement. Our framework is built as an extension of the symbolic conformance testing theory given in [2]. A refinement pair is made of a channel name c and an IOSTS with a final state and a frozen variable χ to be substituted by the targeted attribute variable. This denotes the capacity of refining any reception on a variable x through c by any behavior of the refining IOSTS leading to the final state provided that the variable χ has been first substituted by the variable x. These concrete behaviors are composed of input actions on concrete channels or of internal actions and must define the variable χ. Under theses hypotheses, we have defined the concretization of an abstract IOSTS w.r.t. a family of refinement pairs. The classical *ioco*-relation underlying most conformance testing frameworks has been relaxed by associating to each concrete observable trace of an implementation all possible abstract observable traces which are compatible with the given family of refinement pairs. Thus, the conformance relation has been parameterized by the considered family of refinement pairs. We have then explained on an example how we can derive from test purposes extracted from the abstract specification some concrete test purposes sharing with the implementation the same interface and unfolding the abstract behaviors selected in the abstract test purpose w.r.t. refining IOSTS.

We are currently implementing this approach in the *AGATHA* tool developed at CEA LIST. This integration will be used in the frame of the RNTL French project EDEN2.

References

1. Faivre, A., Gaston, C., Le Gall, P.: Symbolic model based testing for component oriented systems. In: Petrenko, A., Veanes, M., Tretmans, J., Grieskamp, W. (eds.) TestCom/FATES 2007. LNCS, vol. 4581, pp. 90–106. Springer, Heidelberg (2007)
2. Gaston, C., Le Gall, P., Rapin, N., Touil, A.: Symbolic execution techniques for test purpose definition. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, Springer, Heidelberg (2006)
3. Gorrieri, R., Rensink, A.: Handbook of process algebra. In: Ch. Action refinement, pp. 1047–1147. Elsevier, Amsterdam (2001)
4. King, J.-C.: A new approach to program testing. In: Proceedings of the international conference on Reliable software, Los Angeles, California, vol. 21-23, pp. 228–233 (1975)
5. Loeckx, J., Ehrich, H.-D., Wolf, M.: Specification of abstract data types. John Wiley & Sons, Inc., New York (1997)
6. Rapin, N., Le Gall, P., Touil, A.: Symbolic execution techniques for refinement testing. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 131–148. Springer, Heidelberg (2007)
7. Tretmans, J.: Test generation with inputs, outputs, and quiescence. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 127–146. Springer, Heidelberg (1996)
8. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software - Concepts and Tools 17(3), 103–120 (1996)
9. van der Bijl, H.M., Rensink, A., Tretmans, G.J.: Action refinement in conformance testing. In: Khendek, F., Dssouli, R. (eds.) TestCom 2005. LNCS, vol. 3502, pp. 81–96. Springer, Heidelberg (2005)

Implementation Relations for the Distributed Test Architecture*

Robert M. Hierons[1], Mercedes G. Merayo[1], and Manuel Núñez[2]

[1] Department of Information Systems and Computing, Brunel University
Uxbridge, Middlesex, UB8 3PH United Kingdom
rob.hierons@brunel.ac.uk, mgmerayo@fdi.ucm.es
[2] Universidad Complutense de Madrid, Madrid, Spain
mn@sip.ucm.es

Abstract. Some systems interact with their environment at a number of physically distributed interfaces called ports. When testing such a system under test (SUT) it is normal to place a local tester at each port and the local testers form a local test case. If the local testers cannot interact with one another and there is no global clock then we are testing in the distributed test architecture. In this paper we explore the effect of the distributed test architecture when testing an SUT against an input output transition system, adapting the **ioco** implementation relation to this situation. In addition, we define what it means for a local test case to be deterministic, showing that we cannot always implement a deterministic global test case as a deterministic local test case. Finally, we show how a global test case can be mapped to a local test case.

1 Introduction

If the system under test (SUT) has physically distributed interfaces, called ports, then in testing we place a tester at each port. If we are applying black-box testing, these testers cannot communicate with each other, and there is no global clock then we are testing in the distributed test architecture [1]. It is known that the use of the distributed test architecture reduces test effectiveness when testing from a deterministic finite state machine (DFSM) and this topic has received much attention (see, for example, [2,3,4,5,6]). It is sometimes possible to use an external network, through which the testers can communicate, to overcome the problems introduced when testing from a DFSM. However, there are costs associated with deploying such a network and it is not always possible to run test cases that have timing constraints [7].

Previous work on testing in the distributed test architecture has focussed on testing from DFSMs, two effects being identified. First, *controllability* problems

* This research was carried out while the first author was visiting Universidad Complutense de Madrid under a grant *Programa de visitantes distinguidos e investigadores extranjeros en la UCM (Grupo Santander)*. Research partially supported by the Spanish MEC project WEST/FAST (TIN2006-15578-C02-01) and the Marie Curie project MRTN-CT-2003-505121/TAROT.

may occur, where a tester cannot know when to apply an input. Let us suppose, for example, that a test case starts with input x_p at port p, this should lead to output y_p at p only and this is to be followed by input x_q at $q \neq p$. The tester at q cannot know when x_p has been applied, since it does not observe either the input or output from this transition. The second issue is that there may be *observability* problems that can lead to fault masking. Let us suppose, for example, that a test case starts with input x_p at p, this is expected to lead to output y_p at p only, this is to be followed by input x_p at p and this should lead to output y_p at p and y_q at $q \neq p$. The tester at p expects to observe $x_p y_p x_p y_p$ and the tester at q expects to observe y_q. This is still the case if the SUT produces y_p and y_q in response to the first input and y_p in response to the second input: Two faults have masked one another in the sequences used but could lead to failures in different sequences. Work on testing in the distributed test architecture has largely concerned finding test sequences without controllability or observability problems (see, for example, [2,3,4,6]) but recent work has characterized the effect of the distributed test architecture on the ability of testing to distinguish between a DFSM specification and a DFSM implementation [8].

While DFSMs are appropriate for modelling or specifying several important classes of system, they are less expressive than input output transition systems (IOTSs). For example, an IOTS can be nondeterministic and this is potentially important since distributed systems are often nondeterministic. In addition, in a DFSM input and output alternate and this need not be the case in IOTS. This paper investigates the area of testing from an IOTS in the distributed test architecture. For the sake of clarity, we focus on the case where there are two ports U and L, but our framework can be easily extended to cope with more ports. The paper explores the concept of a local test case (t_U, t_L), in which t_U and t_L are local testers at ports U and L respectively, and how we can assign verdicts. We adapt the well known **ioco** implementation relation [9,10] by defining a new implementation relation **dioco** and prove that i **dioco** s for SUT i and specification s if and only if i can fail a certain test run when testing in the distributed test architecture. While **ioco** has been adapted in a number of ways (see, for example, [11,12,13,14,15,16,17]), as far as we know this is the first paper to define an implementation relation based on **ioco** for testing from an IOTS in the distributed test architecture. However, an implementation relation **mioco** has been defined for testing from an IOTS with multiple ports when there is a single tester that controls and observes all of the ports [18].

Interestingly, **ioco** and **dioco** are incomparable if we do not require the specification to be input enabled but otherwise we have that **dioco** is weaker that **ioco**. We define what it means for a local test case (t_U, t_L) to be deterministic and show that there are deterministic global test cases that cannot be implemented using deterministic local test cases. We also show how a global test case t can be mapped to a local test case that implements t, where this is possible. In effect, the notion of a local test case being deterministic captures what it means for a test case to be controllable while **dioco** describes the ability to distinguish between processes and so captures observability problems.

This paper is structured as follows. In Section 2 we give preliminary material and in Section 3 we define local test cases and what it means to pass or fail a test run. Section 4 gives the new implementation relation **dioco** and Section 5 defines what it means for a local test case to be deterministic and shows how given a test case t we can find a local test case that implements t. In Section 6 conclusions are drawn.

2 Preliminaries

In this section we present the main concepts used in the paper. First, we define input output transition systems and notation to deal with sequences of actions that can be performed by a system. After that we will comment on the main differences, with respect to *classical* testing, when testing in the distributed architecture.

2.1 Input Output Transition Systems

An input output transition system is a labelled transition system in which we distinguish between input and output. We use this formalism to define processes.

Definition 1. *An input output transition system s, in short IOTS, is defined by (Q, I, O, T, q_{in}) in which Q is a countable set of states, $q_{in} \in Q$ is the initial state, I is a countable set of inputs, O is a countable set of outputs, and $T \subseteq Q \times (I \cup O \cup \{\tau\}) \times Q$, where τ represents internal (unobservable) actions, is the transition relation. A transition (q, a, q') means that from state q it is possible to move to state q' with action $a \in I \cup O \cup \{\tau\}$. We let $\mathcal{IOTS}(I, O)$ denote the set of IOTSs with input set I and output set O.*

We say that the state $q \in Q$ is quiescent *if from q it is not possible to produce output without first receiving input. We say that the process s is* input enabled *if for all $q \in Q$ and $?i \in I$ there is some $q' \in Q$ such that $(q, ?i, q') \in T$. We say that the process s is* output-divergent *if it can reach a state in which there is an infinite loop that contains outputs and internal actions only.*

Given action a and process s, $a.s$ denotes the process that performs a and then becomes s. Given a countable set S of processes, $\sum S$ denotes the process that can nondeterministically choose to be any one of the processes in S. Given processes $s_1, \ldots, s_k$, we have that $s_1||s_2||\ldots||s_k$ is the process in which $s_1, \ldots s_k$ are composed in parallel and interact by synchronizing on common labels.

In this paper we use I for input and O for output rather than I and U, which are traditionally used in the work on IOTS. This is because U denotes the upper tester in protocol conformance testing and so, in this paper, U and L are used to denote ports. In order to distinguish between input and output we usually precede the name of a label with ? if it is an input and ! if it is an output. We assume that implementations are input enabled. This is a usual condition to ensure that implementations will accept any input provided by the tester. In addition we also assume that all the processes considered in this paper, either implementations or models, are not output-divergent.

It is normal to assume that it is possible for the tester to determine when the SUT is quiescent and this is represented by δ. We can extend the transition relation to include in quiescent states *transitions* labelled by δ.

Definition 2. *Let (Q, I, O, T, q_{in}) be an IOTS. We can extend T, the transition relation, to T_δ by adding the transition (q, δ, q) for each quiescent state q. We let $\mathcal{A}ct$ denote the set of observable actions, that is, $\mathcal{A}ct = I \cup O \cup \{\delta\}$. A* trace *is an element of $\mathcal{A}ct^*$. Given a trace σ we let $in(\sigma)$ denote the sequence of inputs from σ. This can be recursively defined by the following in which ϵ is the empty sequence: $in(\epsilon) = \epsilon$, if $z \in I$ then $in(z\bar{z}) = zin(\bar{z})$ and if $z \notin I$ then $in(z\bar{z}) = in(\bar{z})$.*

Traces are often called *suspension traces*, since they can include quiescence, but since these are the only types of traces we consider we simply call them traces. Let us remark that traces are in $\mathcal{A}ct^*$ and so cannot contain τ. The following is standard notation in the context of **ioco** (see, for example, [9]).

Definition 3. *Let $s = (Q, I, O, T, q_{in})$ be an IOTS. We use the following notation.*

1. *If $(q, a, q') \in T_\delta$, for $a \in \mathcal{A}ct \cup \{\tau\}$, then we write $q \xrightarrow{a} q'$.*
2. *We write $q \overset{a}{\Longrightarrow} q'$, for $a \in \mathcal{A}ct$, if there exist $q_0, \ldots, q_m$ and $k \geq 0$ such that $q = q_0$, $q' = q_m$, $q_0 \xrightarrow{\tau} q_1, \ldots q_{k-1} \xrightarrow{\tau} q_k$, $q_k \xrightarrow{a} q_{k+1}$, $q_{k+1} \xrightarrow{\tau} q_{k+2}, \ldots, q_{m-1} \xrightarrow{\tau} q_m$.*
3. *We write $q \overset{\epsilon}{\Longrightarrow} q'$ if there exist $q_1, \ldots, q_k$, for $k \geq 1$, such that $q = q_1$, $q' = q_k$, $q_1 \xrightarrow{\tau} q_2, \ldots q_{k-1} \xrightarrow{\tau} q_k$.*
4. *We write $q \overset{\sigma}{\Longrightarrow} q'$ for $\sigma = a_1 \ldots a_m \in \mathcal{A}ct^*$ if there exist $q_0, \ldots, q_m$, $q = q_0$, $q' = q_m$ such that for all $1 \leq i < m$ we have that $q_i \overset{a_{i+1}}{\Longrightarrow} q_{i+1}$.*
5. *We write $s \overset{\sigma}{\Longrightarrow}$ if there exists q' such that $q_{in} \overset{\sigma}{\Longrightarrow} q'$ and we say that σ is a* trace *of s.*

Let $q \in Q$ and $\sigma \in \mathcal{A}ct^*$ be a trace. We consider

1. *q **after** $\sigma = \{r \in Q | q \overset{\sigma}{\Longrightarrow} r\}$*
2. *$out(q) = \{!O \in O | q \overset{!O}{\Longrightarrow}\}$*

The last function can be extended to deal with sets in the expected way: Given $Q' \subseteq Q$ we define $out(Q') = \cup_{q \in Q'} out(q)$.

We say that s is deterministic *if for every trace $\sigma \in \mathcal{A}ct^*$ we have that $out(q_{in}$ **after** $\sigma)$ contains at most one element.*

Let us remark that for any state q we have $q \Rightarrow q$ and that $q \xrightarrow{a} q'$ implies $q \overset{a}{\Longrightarrow} q'$. Let us also note that for all process s the empty sequence ϵ is a trace of s. While we do not require the models or implementations to be deterministic it is normal to use test cases that are deterministic. Next we present the standard implementation relation for testing from an IOTS [9,10].

Definition 4. *Given processes i and s we have that i **ioco** s if for every trace σ of s we have that $out(i$ **after** $\sigma) \subseteq out(s$ **after** $\sigma)$.*

2.2 Multi-port Input Output Transition Systems

There are two standard test architectures [1], shown in Figure 1. In the local test architecture a global tester interacts with all of the ports of the SUT. In the distributed test architecture there are ports through which a system interacts with its environment. We focus on the case where there are two ports, traditionally called U and L for the ports connected to the upper and lower testers, respectively. In this paper we usually use the term IOTS for the case where there are multiple ports and when there is only one port we use the term single-port IOTS.

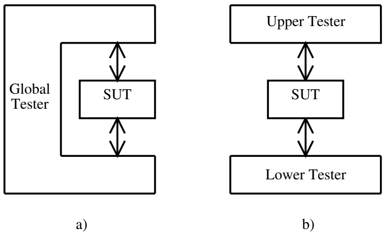


Fig. 1. The local and distributed test architectures

For an IOTS (Q, I, O, T, q_{in}) with ports U and L we partition the set I into sets I_U and I_L of inputs that can be received at U and L respectively and we define a set O_U of output that can be received at U and a set O_L of output that can be received at L. Each element of O is thus in $(O_U \cup \{-\}) \times (O_L \cup \{-\})$ in which $-$ denotes empty output and $(-,-)$ is not an element of O. We assume that the sets I_U, I_L, O_U, O_L are pairwise disjoint. Given $y = (!O_U, !O_L) \in O$ we let $y|_U$ denote $!O_U$ and $y|_L$ denote $!O_L$. If an output tuple has output $!O_p$ at one port p only then we often simply represent this as $!O_p$.

In order to apply tests, if we have a *global tester* that observes all of the ports then it observes a trace in $\mathcal{A}ct^*$, called a *global trace.* Given port p and a global trace $\bar{z}$ we let $\pi_p(\bar{z})$ denote the projection of $\bar{z}$ onto p and this is called a *local trace.* This is defined by the following rules in which $q \neq p$:

1. $\pi_p(\epsilon) = \epsilon$
2. if $z \in (I_p \cup O_p \cup \{\delta\})$ then $\pi_p(z\bar{z}) = z\pi_p(\bar{z})$
3. if $z = (!O_U, !O_L)$, $!O_p \neq -$, then $\pi_p(z\bar{z}) =!O_p\pi_p(\bar{z})$
4. if $z \in (I_q \cup O_q)$ or $z = (!O_U, !O_L)$ for $!O_p = -$ then $\pi_p(z\bar{z}) = \pi_p(\bar{z})$.

Given traces σ and σ' we write $\sigma \sim \sigma'$ if σ and σ' cannot be distinguished when making local observations, that is, $\pi_U(\sigma) = \pi_U(\sigma')$ and $\pi_L(\sigma) = \pi_L(\sigma')$. When testing in the distributed test architecture, each local tester observes a sequence of actions at its port. We cannot distinguish between two traces if they have the same projections at each port. That is why we cannot compare traces by equality but by considering the permutations of traces defined by using $\sim$.

3 Test Cases for the Distributed Test Architecture

Before discussing implementation relations for the distributed test architecture it is necessary to adapt the standard definition of a test case. In this paper a test case t is an IOTS with the same input and output sets as s. We also have that s and t synchronize on values and we say that such a test case is a *global test case*. Thus, if s is in $\mathcal{IOTS}(I, O)$ then every global test case for s is in $\mathcal{IOTS}(I, O \cup \{\delta\})$. As usual, in each state a test case must be able to accept any output from the SUT. Given I and O, the simplest test case is thus a process, that will be called $\perp$, which cannot send input to the SUT and thus whose traces are all elements of $(O \cup \{\delta\})^*$. We let $\perp_p$ denote the null process for port p, whose set of traces is $(O_p \cup \{\delta\})^*$.

As usual, a *test case* is an IOTS with a finite set of states. A test case t is *deterministic* if for every sequence σ we have that t **after** σ contains at most one process and t does not have a state where there is more than one input it can send to the SUT. It is normal to require that test cases are deterministic. In the distributed test architecture we place a local tester at each port and the tester at port p only observes the behaviour at p. We therefore require that a local test case contains a process for each port rather than a single process.

Definition 5. *Let $s \in \mathcal{IOTS}(I, O)$ be an IOTS with ports U and L. A* local test case *is a pair (t_U, t_L) of* local testers *in which:*

1. *t_U is a deterministic test case with input set I_U and output set $O_U \cup \{\delta\}$ and so is in $\mathcal{IOTS}(I_U, O_U \cup \{\delta\})$. In addition, if $t_U \overset{\sigma}{\Longrightarrow} t'_U$ for some σ then t'_U must be able to accept any value from $O_U \cup \{\delta\}$.*
2. *t_L is a deterministic test case with input set I_L and output set $O_L \cup \{\delta\}$ and so is in $\mathcal{IOTS}(I_L, O_L \cup \{\delta\})$. In addition, if $t_L \overset{\sigma}{\Longrightarrow} t'_L$ for some σ then t'_L must be able to accept any value from $O_L \cup \{\delta\}$.*

Since observations are made locally at each port it is natural to give each of the processes t_U and t_L a *pass* state and a *fail* state. However, this approach leads to a loss of precision as we may observe traces at U and L that are individually consistent with s but where, when the logs of these traces are brought together, we know that there has been a failure. We know that there has been a failure if no interleaving of the behaviours observed at the two ports is consistent with the specification s. Let us consider, for example, a process s in which input of $?i_U$ at U is either followed by $!O_U$ at U and then output $!O_L$ at L or by $!O'_U$ at U and then output $!O'_L$ at L. If we use a local test case that applies $?i_U$, the sequence $?i_U!O_U$ is observed at U and $!O'_L$ is observed at L then each local tester observes behaviour that is consistent with projections of traces of the specification s and yet the pair of traces is not consistent with any conforming implementation.

It might seem sensible to represent pass and fail in terms of states of the pair of local testers and have unique *pass* and *fail* states of this. However, an additional complication arises: Test effectiveness is not *monotonic*, that is, the application of an input sequence can reveal a failure but we could extend this input sequence in a manner so that we lose the ability to find a failure. To see this, let us consider the following situation:

1. A specification s that has $?i_U$ at U then it is possible to apply $?i_U$ again and this is followed by output $!O_L$ at L.
2. An implementation i that has $?i_U$ at U, followed by $!O_L$ at L and then it is possible to apply $?i_U$.

If the local tester at U applies $?i_U$ and then observes output, and the local tester at L just observes output then we observe a failure since i will produce $!O_L$ at L when it should not have. However, if the local tester at U applies $?i_U?i_U$ and then observes output, and the local tester at L just observes output then we do not observe a failure since testing will observe $?i_U?i_U$ at U as expected and $!O_L$ at L as expected. As a result, in order to define verdicts we will need to define a mapping from states of a local test case to verdicts. In order to simplify the exposition we avoid this complexity and use the specification s as an *oracle*, as explained below in Definition 7.

It is important to consider what observations local test case (t_U, t_L) can make. As usual, we assume that quiescence can be observed and so (t_U, t_L) can observe the SUT being quiescent.[1] A test run for (t_U, t_L) is a sequence of observations that can occur in testing the SUT i with (t_U, t_L), that is, any sequence of observations that can be made with $t_U||i||t_L$. As usual, we require that testing is guaranteed to terminate in finite time.

Before we formally define the concept of a test run, we introduce new notation regarding how t_U, i and t_L can change through sending messages to one another.

Definition 6. *Let i be a SUT and (t_U, t_L) be a test case. We define the transitions $\xrightarrow{a}_t$ and $\xRightarrow{\sigma}_t$ as follows:*

1. *For $a \in \mathcal{A}ct$, if $i \xrightarrow{a} i'$ and $t_U \xrightarrow{a} t'_U$ then $t_U||i||t_L \xrightarrow{a}_t t'_U||i'||t_L$.*
2. *For $a \in \mathcal{A}ct$, if $i \xrightarrow{a} i'$ and $t_L \xrightarrow{a} t'_L$ then $t_U||i||t_L \xrightarrow{a}_t t_U||i'||t'_L$.*
3. *For $a = (!O_U, !O_L) \in O_U \times O_L$, if $i \xrightarrow{a} i'$, $t_U \xrightarrow{!O_U} t'_U$, and $t_L \xrightarrow{!O_L} t'_L$ then $t_U||i||t_L \xrightarrow{a}_t t'_U||i'||t'_L$.*
4. *If $i \xrightarrow{\tau} i'$ then $t_U||i||t_L \xrightarrow{\tau}_t t_U||i'||t_L$.*
5. *If $t_U||i||t_L \xrightarrow{a}_t t'_U||i'||t'_L$ then we can write $t_U||i||t_L \xrightarrow{a}_t$.*
6. *If $\sigma = a_1 \dots a_k \in (\mathcal{A}ct \cup \{\tau\})^*$, $t^1_U = t_U$, $i^1 = i$, $t^1_L = t_L$, for $1 \leq j \leq k$ we have $t^j_U||i^j||t^j_L \xrightarrow{a_j}_t t^{j+1}_U||i^{j+1}||t^{j+1}_L$, $t^{k+1}_U = t'_U$, $i^{k+1} = i'$, $t^{k+1}_L = t'_L$, then $t_U||i||t_L \xrightarrow{\sigma}_t t'_U||i'||t'_L$.*
7. *If $t_U||i||t_L \xrightarrow{\sigma}_t t'_U||i'||t'_L$ then we can write $t_U||i||t_L \xrightarrow{\sigma}_t$.*
8. *If $\sigma = a_1 \dots a_k \in \mathcal{A}ct^*$, $\sigma_1 \in \tau^* a_1 \tau^* \dots \tau^* a_k \tau^*$ and $t_U||i||t_L \xrightarrow{\sigma_1}_t t'_U||i'||t'_L$ then we can write $t_U||i||t_L \xRightarrow{\sigma}_t t'_U||i'||t'_L$.*
9. *If $t_U||i||t_L \xRightarrow{\sigma}_t t'_U||i'||t'_L$ then we can write $t_U||i||t_L \xRightarrow{\sigma}_t$.*

[1] While the observation of quiescence in the distributed test architecture is slightly different, since it is possible to have inactivity at one port when there is activity at another, in practice there are few differences since in testing we know the test case being applied.

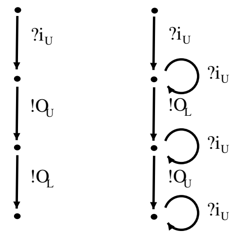


Fig. 2. Processes s_1 and i_1

We are now in a position to define what it means for an implementation to pass a test run with a local test case and thus to pass a local test case. We require a test run to end with i being quiescent. In order to see why we require this consider s_1 and i_1 shown in Figure 2. It is clear that i is a good implementation of s when considering the distributed test architecture. Nevertheless, if we consider the trace $?i_U!O_L$ of i_1 we find that this is not a permutation of a trace of s. However, testing cannot observe $?i_U!O_L$. This differs from the case where we observe global traces: if we can observe a global trace σ then we can construct the prefixes of σ. However, in general this cannot be done when testing in the distributed test architecture.

The following defines test runs with local test cases and what it means for the SUT to pass a test run and to pass a local test case.

Definition 7. *Let s be a specification, i be a SUT and (t_U, t_L) be a local test case. We introduce the following notation.*

1. *A trace σ is a* test run *for i with local test case (t_U, t_L) if there exists t'_U, t'_L, and i' such that $t_U||i||t_L \overset{\sigma}{\Longrightarrow}_t t'_U||i'||t'_L$ and $t'_U||i'||t'_L$ is quiescent.*
2. *Implementation i* passes *a test run σ with (t_U, t_L) for s if either of the following two conditions hold.*
   (a) There does not exist a trace σ' of s such that $in(\sigma) \sim in(\sigma')$.
   (b) there exists some $\sigma' \sim \sigma$ that is a trace of s.
3. *If i passes the test run σ, and s can be inferred from the context, we write $t_U||i||t_L \overset{\sigma}{\Longrightarrow} pass$. Otherwise we say i* fails *the test run σ and we write $t_U||i||t_L \overset{\sigma}{\Longrightarrow} fail$.*
4. *Implementation i* passes *test case (t_U, t_L) for s if i passes every possible test run of i with (t_U, t_L) for s and this is denoted $PASSES(i, t_U, t_L, s)$. Otherwise i* fails *(t_U, t_L) and this is denoted $FAILS(i, t_U, t_L, s)$.*

The first clause of the second item corresponds to σ not being in response to an input sequence $in(\sigma)$ such that the behaviour of s is defined for an input sequence that is indistinguishable from $in(\sigma)$ in the distributed test architecture. Moreover, while the second clause of the same item does not say that s must be quiescent after σ' this is implicit when i passes a local test case (t_U, t_L) since if i is quiescent after σ then i is quiescent after $\sigma\delta$ and so this is also a possible test run for i with (t_U, t_L).

Even if we bring together the logs from the testers at the end of a test run, the use of the distributed test architecture can reduce the ability of testing to distinguish between the specification s and the SUT i.

Proposition 1. *It is possible for an implementation to fail a test case if global traces are observed and yet pass the test case if only local traces are observed.*

Proof. Consider the specification s_1 and the implementation i_1 illustrated in Figure 2. Here the only sequence of actions in s_1 is $?i_U$ at U followed by $!O_U$ at U, then $!O_L$ at L. The implementation i has $?i_U$ at U followed by $!O_L$ at L, then $!O_U$ at U. So, clearly i_1 does not conform to s_1 under the **ioco** relation and will fail a global test case that applies $?i_U$. However, it is not possible for the local testers to observe the difference if we apply $?i_U$ since the tester at U will see the expected sequence of actions $?i_U!O_U$ and the tester at L will observe the expected sequence of actions $!O_L$.

4 A New Implementation Relation

We have seen that an implementation may pass a test case when we only make local observations and yet fail the test case if observations are made globally. This suggest that the implementation relation required when making local observations will differ from the implementation relation **ioco** used when making global observations. In this section we define a new implementation relation.

It might seem natural to define the new implementation relation so that the behaviour of the implementation i at a port p must conform to the behaviour of s at port p and this must hold for every port. However, we have already seen that there could be a trace σ such that $i \overset{\sigma}{\Longrightarrow}$, the projections of σ at ports U and L are individually consistent with traces of s and yet there is no $\sigma' \sim \sigma$ that is a trace of s. As a result, we require an implementation relation that compares pairs of local traces with global traces of the specification.

We want the new implementation relation, that we call **dioco**, to correspond to the ability of testing to determine when an implementation conforms to a given specification when testing in the distributed test architecture. We can define this implementation relation in terms of the traces in the implementation.

Definition 8. *Given specification s and implementation i we have that i **dioco** s if for every trace σ such that $i \overset{\sigma}{\Longrightarrow} i'$ for some i' that is quiescent, if there is a trace σ_1 of s such that $in(\sigma_1) \sim in(\sigma)$ then there exists a trace σ' such that $s \overset{\sigma'}{\Longrightarrow} s'$ and $\sigma' \sim \sigma$.*

The implementation relation **dioco** captures our notion of test run and failing a test run with a local test case.

Proposition 2. *Given a specification s and an implementation i, i **dioco** s if and only if for every local test case (t_U, t_L) we have that $PASSES(i, t_U, t_L, s)$.*

Proof. First let us assume that i **dioco** s and let (t_U, t_L) be a local test case. We require to prove that i passes all possible test runs with (t_U, t_L) for s. Let σ denote some trace that can be produced by a test run of i with (t_U, t_L). If there does not exist a trace σ' of s such that $in(\sigma) \sim in(\sigma')$ then i passes this test run by definition. We therefore assume that there is some such σ'. But, since i **dioco** s we must have that there is a trace σ' of s such that $\sigma' \sim \sigma$ and so i passes this test run with (t_U, t_L). It thus follows that i passes all possible test runs with (t_U, t_L) for s, as required.

Now let us assume that i passes all possible test runs and let σ be a trace such that $i \overset{\sigma}{\Longrightarrow} i'$ for some i' that is quiescent. Thus, we consider the situation in which there exists a trace σ_1 of s such that $in(\sigma_1) \sim in(\sigma)$ and we are required to prove that there exists some σ' such that $s \overset{\sigma'}{\Longrightarrow}$ and $\sigma' \sim \sigma$. Let $t_p = \pi_p(\sigma)$ for $p \in \{U, L\}$. Then clearly σ is a possible test run of i with local test case (t_U, t_L) and so the result follows from the fact that i must pass all possible test runs with (t_U, t_L).

The next result shows that the implementation relations **dioco** and **ioco** are incomparable.

Proposition 3. *There exist processes s and i such that i **dioco** s but not i **ioco** s. There also exist processes s and i such that i **ioco** s but not i **dioco** s.*

Proof. Consider the processes s_1 and i_1 given in Figure 2. It is clear that i_1 **dioco** s_1 since the local testers cannot distinguish between the traces $?i_U!O_U!O_L$ and $?i_U!O_L!O_U$. However, it is also clear that $out(i_1$ **after** $?i_U) \not\subseteq out(s_1$ **after** $?i_U)$ and so we do not have that i_1 **ioco** s_1.

Now consider processes s_2 and i_2 given in Figure 3. The only traces that i_2 and s_2 have in common are those in the sets δ^*, $\delta^*?i_L\delta^*$, and $\delta^*?i_L\delta^*?i_U\delta^*$ and for each of these i_2 and s_2 have the same set of possible outputs, δ^*. Thus, i_2 **ioco** s_2. However, local testers cannot distinguish between $?i_L?i_U$ and $?i_U?i_L$. We have that the second of these sequences leads to an output of $!O_U$ in i_2 and this is not allowed after $?i_L?i_U$ in s_2. As a result we do not have that i_2 **dioco** s_2, as required.

The use of the distributed test architecture reduces the ability of testing to observe behaviours of the SUT and thus it is natural to expect that there are cases where the resultant implementation relation allows an implementation i to conform to s even though we do not have that i **ioco** s. However, one would expect **dioco** to be strictly weaker than **ioco** since all observations that can be made locally can also be made globally. The example presented in Figure 3, and used in Proposition 3, shows that this is not the case. This example relies on s not being input enabled: We do not have i **dioco** s because the 'problematic' behaviour in i_2 occurs after a trace σ that is not in s_2 but such that there is a permutation $\sigma' \sim \sigma$ that is in s_2. The following results show that if the specification and implementation are input enabled then we obtain the expected result: **dioco** is strictly weaker than **ioco**.

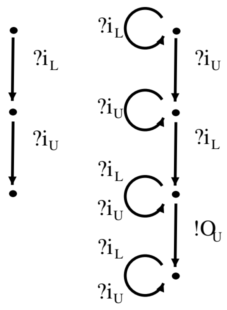


Fig. 3. Processes s_2 and i_2

Proposition 4. *If s and i are input enabled and i* **ioco** *s then every trace of i is also a trace of s.*

Proposition 5. *If s and i are input enabled then whenever we have that i* **ioco** *s it must be the case that i* **dioco** *s.*

Proof. Assume that i **ioco** s, $i \overset{\sigma}{\Longrightarrow} i'$ for some i' that is quiescent, and there is a trace σ_1 of s such that $in(\sigma_1) \sim in(\sigma)$. It is sufficient to prove that there is a trace σ' of s such that $\sigma' \sim \sigma$. This follows immediately from the fact that, by Proposition 4, σ is a trace of s.

While for input enabled specification s and implementation i we have that i **ioco** s implies i **dioco** s, the converse does not hold. In order to see this, consider the processes in Figure 4. It is clear that these are incomparable under **ioco** since if an output is produced then the value of this is determined by the initial input and the mapping from initial input to output is different in the two cases. However, in each case the two paths to the states from which there can be output simply require that there is at least one use of $?i_U$ and at least one use of $?i_L$. As a result, the sets of paths to the two states from which output can be produced are indistinguishable when observing events locally and so the two processes cannot be distinguished when testing in the distributed test architecture.

It is usual to insist that there is no 'unnecessary nondeterminism' and thus the tester cannot reach a state in which it could provide alternative inputs to the implementation (see, for example, [9]). This ensures that the test is controllable. In the next section we interpret this in the context of the distributed test architecture.

5 Deterministic Test Cases

Since test objectives are usually stated at the specification level it is natural to initially generate a global test case that achieves a given test objective. However, there is then the challenge of producing local testers that implement such a global

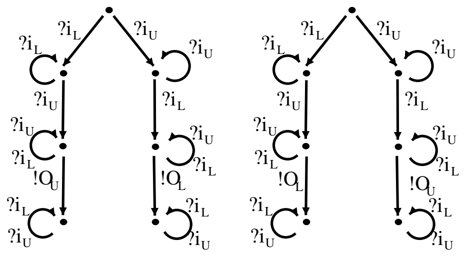


Fig. 4. Processes that are not related under **ioco**

test case. In this section we assume that a deterministic global test case t has been produced for testing an implementation i against specification s.

It is normal to use deterministic test cases and so for (t_U, t_L) we already assumed in Definition 5 that t_U and t_L are deterministic. However, it is possible that (t_U, t_L) is nondeterministic despite t_U and t_L being deterministic. A simple example of this is any pair (t_U, t_L) in which t_U and t_L both start by sending an input to the implementation: Even if t_U and t_L are deterministic, the order in which the implementation receives these inputs from t_U and t_L is not predictable. Nondeterminism occurs in (t_U, t_L) if both t_U and t_L can provide an input to the implementation at some point in a test run. In this section we define what it means for local test case (t_U, t_L) to be deterministic for a given specification s and show how we can map a global test case to a local test case.

It might seem desirable that a local test case is deterministic for any possible implementation. However, in Proposition 6 we will prove that for an input enabled specification if the interaction of a deterministic local test case (t_U, t_L) with an implementation i can lead to nondeterminism then there is a local test case (t'_U, t'_L) that i fails and shows how (t'_U, t'_L) can be constructed from (t_U, t_L). Thus, it is only necessary to consider traces that can be produced by the specification and test case interacting. Let us note that since the testers are local, this is equivalent to considering all traces that can be produced by the local test case combined with the specification and all permutations that preserve the traces at each port.

It is interesting to note that for any local test case (t_U, t_L) such that t_U and t_L both have the potential to send input to the SUT, there is some behaviour of a possible implementation that will lead to nondeterminism. This is the case since there must exist a possible trace σ_U at U after which t_U will send input to the SUT and a possible trace σ_L at L after which t_L will send input to the SUT and so there is nondeterminism if the SUT produces any trace $\sigma' \sim \sigma_U \sigma_L$ in testing. As a result, it is unreasonable to require more than for the local test case to be deterministic for all possible behaviours of the specification.

Definition 9. *Given a specification s we say that the local test case (t_U, t_L) is* deterministic for s *if there do not exist traces σ_1 and σ_2, with $\sigma_2 \sim \sigma_1$, and $a_1, a_2 \in I$, with $a_1 \neq a_2$, such that $t_U||s||t_L \xRightarrow{\sigma_1 a_1}_t$ and $t_U||s||t_L \xRightarrow{\sigma_2 a_2}_t$.*

Given a specification s, we let $\mathcal{T}_D(s)$ denote the set of local test cases that are deterministic for s.

Interestingly, this is similar to the notion of local choice defined in the context of MSCs [19]. We can now prove that given local test case (t_U, t_L) that is deterministic for specification s, the application of (t_U, t_L) to an implementation i can only be nondeterministic through an erroneous behaviour of i and there is a prefix of (t_U, t_L) that is capable of detecting this erroneous behaviour through local observations. The proof will use the notion of a prefix of a local test case (t_U, t_L), which is a local test case that can be generated from (t_U, t_L) by replacing one or more of the input states[2] of t_U and t_L by processes that can only receive output.

Proposition 6. *Let us suppose that s is an input enabled specification, i is an implementation, and $(t_U, t_L) \in \mathcal{T}_D(s)$ is a deterministic test case for s. If there exists σ_1 and σ_2, with $\sigma_2 \sim \sigma_1$, and $a_1, a_2 \in I$, with $a_1 \neq a_2$, such that $t_U||i||t_L \overset{\sigma_1}{\Longrightarrow}_t t'_U||i'||t'_L$, $t_U||i||t_L \overset{\sigma_2}{\Longrightarrow}_t t''_U||i''||t''_L$, $t'_U||i'||t'_L \overset{a_1}{\Longrightarrow}$, and $t''_U||i''||t''_L \overset{a_2}{\Longrightarrow}$ then there exists a prefix (t^1_U, t^1_L) of (t_U, t_L) such that i fails (t^1_U, t^1_L).*

Proof. By definition, since (t_U, t_L) is deterministic for s, $\sigma_1 \sim \sigma_2$ can be followed by different inputs when applying (t_U, t_L), we must have that $\neg(s \overset{\sigma'}{\Longrightarrow})$ for all $\sigma' \sim \sigma_1$. Let σ be the shortest prefix of σ_1 such that $t_U||i||t_L \overset{\sigma}{\Longrightarrow}_t t^1_U||i^1||t^1_L$ for some i^1 that is quiescent after σ and no $\sigma'_1 \sim \sigma$ is a trace of s. If $t^1_U||i^1||t^1_L$ cannot perform any actions then σ is a trace that can be produced by a test run of i with (t_U, t_L) and so the result follows from the fact that s is input enabled. Otherwise we have that (t^1_U, t^1_L) is in an input state and so we can define a prefix (t''_U, t''_L) of (t_U, t_L) by replacing the processes t^1_U and t^1_L, of t_U and t_L, by the processes $\perp_U$ and $\perp_L$, respectively, that cannot send input to the SUT. Then the interaction of i with (t''_U, t''_L) can lead to testing terminating with the trace σ and thus at least one tester observing a failure.

Given local test case (t_U, t_L), it is possible to produce the set of prefixes of (t_U, t_L) and use these in testing. If we do this then an implementation that passes all of these local test cases does not lead to nondeterminism when tested with (t_U, t_L). If we have a deterministic global test case t, we want to devise local testers t_U and t_L that implement t.

Definition 10. *A local test case (t_U, t_L)* implements *the global test case t for specification s if we have that the interaction between (t_U, t_L) and s can lead to trace σ if and only if the interaction between t and s can produce trace σ. More formally, for every trace σ we have that $t_U||s||t_L \overset{\sigma}{\Longrightarrow}$ if and only if $t||s \overset{\sigma}{\Longrightarrow}$.*

We can produce local testers by taking projections of a global test case t. In taking the projection of t at p, we will eliminate actions at port $q \neq p$.

[2] An input state of process s is a reachable state r that can perform an input, that is, there exists σ such that $s \overset{\sigma}{\Longrightarrow} r$ and an input $?i$ such that $r \xrightarrow{?i}$.

Definition 11. *Given global test case t and port p, we let $local_p(t)$ denote the local tester at p defined by the following rules.*

1. *If t is the null process $\perp$, that cannot send an input, then $local_p(t)$ is $\perp_p$.*
2. *If $a \in I_p \cup O_p \cup \{\delta\}$ then $local_p(a.s) = a.local_p(s)$*
3. *If $a \in I_q \cup O_q$, $q \neq p$, then $local_p(a.s) = local_p(s)$*
4. *If $a = (!O_U, !O_L)$, $!O_p \neq -$, then $local_p(a.s) = !O_p.local_p(s)$*
5. *If $a = (!O_U, !O_L)$, $!O_p = -$, then $local_p(a.s) = local_p(s)$*
6. *$local_p(t_1 + \ldots + t_k) = local_p(t_1) + \ldots + local_p(t_k)$.*

If we give the function $local_p$ a deterministic global test case then it returns the local testers we require.

Proposition 7. *Given deterministic global test case t and $t_p = local_p(t)$ for $p \in \{U, L\}$ we have that if the local test case (t_U, t_L) is deterministic then (t_U, t_L) implements t.*

Proof. We prove the result by induction on the size of t, which is the number of nodes of the tree formed from t. Clearly the result holds for the base case where t has only one node and so is the null test case $\perp$. Inductive hypothesis: We assume that the result holds for every global test case of size less than k, for $k > 1$, and t has size k. There are three cases to consider.

Case 1: t starts by sending input $?i$ to the SUT, that is, $t \xrightarrow{?i} t'$ for some t'. Since t is deterministic, t' is uniquely defined. Without loss of generality, $?i$ is at port U and $t_U \xrightarrow{?i} t'_U$ (the proof for $i?$ being at port L is similar). Let S be the set of processes that s can become after $?i$, that is, $S = \{s_1 | s \xrightarrow{?i} s_1\}$, and let us consider $s' = \sum S$. Clearly, t' and (t'_U, t_L) are deterministic for s' and the size of t' is less than k. Further, $t_L = local_L(t')$ and $t'_U = local_U(t')$ and so, by the inductive hypothesis, we have that a trace σ' can be produced by the interaction between t' and s' if and only if it can be produced by the interaction between (t'_U, t_L) and s'. A trace σ can be produced by the interaction between t and s if and only if σ is $?i\sigma'$ for some trace σ' that can be produced by an interaction between s' and t'. Since (t_U, t_L) is deterministic for s we also have that a trace σ can be produced by the interaction between (t_U, t_L) and s if and only if σ is $?i\sigma'$ for some trace σ' that can be produced by an interaction between s' and (t'_U, t_L). The result thus follows.

Case 2: t starts with an output from the SUT, possibly branching on different outputs. For each output y that s can produce from its initial state we let t^y, t^y_U, t^y_L, and s^y be defined by $t \xrightarrow{y} t^y$ and $t_U||s||t_L \xrightarrow{y}_t t^y_U||s^y||t^y_L$. Thus, $s^y = \sum S$, being S the set of processes that s can become after y. Let $y_p = y|_p$ for $p \in \{U, L\}$ and so $t_p \xrightarrow{y_p} t^y_p$. Clearly t^y and (t^y_U, t^y_L) are deterministic for s^y. We can now apply the inductive hypothesis to t^y, (t^y_U, t^y_L), and s^y and the result follows.

Case 3: t starts with δ and so t_U and t_L both start with δ. Then $s \overset{\delta}{\Longrightarrow} s$ and so the result follows from the inductive hypothesis.

6 Conclusions

This paper has investigated testing from an input output transition system in the distributed test architecture. The problem of testing from a deterministic finite state machine in this architecture has received much attention but, while deterministic finite state machines are appropriate for modelling several important classes of system, input output transition systems are more general.

When testing in the distributed test architecture each tester only observes the actions at its port. As a result, it is not possible to reconstruct the global trace after testing and this has an effect on the ability of testing to distinguish between two processes. We introduced a new implementation relation **dioco** that captures the ability of testing to compare an implementation and a specification: i **dioco** s if and only if it is possible for testing to show that i does not correctly implement s when testing in the distributed test architecture.

It is normal to require a test case to be deterministic. However, there are test cases that are deterministic if there is a single global tester that interacts with the SUT at all its ports but that cannot be implemented as a deterministic local test case when testing in the distributed test architecture. We have defined a function that takes a global test case and returns a local test case that consists of a tester for each port. We have proved that this function returns a local test case if and only if it is possible to implement the global test case using a deterministic local test case.

There are several avenues of future work. First, we could parameterize the implementation relation with a set of input sequences or traces. In addition, there is the problem of automatically generating test cases that can be implemented as deterministic local test cases. It has been shown that controllability and observability problems, when testing from a deterministic finite state machine, can be overcome if the testers can exchange coordination messages through an external network and there should be scope for using coordination messages when testing from an input output transition system. Finally, recent work has described systems in which an operation is triggered by receiving input at more than one port [20] and it would be interesting to extend the work to such systems.

References

1. ISO/IEC JTC 1, J.T.C.: International Standard ISO/IEC 9646-1. Information Technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 1: General concepts. ISO/IEC (1994)
2. Sarikaya, B., Bochmann, G.v.: Synchronization and specification issues in protocol testing. IEEE Transactions on Communications 32, 389–395 (1984)
3. Luo, G., Dssouli, R., Bochmann, G.v.: Generating synchronizable test sequences based on finite state machine with distributed ports. In: 6th IFIP Workshop on Protocol Test Systems, IWPTS 1993, pp. 139–153. North-Holland, Amsterdam (1993)
4. Tai, K.C., Young, Y.C.: Synchronizable test sequences of finite state machines. Computer Networks and ISDN Systems 30(12), 1111–1134 (1998)

5. Rafiq, O., Cacciari, L.: Coordination algorithm for distributed testing. The Journal of Supercomputing 24(2), 203–211 (2003)
6. Ural, H., Williams, C.: Constructing checking sequences for distributed testing. Formal Aspects of Computing 18(1), 84–101 (2006)
7. Khoumsi, A.: A temporal approach for testing distributed systems. IEEE Transactions on Software Engineering 28(11), 1085–1103 (2002)
8. Hierons, R.M., Ural, H.: The effect of the distributed test architecture on the power of testing. The Computer Journal (to appear, 2008)
9. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software – Concepts and Tools 17(3), 103–120 (1996)
10. Tretmans, J.: Testing concurrent systems: A formal approach. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 46–65. Springer, Heidelberg (1999)
11. Núñez, M., Rodríguez, I.: Towards testing stochastic timed systems. In: König, H., Heiner, M., Wolisz, A. (eds.) FORTE 2003. LNCS, vol. 2767, pp. 335–350. Springer, Heidelberg (2003)
12. Brandán Briones, L., Brinksma, E.: A test generation framework for quiescent real-time systems. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 64–78. Springer, Heidelberg (2005)
13. Krichen, M., Tripakis, S.: Black-box conformance testing for real-time systems. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 109–126. Springer, Heidelberg (2004)
14. Bijl, M.v., Rensink, A., Tretmans, J.: Action refinement in conformance testing. In: Khendek, F., Dssouli, R. (eds.) TestCom 2005. LNCS, vol. 3502, pp. 81–96. Springer, Heidelberg (2005)
15. López, N., Núñez, M., Rodríguez, I.: Specification, testing and implementation relations for symbolic-probabilistic systems. Theoretical Computer Science 353(1-3), 228–248 (2006)
16. Frantzen, L., Tretmans, J., Willemse, T.: A symbolic framework for model-based testing. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) FATES 2006 and RV 2006. LNCS, vol. 4262, pp. 40–54. Springer, Heidelberg (2006)
17. Merayo, M., Núñez, M., Rodríguez, I.: Formal testing from timed finite state machines. Computer Networks (to appear, 2008)
18. Brinksma, E., Heerink, L., Tretmans, J.: Factorized test generation for multi-input/output transition systems. In: IFIP TC6 11th International Workshop on Testing Communicating Systems (IWTCS), pp. 67–82. Kluwer, Dordrecht (1998)
19. Alur, R., Etessami, K., Yannakakis, M.: Inference of message sequence charts. IEEE Transactions on Software Engineering 29(7), 623–633 (2003)
20. Haar, S., Jard, C., Jourdan, G.V.: Testing input/output partial order automata. In: Petrenko, A., Veanes, M., Tretmans, J., Grieskamp, W. (eds.) TestCom/FATES 2007. LNCS, vol. 4581, pp. 171–185. Springer, Heidelberg (2007)

Modular System Verification by Inference, Testing and Reachability Analysis

Roland Groz[1], Keqin Li[2], Alexandre Petrenko[3], and Muzammil Shahbaz[4]

[1] LIG, Grenoble Computer Science Lab, France
Roland.Groz@imag.fr
[2] SAP Research, France
Keqin.Li@sap.com
[3] CRIM, Canada
Alexandre.Petrenko@crim.ca
[4] France Télécom R&D/Orange Labs, France
Muhammad.MuzammilShahbaz@orange-ftgroup.com

Abstract. Verification of a modular system composed of communicating components is a difficult problem, especially when the models of the components are not available. Conventional testing techniques are not efficient in detecting erroneous interactions of components because such interactions often occur as interleavings of events that are difficult to reproduce in a modular system. The problem of detecting intermittent errors in the absence of models of components is addressed in this paper. A method to infer a controllable approximation of components through testing is elaborated. The inferred finite state models of components are used to detect intermittent errors and other compositional problems in the system through reachability analysis. The models are refined at each analysis step thus making the approach iterative.

1 Introduction

Integration of components is now a major mode of software development. Very often, components coming from outside sources (such as COTS) have to be connected to build a system. In most cases, the components do not come with a formal model, just with executable or in some cases source code. At the same time, the interaction of components may lead to integration bugs that may be hard to find and trace, especially in the absence of any model or other development information. In this paper, we are targeting compositional problems in the behaviours of a system composed of communicating components. Specifically, we aim at identifying intermittent (sporadic) errors occurring in event interleavings that are difficult to reproduce in an integrated system. Generally speaking, the system may eventually produce several event interleavings in response to a given external input sequence, if that sequence is applied several times. This occurs in particular when the execution order of the components changes over time from one experiment to another, typically because of different scheduling, varying load and communication jitters. However, it is unrealistic to expect to be able to enforce all the interleavings during testing. Therefore, the intermittent errors are hard to elicit and to reproduce in functional testing.

On the other hand, all potential interleavings can be checked for potential errors by instrumenting the modular system and executing it in a controlled environment to observe all the possible executions and interactions of all its components, see, e.g., Verisoft [6]. For components without source code, this approach may not be applicable and a model-based approach can be attempted. Indeed, if component models are available, the global model state space can be exhaustively searched (reachability analysis) to look for compositional problems, using, for instance, a model checker. As stated earlier, the models usually do not exist. One possibility would be to reverse engineer them from the code, but reconstruction based on static analysis has a number of limitations. The main alternative is to infer them from executions. However, given the complexity of typical software components, it is unrealistic to assume that components could be modelled with a perfect abstraction in a finite, compact representation. Inferring approximated models of components in a given modular system appears to be more realistic.

In this paper, we are developing an approach to verify a modular system by inferring tunable approximated models of its components through testing and performing reachability analysis to detect intermittent errors and other compositional problems in the system. This approach possesses two main advantages regarding the models that are inferred. First, it allows derivation of models of the components describing the behaviours that can actually be exhibited in the integrated system. Typically, components bundle a number of functions, but it is often the case that only a subset of those functions are used in the system, so inferring them in isolation is hard if not impossible; whereas our approach delivers models which omit behaviours unused in the integrated system. Second, the models are determined with a controllable precision to balance between the level of abstraction and the amount of efforts needed to obtain the models.

We make the following assumptions about a given system:

- The system interacts with a slow environment which submits external inputs only when the system stabilizes.
- Components are black boxes that behave as finite state (Mealy) machines and interact asynchronously. In response to an input, a component can produce several outputs to other components or the environment of the system.
- Each component is deterministic; however, due to possible jitter in communication delays, or scheduling of components, the system might not be.
- The external input actions of the system are known.
- Every output action of the components can be observed in testing.

No additional information about the system, such as the number of states, a priori given positive or negative samples of its behavior or teacher [9], often used in traditional model learning, is available for model inference. The components are modelled using Finite State Machine (FSM) with multiple outputs, or equivalently Input Output Transition System (IOTS) (with restrictions). A modular system is composed of IOTS components that communicate asynchronously through queues modelled by IOTS. We define an approximation of an FSM, called k-quotient, for any given positive integer k, based on state distinguishability. The precision of this approximation can be controlled by the parameter k, which is important, since identification of a state machine is in general infeasible without knowing the number of its states.

The first contribution of this paper is an algorithm that computes a k-quotient for a component (thus, for a whole system) by testing in the two following steps: behavior exploration bounded by the parameter k and "folding" of the observed behavior by state merging using trace inclusion relation. We then elaborate an approach to infer a k-quotient of a modular system. The k-quotient of the modular system with observable internal actions in the form of an IOTS is used to infer initial models of components by projecting the quotient. Reachability analysis of the models is next performed to identify witness (diagnostic) traces of a composition problem, such as unspecified receptions, livelocks, and races. The identified witness needs to be tested in the real system, to check whether it is an artifact coming from our approximations or the system indeed has this problem. However, since we cannot control the delays occurring in the integrated system, each component is tested in isolation on a projection of the witness trace. A witness refuted by testing the components yields new observations. The models are then refined using new observations and the process iterates until the obtained models are well-formed. The obtained component models are consistent with all observations made on the real system. Once composed, they are at least as accurate as the k-quotient model of the system. At the same time, the obtained models are well-formed, i.e., have no compositional problems that may otherwise occur with sequences of at least k external inputs to the system.

The paper is organized as follows. Section 2 defines a k-quotient of an FSM and presents an algorithm for its inference. Section 3 restates the results of Section 2 for Input Output Transition Systems. Inference of modular systems is elaborated in Section 4. The notion of a slow asynchronous product is defined to formalize the interactions of components in a slow environment and several known compositional problems along with the witness traces are formally defined. The approach is illustrated on a small example. Section 5 discusses the related work. Section 6 concludes the paper.

2 Inferring Finite State Machine

2.1 Basic Definitions

In this section, some basic definitions about finite state machines with multiple outputs are given.

A *Finite State Machine with multiple outputs* (FSM) A is a 6-tuple (S, s_0, I, O, E, h), where

- S is a finite set of states with the initial state s_0;
- I and O are finite non-empty disjoint sets of inputs and outputs, respectively;
- E is a finite set of finite sequences of outputs in O (may include the empty sequence ε);
- h is a behavior function h: $S \times I \rightarrow 2^{S\times E}$, where $2^{S\times E}$ is the powerset of $S \times E$.

FSM $A = (S, s_0, I, O, E, h)$ is

- *completely specified* (a complete FSM) if $h(s, a) \neq \emptyset$ 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 $A_\times = (S, s_0, I \times E, \delta)$, where $\delta(s, a\beta) \ni s'$ iff $(s', \beta) \in h(s, a)$, is deterministic.

In this paper, we consider only observable machines; one could use a standard procedure for automata determinization to transform a given FSM into an observable one. We use a, b, c for input symbols, α, β, γ for input (or output) sequences, s, t, p, q for states, and u, v, w for traces.

In FSM $A = (S, s_0, I, O, E, h)$, $(s, a\beta, t)$ is a *transition* if $s, t \in S$ and $(t, \beta) \in h(s, a)$. A *path* from state s_1 to s_{n+1} is a sequence of transitions $(s_1, a_1\beta_1, s_2)(s_2, a_2\beta_2, s_3)\ldots(s_n, a_n\beta_n, s_{n+1})$ s.t. $(s_{i+1}, \beta_i) \in h(s_i, a_i)$ $(1 \leq i \leq n)$. The length of the path is n. A sequence $u \in (I \times E)^*$ is called a *trace* of FSM A in state $s_1 \in S$, if there exists a path $(s_1, a_1\beta_1, s_2)(s_2, a_2\beta_2, s_3)\ldots(s_n, a_n\beta_n, s_{n+1})$ s.t. $u = a_1\beta_1a_2\beta_2\ldots. a_n\beta_n$. Note that a trace of A in state s_0 is a word of the automaton $A_\times$.

Let $Tr(s)$ denote the set of all traces of A in state s, while $Tr(A)$ denotes the set of traces of A in the initial state. Given sequence $u \in (I \times E)^*$, the *input projection* of u, denoted $u_{\downarrow I}$, is a sequence obtained from u by erasing symbols in O. Input sequence $\beta \in I^*$ is a *defined* input sequence in state s of A if there exists $u \in Tr(s)$ s.t. $\beta = u_{\downarrow I}$. We use $\Omega(s)$ to denote the set of all defined input sequences for state s.

Given FSM $A = (S, s_0, I, O, E, h)$ and $s, t \in S$, s and t are *equivalent*, $s \cong t$, if $Tr(s) = Tr(t)$. States s and t are *distinguishable*, $s \ncong t$, if there exists $\alpha \in \Omega(s) \cap \Omega(t)$ s.t. $\{u \in Tr(s) \mid u_{\downarrow I} = \alpha\} \neq \{u \in Tr(t) \mid u_{\downarrow I} = \alpha\}$, called an input sequence *distinguishing* s and t.

We use $Tr^k(s)$ to denote the set of traces in $s \in S$ each of which has at most k inputs, i.e., $Tr^k(s) = \{u \mid u \in Tr(s) \wedge |u_{\downarrow I}| \leq k\}$. Two states $s, t \in S$ are *k-equivalent* iff $Tr^k(s) = Tr^k(t)$, otherwise, if $Tr^k(s) \neq Tr^k(t)$, states s and t are *k-distinguishable*, in this case there exists a distinguishing sequence of the length at most k.

Given two complete FSM $L = (S, s_0, I, O, E_L, h_L)$ and $K = (P, p_0, I, O, E_K, h_K)$, L and K are (k-)equivalent, iff s_0 and p_0 are (k-)equivalent.

2.2 Initial k-Quotient

Definition 1. Given two complete FSM $L = (S, s_0, I, O, E_L, h_L)$ and $K = (P, p_0, I, O, E_K, h_K)$, K is an *initial k-quotient* of L (or simply k-quotient) if

1. $P \subset 2^S$ s.t. $s_0 \in p_0$ and if $s \in p_1$ and $t \in p_2$ for $p_1, p_2 \in P$ then
   - $p_1 = p_2$, if s and t are k-equivalent or
   - $p_1 \neq p_2$, if s and t are k-distinguishable.
2. For all $p \in P$ there exists $s \in p$, s.t. for all $a \in I$
   - $(s', \beta) \in h_L(s, a)$ implies that there exists $p' \in P$, s.t. $(p', \beta) \in h_K(p, a)$, and $s' \in p'$;
   - $(p', \beta) \in h_K(p, a)$ implies that there exists $s' \in S$, s.t. $(s', \beta) \in h_L(s, a)$, and $s' \in p'$.

Initial k-quotients possess the following properties.

Theorem 1. Given FSM K, an initial k-quotient of a complete FSM L, if all the distinguishable states of L are k-distinguishable, then FSM L and K are equivalent; otherwise, if some distinguishable, but k-equivalent, states of L are reachable from the initial state, then L and K are k-equivalent, but are distinguishable.

A k-quotient of an FSM is, thus, its approximation, whose precision can be varied with the parameter k. We omit the proofs of the theorems in the paper due to space limit.

2.3 Inferring k-Quotient of FSM

We want to infer a k-quotient of an FSM A by testing. We assume that A is completely specified and deterministic; moreover, only its input set I is known, while the output set (or at least part of it) will be determined from A's outputs.

The basic idea of our inference method is to observe the traces of the unknown FSM A from all k-distinguishable states that can be reached from the initial state, by applying in each state all the input sequences of length k. To represent the observed traces of FSM A, we use a tree FSM.

Definition 2. Given a (prefix closed[1]) set U of observed traces of A over input set I and output set O, the *observation tree* FSM is $(U, \varepsilon, I, O, E_U, h_U)$, where the state set is U, $E_U = \{\beta \in O^* \mid \exists a \in I, \exists u \in U \text{ s.t. } ua\beta \in U\}$, and $h_U(u, a) = \{(ua\beta, \beta) \mid \exists \beta \in O^* \text{ s.t. } ua\beta \in U\}$.

We use U to refer to both, a prefix-closed set of FSM traces and the corresponding tree FSM $(U, \varepsilon, I, O, E_U, h_U)$. In a (tree) FSM U, a state u is a *k-predecessor* of state w, iff u is a proper prefix of w and $|w_{\downarrow I}| - |u_{\downarrow I}| \leq k$.

The inference method includes two steps, behavior exploration resulting in an observation tree and state merging in the tree which yields an FSM. The exploration step terminates when no new state, i.e., k-distinguishable from all the other visited states, can be reached from the initial state. The output of the exploration procedure is an observation tree U. In the next step, we obtain a model M of the FSM A by merging states of U using trace inclusion relation between states.

In the exploration step, we perform a Breadth First Search (BFS) on U, starting with $U = \{\varepsilon\}$, to find a state $u \in U$, which is unmarked and has no k-predecessor marked "prune" (marking is done as explained below), and explore the behavior of the given machine from the state u by applying all the possible input sequences of length k, observing the outputs, and adding the observed traces into U. If the state u is k-equivalent to an already visited state, we mark it "prune".

In the state merging step, we traverse the obtained observation tree U following a BFS and if the trace set of an already visited state is a superset of traces of the current state we merge the two states.

[1] Recall that a symbol of an FSM trace is a pair of an input from I and a sequence of outputs from O^*, so every prefix takes an FSM from its initial state into some state.

The following theorem claims that the above method can be used to infer an initial k-quotient of an FSM being tested.

Theorem 2. If the inference method is applied to a deterministic FSM A and yields an FSM M, then FSM M is equivalent to an initial k-quotient of FSM A.

3 Inferring Input/Output Transition System

3.1 Basic Definitions

Certain operations on FSMs, such as composition, are easier to formulate using their transition system counterparts. An *input/output transition system* (IOTS) L is a quintuple $<S, I, O, \lambda, s_0>$, where S is a set of states; I and O are disjoint sets of input and output actions, respectively; $\lambda \subseteq S \times (I \cup O \cup \{\tau\}) \times S$ is the transition relation, with the symbol τ denoting internal actions; and s_0 is the initial state.

$(t, a, s) \in \lambda$ is called a *transition*; (t, a, s) is *input*, *output* or *internal* transition, if $a \in I$, $a \in O$ or $a = \tau$, respectively. Given IOTS L, a *path* from state s_1 to state s_{n+1} is a sequence of transitions $p = (s_1, a_1, s_2)(s_2, a_2, s_3)\ldots(s_n, a_n, s_{n+1})$, s.t. $(s_i, a_i, s_{i+1}) \in \lambda$ for $i = 1, \ldots, n$.

Let ε denote the empty sequence of actions. The *projection operator* $\downarrow A$, which projects sequences of actions onto the set $A \subseteq I \cup O \cup \{\tau\}$, is recursively defined as $\varepsilon_{\downarrow A} = \varepsilon$, $(va)_{\downarrow A} = v_{\downarrow A}a$ if $a \in A$, and $(va)_{\downarrow A} = v_{\downarrow A}$ otherwise, where $v \in (I \cup O \cup \{\tau\})^*$ and $a \in I \cup O \cup \{\tau\}$. We also lift the projection operator to IOTS, i.e., given IOTS $L = <S, I, O, \lambda, s_0>$ and set $A \subseteq I \cup O$, the IOTS $L_{\downarrow A}$ is obtained by first replacing each transition $(t, a, s) \in \lambda$ s.t. $a \notin A$ by internal transition (t, τ, s) and then determinizing the obtained IOTS (with tau-reduction) [18].

A sequence $u \in (I \cup O)^*$ is called a *trace* of IOTS L in state $s_1 \in S$ if there exists a path $(s_1, a_1, s_2)(s_2, a_2, s_3)\ldots(s_n, a_n, s_{n+1})$, s.t. $u = (a_1\ldots a_n)_{\downarrow(I \cup O)}$. Similar to FSM, we use $Tr(T)$ to denote the set of traces in states $T \subseteq S$, while $Tr(L)$ to denote the set of traces of L in the initial state. We use $Tr^k(s)$ to denote the set of traces in $s \in S$, each of which has at most k input actions, i.e., $Tr^k(s) = \{u \mid u \in Tr(s) \wedge |u_{\downarrow I}| \leq k\}$. *k-equivalent* and *k-distinguishable* states are defined similar to that of FSM.

We use $init(s)$ to denote the set of actions enabled in state s, i.e., $init(s) = \{a \in (I \cup O \cup \{\tau\}) \mid \exists t \in T \text{ s.t. } (s, a, t) \in \lambda\}$. L is *input-enabled* if all input actions are enabled in each state, i.e., $I \subseteq init(s)$ for each $s \in S$; L is *fully specified* if either all or no input actions are enabled in each state, i.e., either $I \subseteq init(s)$ or $I \cap init(s) = \emptyset$ for each $s \in S$. If L is not fully specified, it is *partially specified*. In state s of a fully specified IOTS L, either L does not read input at all if no input is enabled in s, or L's behavior after any input is defined in s. The response of L to any input, therefore, is predictable, and hence we call L "fully specified". L is *deterministic* if it has no internal transitions and λ is a function $S \times (I \cup O) \rightarrow S$. State $s \in S$ is *stable* if no output or internal actions are enabled in s, i.e., $init(s) \cap (O \cup \{\tau\}) = \emptyset$, otherwise it is *unstable*. IOTS L is *conflict-free* if input actions are only enabled in stable states. Intuitively, such a system is allowed to produce all possible outputs in response to input before the environment offers a next input, similar to FSM, where input/output transitions are atomic. In fact, any FSM, once each of its transitions is unfolded into input transition followed by output transition (which is omitted, if the output is empty, $\varepsilon \in E$), to an intermediate state, yields a conflict-free IOTS.

In a conflict-free IOTS, there is no nondeterminism related to concurrency of inputs and outputs, but there may still be nondeterministic choices of outputs. A conflict-free IOTS L is output deterministic if it produces a single output sequence in response to any input sequence. Formally, $L = <S, I, O, \lambda, s_0>$ is *output-deterministic* iff $\alpha_{\downarrow I} = \beta_{\downarrow I}$ implies $\alpha = \beta$ for any $\alpha, \beta \in Tr(s_0)$, otherwise it is *output-nondeterministic*. Thus, an IOTS can be output-nondeterministic and, at the same time, deterministic.

State $s \in S$ is a *deadlock* if no action is enabled in it, i.e., $init(s) = \emptyset$. State $s \in S$ is a *livelock* if there is a cycling path of output or internal transitions that includes s; if the path includes only internal transitions then livelock is *internal*, otherwise it is an *output* livelock. IOTS L is *deadlock-free* or *livelock-free*, if there is no deadlock or livelock state reachable from a starting state, respectively. L is *input-progressive* if it is deadlock-free and livelock-free. If L is input-progressive, input actions are defined in each stable state, and it enters a stable state executing fewer than $|S|$ transitions after any input.

3.2 Inferring *k*-Quotient of IOTS

As mentioned earlier, livelock-free, deterministic, and conflict-free finite IOTS can be considered as another representation of observable FSM. This indicates that the inference method for FSM can be used for this kind of IOTS. We assume that an unknown IOTS A is finite, fully specified, deterministic, output-deterministic, and conflict-free, moreover, its input action set I is known. Notice that by requiring IOTS be output-deterministic we make the results of Section 2 directly applicable to IOTS. Since a deadlock state cannot observationally be distinguished from a stable state where all inputs cause looping transitions, we also assume that the IOTS A has no deadlock.

The observation tree is defined as follows: given a (prefix closed) set U of observed traces of A over input action set I and output action set O, the observation tree is an IOTS $<U, I, O, \lambda_T, \varepsilon>$, where the state set is U, and $(\beta, a, \beta a) \in \lambda_T$ iff $\beta a \in U$. A state β of U is stable if $\beta a \notin U$ for all $a \in O$. We use U to refer to both, a prefix-closed set of traces and the IOTS $<U, I, O, \lambda_T, \varepsilon>$. Based on the observation tree, we can determine k-equivalence of stable states.

In a (tree) IOTS U, a stable state u is a *k-predecessor* of stable state w, iff u is a proper prefix of w and $|w_{\downarrow I}| - |u_{\downarrow I}| \leq k$.

The inference method for IOTS also includes two steps, behavior exploration and state merging.

In the exploration step, we perform a BFS on stable states of U, starting with $U = \{\varepsilon\}$, to find a stable state $u \in U$, which is unmarked and has no k-predecessor marked "prune", and explore the behavior of the given IOTS from the state u by applying all the possible input sequences of length k, observing the outputs, and adding the observed traces into U. If the state u is k-equivalent to an already visited stable state, we mark it "prune". Applying an input action to IOTS A, we wait for output actions. By our assumption, the IOTS A has no deadlock; at the same time, it may still have output livelocks. To detect output livelock in a black box, we set a bound for the maximum length of output sequences the IOTS can produce in response to a single input action. This bound cannot exceed the number of states in the IOTS. If the length of observed output sequence exceeds the given bound, exploration terminates and output livelock is declared.

In the state merging step, we traverse the stable states of the obtained observation tree U following a BFS and if the trace set of an already visited stable state is a superset of traces of the current stable state we merge the two states.

Since livelock-free, deterministic and conflict-free IOTS is, in fact, another representation of observable FSM, Theorem 2 still applies to the above procedure once initial k-quotient of FSM is replaced by its IOTS counterpart, which definition we omit here for simplicity.

4 Inference of Modular Systems

4.1 Basic Definitions

In this paper, we use queued communications between system's components. Modeling queues, we distinguish the same action at the two ends of a queue by using the relabeling $'$ operator [1]. The operator is defined on input actions: for $a \in I$, $(a)' = a'$, and $(a')' = a$. It is lifted to the sets of input actions, traces, and IOTS: for an action set I, $I' = \{a' \mid a \in I\}$; for traces, $'$ is recursively defined as $\varepsilon' = \varepsilon$ and $(ua)' = u'a'$ for trace u if a is an input action, otherwise $(ua)' = u'a$; L' is obtained from L by relabeling each action $a \in I$ to a'. A (unbounded) *queue with input set I*, is an IOTS $<I^*, I, I', \lambda_I, \varepsilon>$, denoted Q_I, where the state set I^* and transition relation $\lambda_I = \{(u, a, ua) \mid u, ua \in I^*\} \cup \{(av, a', v) \mid av, v \in I^*\}$. The only stable state of a queue is its initial state.

We consider a modular system consisting of components communicating asynchronously, where each component reads inputs from its input queue and writes outputs to other components' input queues. We assume in this paper that the components are conflict-free, as well as input-progressive; moreover, without losing generality, we also assume that they are deterministic. Each component is not input-enabled, but the composition of the component with its input queue is input-enabled. It means that even if the component does not read an input in some state, the input is not lost, kept in the input queue, and can be consumed in subsequent state reached by internal or output transitions, in other words, we do not need to assume that inputs can be blocked or refused, as in other work [2].

Let $C = \{C_1, C_2, \ldots, C_n\}$ be a set of component IOTS's, where $C_i = <S_i, I_i, O_i, \lambda_i, s_{0i}>$, s_{0i} is a stable initial state, s.t. $I_i \cap I_j = \emptyset$ and $O_i \cap O_j = \emptyset$ for $i \neq j$. Let I and O denote the union of all input action sets and output action sets, respectively. Two components C_1 and C_2 *communicate* if $I_1 \cap O_2 \neq \emptyset$ or $O_1 \cap I_2 \neq \emptyset$. For each component C_i with the input set I_i, there is an unbounded input queue $Q_{I_i} = <I_i^*, I_i, I_i', \lambda_{I_i}, \varepsilon_i>$, thus, each component consumes inputs from its input queue and produces an external output or internal output, the later is stored in input queue of other component. The set $I_{\text{ext}} = I\backslash O$ contains *external inputs*; components constitute a *closed* system, if I_{ext} is empty, otherwise an *open* system. Let $O_{\text{ext}} = O \backslash I$ be the set of *external outputs* of the system. In this paper, we consider only an open system $C = \{C_1, C_2, \ldots, C_n\}$ with at least one external output, s.t. each component communicates in the system, more precisely, we assume that for each component C_i it holds that if $a \in I_i$ then either $a \in I_{\text{ext}}$ or $a \in O_j$ and if $a \in O_i$ then $a \in O_{\text{ext}}$ or $a \in I_j$ for some C_j.

A behavior of an open system may vary, depending on the speed of its environment. We distinguish two types of the environment, fast and slow. A *fast* environment can supply external inputs at any state of a system with which it communicates. A *slow* environment does so only when the system is in a stable global state.

The behavior of the system operating in the fast environment is described by the IOTS $C_1' \parallel C_2' \parallel \ldots C_n' \parallel Q_{I_1} \parallel Q_{I_2} \parallel \ldots \parallel Q_{I_n}$, where $\parallel$ is the standard LTS parallel composition operator, $C_i' = \langle S_i, I_i', O_i, \lambda_i, s_{0i} \rangle$, for $C_i = \langle S_i, I_i, O_i, \lambda_i, s_{0i} \rangle$, and $Q_{I_i} = \langle I_i^*, I_i, I_i', \lambda_{I_i}, \varepsilon_i \rangle$. To describe the behavior in case of the slow environment, we modify the $\parallel$ operator to allow external inputs only in stable global states.

Definition 3. Given a system of communicating components $C = \{C_1, C_2, \ldots, C_n\}$, where $C_i = \langle S_i, I_i, O_i, \lambda_i, s_{0i} \rangle$ and the set of queues over input alphabets of the components $Q = \{Q_{I_1}, \ldots, Q_{I_n}\}$, where $Q_{I_i} = \langle I_i^*, I_i, I_i', \lambda_{I_i}, \varepsilon_i \rangle$, the *slow asynchronous product* of C, denoted $\Sigma = \Pi_{i=1}^{n} C_i$, is the IOTS $\langle R, I_{\text{ext}}, O, \lambda, s_{01} \ldots s_{0n} \varepsilon_1 \ldots \varepsilon_n \rangle$, where $I_{\text{ext}} = I \backslash O$, $I = I_1 \cup \ldots \cup I_n$ and $O = I_1' \cup \ldots \cup I_n' \cup O_1 \cup \ldots \cup O_n$; the set of states $R \subseteq S_1 \times \ldots \times S_n \times I_1^* \times \ldots \times I_n^*$ and the transition relation λ are the smallest sets obtained by applying the following inference rules:

- $s_{01} \ldots s_{0n} \varepsilon_1 \ldots \varepsilon_n \in R$;
- if $a \in I_{\text{ext}} \cap I_i$, $(s_1 \ldots s_n \varepsilon_1 \ldots \varepsilon_n) \in R$ s.t. states $s_1, \ldots, s_n$ are stable, then $(s_1 \ldots s_n \varepsilon_1 \ldots \varepsilon_n, a, s_1 \ldots s_n b_1 \ldots b_n) \in \lambda$ and $(s_1 \ldots s_n b_1 \ldots b_n) \in R$ s.t. $b_i = a$, and $b_j = \varepsilon_j$ for $j \neq i$ (external input is buffered in the queue of the component, which has it as input);
- if $a \in I_i$, $(s_1 \ldots s_n b_1 \ldots b_n) \in R$ s.t. $a \in init(s_i)$, $b_i = av$, then $(s_1 \ldots s_n b_1 \ldots b_n, a', s_1' \ldots s_n' c_1 \ldots c_n) \in \lambda$ and $(s_1' \ldots s_n' c_1 \ldots c_n) \in R$, s.t. $(s_i, a, s_i') \in \lambda_i$, and $c_i = v$; $s_j' = s_j$ and $c_j = b_j$ for $j \neq i$ (input is consumed from a queue, and input transition is executed);
- if $a \in O_i$, $(s_1 \ldots s_n b_1 \ldots b_n) \in R$ s.t. $a \in init(s_i)$, then $(s_1 \ldots s_n b_1 \ldots b_n, a, s_1' \ldots s_n' c_1 \ldots c_n) \in \lambda$ and $(s_1' \ldots s_n' c_1 \ldots c_n) \in R$, s.t. $(s_i, a, s_i') \in \lambda_i$, $s_j' = s_j$ for all $j \neq i$, and if $a \in I_j$ then $c_j = b_j a$, otherwise, $c_j = b_j$ (output transition is executed, and output is buffered in the queue of the component, for which it is an input).

Notice that all components' inputs, save external ones, as well as outputs, become (observable) outputs of the composition, as is usually the case for input-output automata composition [3]. The notion of slow asynchronous product defined here is similar to what has already been implemented for SDL tools such as Object Geode.

Definition 4. A system C of communicating components in slow environment has

- *unspecified reception*, if in the product Σ, there exists a state $(s_1 \ldots s_n b_1 \ldots b_n)$ s.t. for some component C_i, a is a prefix of b_i and $a \in I_i$, but $a \notin init(s_i)$;
- *compositional livelock*, if Σ has a livelock state;

- *divergence*, if in Σ, there exists a path from state $(s_1...s_nb_1...b_n)$ to state $(s_1...s_nc_1...c_n)$ s.t. each b_i is a prefix of c_i, and there exists $d \neq \varepsilon$, s.t. b_idd is a prefix of c_i.
- *races*, if there exist traces $\alpha, \beta \in Tr(\Sigma)$ s.t. $\alpha\downarrow_{I_{\text{ext}}} = \beta\downarrow_{I_{\text{ext}}}$ and $\alpha\downarrow_{O_{\text{ext}}} \neq \beta\downarrow_{O_{\text{ext}}}$.

The system with at least one of the above properties is said to have a *compositional* problem, the system with no compositional problem is *well-formed*. Notice that unspecified receptions cause compositional deadlocks, divergence causes buffer overflow, and races are a witness of a nondeterministic behavior of a system composed of deterministic components.

Given a system of communicating components, the slow asynchronous product can be constructed and, thus, its well-formedness can be checked using a classical reachability analysis (RA) procedure which we will not discuss further in this paper. We simply assume that given a system $C = \{C_1, C_2, ..., C_n\}$, the procedure either confirms that the system is well-formed or outputs the following *witness* (diagnostic) traces for:

- unspecified reception of $a \in I_i$, a trace βa, s.t. β takes the product into a state, where the action a is not enabled in the corresponding state of some component C_i whose input queue contains just a;
- compositional livelock and divergence, a trace $\alpha\beta$, s.t. α takes the product into a state of a cycle or path, respectively, labeled by the sequence β;
- races, two traces α and β with a common external input projection which leads to races.

4.2 The Approach

To infer models of communicating components, one can determine a k-quotient of each component in isolation using the method of Section 3. This approach requires that an appropriate value of the parameter k be chosen individually for each component. Intuitively, a component with more states would require a bigger value than components with fewer states, however, it is unclear how one can determine these parameters without taking into account the components' interactions, except for (rare) cases when the maximal number of states is a priori known for each component. A trial-and-error approach can be followed, and even if it succeeds, it may result in "redundant" models of components which describe functionalities unused in a given system.

Another approach which we follow in this paper is to infer first a k-quotient of the slow asynchronous product (thus, assuming that internal outputs are observable for testing) and determine models of components by projecting the product onto the alphabets of each component and refining them if needed. If the given system has "a single message in transit", (see, e.g., [16]) then the slow asynchronous product is output-deterministic and the inference method of Section 3 directly applies. If, however, several actions can concurrently be executed in the system, the product becomes output-nondeterministic. This means that the system could produce several output interleavings in response to a given external input sequence, if that sequence is applied several times. The problems coming from such causes are often hard to elicit and to reproduce in functional testing; they often appear as a side effect in stress testing, but are harder to analyze in that stage. The crux of our approach is precisely to be able to identify such intermittent problems that occur only under specific circumstances in integrated systems.

Therefore, we assume that during the exploration the system behaves as an output-deterministic IOTS and use the inference method of Section 3. We do not rely on "all weather", aka "complete testing", assumption [4]. First, such an assumption is very costly because each input sequence must be tested a potentially huge number of times (esp. for long sequences); second, it is often unrealistic to assume that all interleavings will be observed in a given test configuration, and if configurations must be changed for each occurrence of an input sequence, this is even more costly in test execution time.

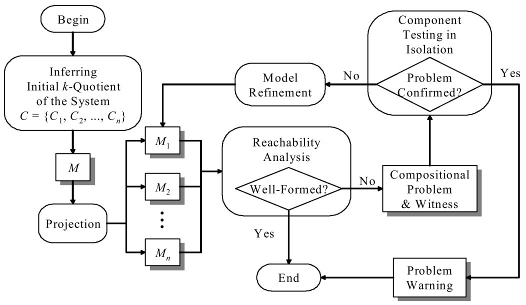

Begin
Inferring Initial k-Quotient of the System $C = \{C_1, C_2, ..., C_n\}$
M
Projection
M_1
M_2
M_n
Model Refinement
No
Component Testing in Isolation
Problem Confirmed?
Yes
Reachability Analysis
Well-Formed?
No
Compositional Problem & Witness
Yes
End
Problem Warning

Fig. 1. The verification approach

To identify and check all possible interleavings, we use the RA procedure. Even if the components themselves constitute a well-formed system the inferred models do not necessarily do so. Each obtained model is only an approximation of the actual behavior of a real component and may possess a behavior absent in the component, so the inferred models may exhibit compositional problems. RA can discover them by exploring all the possible execution interleavings.

The models exhibit a compositional problem in two cases: either the problem exists in the real system or the inferred models are not adequate models of the behavior of the system exposed during the exploration step and need to be further refined. To confirm that a compositional problem detected in RA exists in the real system, one can check whether each projection of a witness trace, which was not observed in the exploration step, belongs to the set of traces of the corresponding component by executing the projected trace against the component in isolation. The actions of the witness trace determine the order of testing the components involved in the execution, which terminates as soon as some component produces a trace different from the expected one. In this case, the compositional problem is refuted, and the newly obtained trace is used to refine the model of that component. The iterative process terminates when either a compositional problem is confirmed to exist in the real system or a well-formed system of models is obtained. Notice the real system to have compositional livelock or divergence should exhibit a cyclic behavior in all the involved components when they are tested using the projected witness traces. In black-box testing, we can only confirm this by repeatedly applying input sequence of a trace to each component. The verification approach is summarized in Figure 1.

While the main steps of this procedure are intuitive and clear from the previous discussions, the model refinement needs some explanation.

Let U be a global observation tree obtained by exploring the behavior of a given system of communicating components $C = \{C_1, C_2, \ldots, C_n\}$. Moreover, let M be an IOTS obtained by merging states of the global observation tree and $M_1, M_2, \ldots, M_n$ be the IOTS models of the components obtained by projecting M onto their alphabets. Assume that for some witness trace, the component C_i in response to the input projection of the witness trace produces a trace α which is not in IOTS M_i. To refine the latter, we add this trace to the local observation tree U_i (obtained by projecting the global observation tree U), thus, $U_i := U_i \cup \{\alpha\}$ and merge states in the updated tree using the above given procedure to obtain a refined model M_i'. The refined model is used to check again the well-formedness of the current models.

4.3 Example

We illustrate the approach for inference of communicating components using the example shown in Figure 2. We infer a k-quotient of the product of the given system for $k = 1$. The global observation tree U obtained after applying the external input x is shown in Figure 3 (we use the actual components in Figure 2 to determine the reaction of the system).

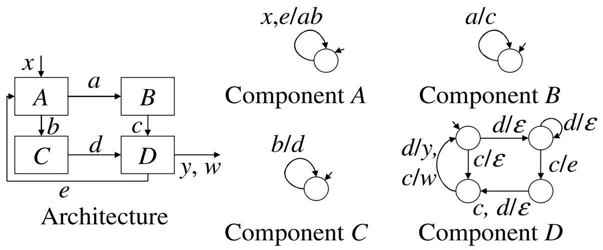


Fig. 2. A Modular System

The final state needs to be explored, so the input sequence xx is now applied to the system, the input x creates in the global observation tree the same trace, so we conclude that the stable states ε and $xx'aa'bcb'c'dd'y$ are 1-equivalent. The behavior exploration procedure terminates, and both stable states are merged, to obtain the IOTS M which is a 1-quotient of the product, it is shown in Figure 3 with the dashed transition labeled y. Next, we determine models by projecting the IOTS M. M_2 and M_3 are trace included by B and C respectively. M_1 and M_4 are shown in Figures 4 and 5.

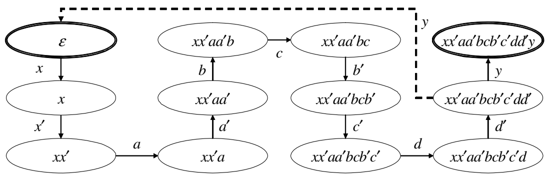


Fig. 3. The global observation tree U after applying x; stable states are depicted in bold

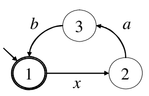


Fig. 4. M_1

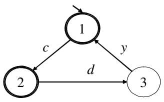


Fig. 5. M_4

Now we perform RA of the system composed of M_1, M_2, M_3 and M_4 to check whether it is well-formed and find that there exists an unspecified reception. The witness trace is $xx'aa'bb'd$. The components M_1 and M_3 are involved in the execution of this trace, their traces are $x'ab$ and $b'd$, respectively; the component M_4 has unspecified reception of d. The traces $x'ab$ and $b'd$ are already in the global observation tree, so only the component D has to be tested in isolation by applying the input d. We observe that the component has no output in response to this input, so the obtained trace d is added to its observation tree, as shown in the first row of Table 1. The table contains local observations trees and models of the components A and D as well as a witness trace for each version of the models. The table illustrates the iterative process of refining the two models based on detected unspecified receptions. In these figures, "?" and "!" are used to represent input and output respectively, as usual.

Using models in the last row of the table, we detect a livelock, the witness traces are xx', leading to livelock and $aa'bb'dd'cc'ee'$ which labels it. The corresponding projections of $xx'aa'bb'dd'cc'ee'$ are $x'abde'$, $a'c$, $b'd$, and $d'c'e$. To confirm this livelock in the real system, we proceed as follows. Assuming that in each component a trace executed consecutively three times indicates that the component cycles, we add to the above projections two more instances of the trace labeling a cycle of the component, obtaining $x'(abe')^3$, $(a'c)^3$, $(b'd)^3$, and $(d'c'e)^3$.

None of these traces is in the local observation trees, so we have to apply *xeee*, *aaa*, *bbb*, *dcdcdc* to the components A, B, C, and D, respectively. We use again the real components to obtain the following traces: *xabeabeabeab*, *acacac*, *bdbdbd*, and *dcedcwdce*. Only component D produces a new trace *dcedcwdce*, the current model of D expects *dcedcedce* instead, so we add the trace *dcedcwdce* to the observation tree and refine the model of D to obtain the IOTS shown in Table 2 which illustrates final model refinements.

The RA of the obtained models results now in the witness traces for races: $xx'aa'cc'bb'dd'y$ and $xx'aa'bb'dd'cc'ee'aa'bb'dd'cc'w$. In fact, the external input sequence x yields several traces in the product which differ in their output projections, y and w. We project the two traces to each component, and find that all the obtained traces are already in the corresponding local observation trees. The compositional problem is confirmed. With this report the procedure terminates. Even though the inferred models exhibit compositional problem, they are trace included in the actual models.

Table 1. Iterative model refinement

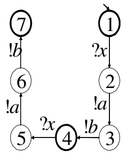
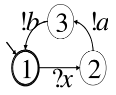
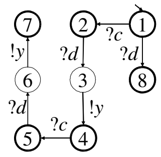
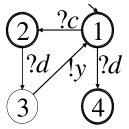
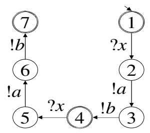
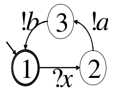
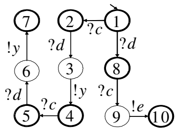
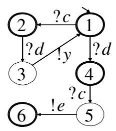
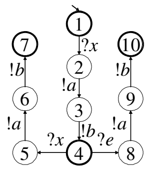
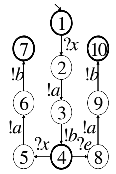
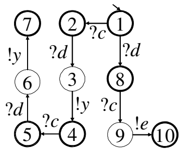
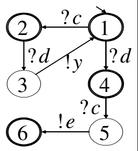
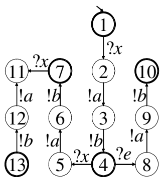
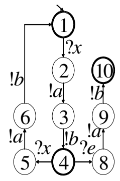
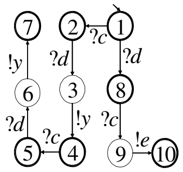
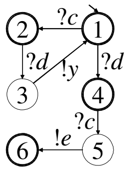
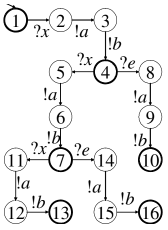
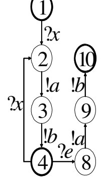
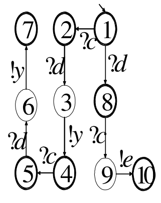
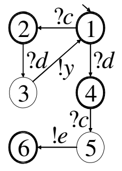
U_1	M_1	U_4	M_4	Witness traces
				*xx′aa′bb ′dd′c*
				*xx′aa′bb ′dd′cc′e*
				*xx′aa′cc ′bb′dd′y xx′aa′cc ′bb′dd′y x*
				*xx′aa′cc ′bb′dd′y xx′aa′bb ′dd′cc′e*
				*xx′aa′bb ′dd′cc′e e′aa′c*

Table 1. (*continued*)

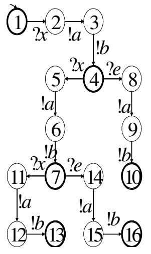
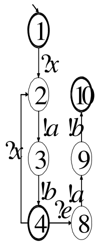
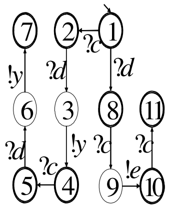
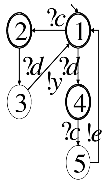
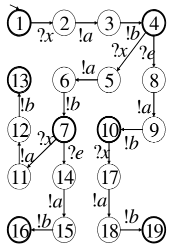
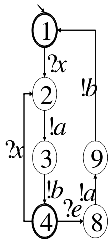
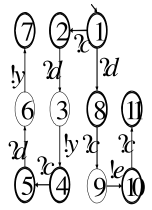
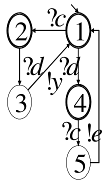
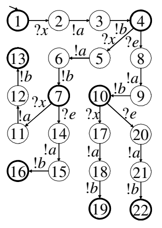
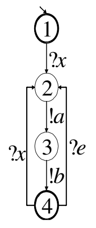
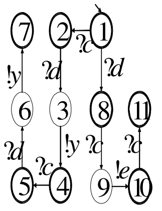
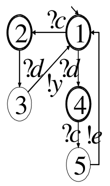
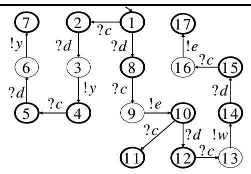
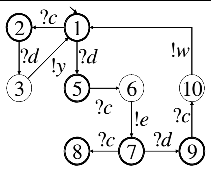
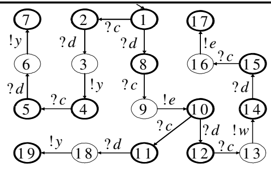
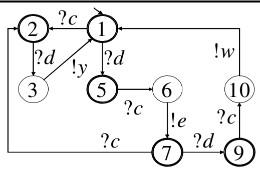
				*xx′aa′bb* *′dd′cc′e* *e′aa′cc′* *bb′dd′yx*
				*xx′aa′bb* *′dd′cc′e* *e′aa′bb′* *dd′cc′e*
				*xx′* and *aa′bb′d* *d′cc′ee′*

Table 2. Final Iterations

U_4	M_4	Witness traces
		*xx′aa′bb′dd′cc′ee* *′aa′cc′bb′d*
		*xx′aa′cc′bb′dd′y* and *xx′aa′bb′dd′cc′ee* *′aa′bb′dd′cc′w*

5 Related Work

Finite State Machine learning is widely addressed, specifically in the grammatical inference works, e.g., [9], [8] etc. Angluin presents a polynomial time algorithm [5] to infer regular language as deterministic finite automaton. She uses an equivalence oracle which provides a (minimal) counterexample. In our work, the state *k*-equivalence relation can be viewed as an approximated equivalence oracle. Learning and testing through model checking approach is used to infer a grey box system [15]. However, the upper bound on the number of states in the system is required to test conformance of the conjectured model to the actual system. On the other hand, the notion of *k*-quotient provides a means for inferring a variable size approximation without upper bound on the number of states.

The observation tree FSM used in this paper is, in fact, an input-output version of a prefix tree machine [13] used in various techniques for learning an automaton from positive examples. However, we do not require samples of the behavior given for inference, the samples (traces) are obtained by testing. At the same time, as in a common paradigm in learning from positive examples, the observation tree FSM is also iteratively merged. The rule used for state merging in the proposed approach is language containment (input-output trace inclusion). The exploration step ensures that the observation tree FSM contains *k*-equivalent states; these states have the same traces for *k* consecutive inputs. As in [14], the parameter *k* allows one to control the precision and complexity of the synthesized machine. However, differently from that work, the inferred *k*-quotient is a deterministic FSM.

There is much less work published on the inference of modular systems. The work [17] relies on observed traces to construct automata models of communicating components which are then model-checked using user defined properties. An object-flattening technique [11] is used to collect system behavior and then invariants are calculated on the behaviors to check against the new version of the system. This work is more related to regression testing. Moreover, the system behavior is observed while the system is running as in [17]. In this paper, we rely on testing communicating system by stimulating it through external inputs and then use the observations to obtain tunable approximated models. The verification of black box communicating system on the architectural level is also addressed recently [7]. Similar to the previous approaches, the system is monitored at runtime by instrumenting the middleware, no testing strategy is used. On the contrary, we infer models by testing and use them to check the system for compositional problems. Our past work [10] and [12] in this domain also concerned inference of components as finite state machines through testing. In fact, we were implicitly inferring a 1-quotient of each component in isolation without defining *k*-quotient. Moreover, the previous approaches focused on learning components separately, one component at a time; whereas in this paper, we proposed to test the integrated system avoiding thus unnecessary testing efforts to learn models only related to the composition.

6 Conclusion

In this paper, we offered a solution to the problem of modular system verification by blending together techniques of inference, testing, and reachability.

We first suggested the notion of an approximation of a Mealy machine, called *k*-quotient, where all *k*-equivalent states of the given machine are represented as a single state of the quotient, provided that at least one of them is on a path from the initial state which includes only pairwise *k*-distinguishable states. The precision of this approximation can be varied with the parameter *k*. We then proposed an approach to infer models of a modular system which starts with exhaustive (limited by some test length) testing of the integrated system and iterates between RA of intermediate models and pinpointed testing of components in isolation. A *k*-quotient of the modular system with observable internal actions in the form of an IOTS is used to infer initial models of components by projecting the quotient. RA of the models is next used to identify composition problems, such as unspecified receptions, livelocks, divergences, and races. A witness (diagnostic) trace is then used to test concerned components in isolation either to confirm that a problem exists in the real system or to obtain new observations. The models are then refined using new observations until the obtained models are well-formed.

The proposed approach relies on application of all external input sequences of length *k*, however, the parameter allows one to find a compromise between complexity of testing of the integrated system and precision of the resulting models. Moreover, the use of all input sequences of given length is completely avoided in testing a component in isolation, since only single diagnostic test is executed in each iteration. Another advantage of the approach is that inferred models capture the functionalities of components used in the given system; unused behaviors of components are not modeled.

As a future work, it would be interesting to investigate whether instead of testing a number of components in isolation based on a witness trace one would test just a subsystem consisting of these components to reduce the number of iterations needed to infer well-formed models. There are also a number of options for treating witness traces to update the global observation tree once the individual models are refined. This may help converge faster and shorten the RA process. It is known that reachability analysis can provide more than one witness traces, evidencing multiple problems in one step. The treatment of multiple traces at a time could be another improvement in the approach.

References

1. Huo, J., Petrenko, A.: Covering Transitions of Concurrent Systems through Queues. In: ISSRE, pp. 335–345 (2005)
2. Tretmans, J.: Test Generation with Inputs, Outputs and Repetitive Quiescence. Software - Concepts and Tools 17(3), 103–120 (1996)
3. Lynch, N., Tuttle, M.: An Introduction to Input/output Automata. CWI-Quarterly 2(3), 219–246 (1989)
4. Luo, G., Bochmann, G.v., Petrenko, A.: Test Selection Based on Communicating Nondeterministic Finite State Machines Using a Generalized Wp-Method. IEEE Transactions on Software Engineering (February 1994)
5. Angluin, D.: Learning Regular Sets from Queries and Counterexamples. Information and Computation 2, 87–106 (1987)

6. Godefroid, P.: Model Checking for Programming Languages Using VeriSoft. In: POPL, pp. 174–186 (1997)
7. Bertolino, A., Muccini, H., Polini, A.: Architectural Verification of Black-box Component-based Systems. In: Guelfi, N., Buchs, D. (eds.) RISE 2006. LNCS, vol. 4401, pp. 98–113. Springer, Heidelberg (2007)
8. Balcazar, J.L., Diaz, J., Gavalda, R.: Algorithms for learning finite automata from queries: A unified view. In: AALC, pp. 53–72 (1997)
9. Kearns, M.J., Vazirani, U.V.: An introduction to Computational Learning Theory. MIT Press, Cambridge (1994)
10. Li, K., Groz, R., Shahbaz, M.: Integration Testing of Components Guided by Incremental State Machine Learning. In: TAIC PART, pp. 231–247 (2006)
11. Mariani, L., Pezzè, M.: Behavior Capture and Test: Automated Analysis of Component Integration. In: ICECCS, pp. 292–301 (2005)
12. Shahbaz, M., Li, K., Groz, R.: Learning and Integration of Parameterized Components Through Testing. In: TestCom, pp. 319–334 (2007)
13. Cook, J.E., Wolf, A.L.: Discovering Models of Software Processes from Event-Based Data. ACM Trans. Softw. Eng. Methodol. 7(3), 215–249 (1998)
14. Biermann, A., Feldman, J.: On the Synthesis of Finite State Machines from Samples of their Behavior. IEEE Transactions on Computers 21(6), 592–597 (1972)
15. Elkind, E., Genest, B., Peled, D., Qu, H.: Grey-box Checking. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 420–435. Springer, Heidelberg (2006)
16. Petrenko, A., Yevtushenko, N.: Solving Asynchronous Equations. In: FORTE, pp. 231–247 (1998)
17. Hallal, H.H., Boroday, S., Petrenko, A., Ulrich, A.: A Formal Approach to Testing Properties in Causally Consistent Distributed Traces. Formal Aspects of Computing 18(1), 63–83 (2006)
18. Jéron, T., Morel, P.: Test Generation Derived from Model-Checking. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 108–121. Springer, Heidelberg (1999)

Test Plan Generation for Concurrent Real-Time Systems Based on Zone Coverage Analysis

Farn Wang[1,2] and Geng-Dian Huang[1]

[1] Dept. of Electrical Engineering, National Taiwan University, Taiwan, ROC
[2] Grad. Inst. of Electronic Engineering, National Taiwan University, Taiwan, ROC
farn@cc.ee.ntu.edu.tw

Abstract. The state space explosion due to concurrency and timing constraints of concurrent real-time systems (CRTS) presents significant challenges to the verification engineers. In this paper, we investigate how to use coverage techniques to generate efficient test plans for such systems. We first discuss how to use communicating timed automata to model CRTS. We present a new coverage technique, AZC (active zone coverage), based on the zone equivalence relation between states of CRTS. We discuss techniques to estimate AZC values of active zones represented in BDD-like diagrams. We explain how to construct zone trees and map their root-to-leaf paths to test cases. We then present an algorithm to generate test plans by prioritizing the test cases. The test plans that we generate can efficiently achieve full coverage in AZC. We have implemented our ideas with our TCTL model-checker RED. Experiment report with the Bluetooth L2CAP showed improvement of the coverage growth rate in the test plan execution.

1 Introduction

Although there have been several alternative technologies, e.g. model-checking and simulation, *testing* [4, 19, 29, 30] is still the major verification technique in the software industry over the last few decades. To verify a *system under test* (*SUT*), we need a set of *test cases*. Each test case specifies a sequence of input events and expected output events from the SUT. A *test plan* is a sequence of test cases to be executed in succession. A bad test plan may spend valuable time in repetitively testing equivalent behaviors and missing some other faulty behaviors. In contrast, a good test plan can methodically and efficiently execute test cases to give us high confidence of the SUT in a short time. At this moment, the generation of efficient test plans is still in the center of software testing research.

However, the complexity of new-generation *concurrent real-time systems* (*CRTS*) is driving the cost of testing to over fifty percent of the development budgets. Especially, for CRTS, there are infinitely many time instances in which an event can happen. Thus it is impossible to test all behaviors in either theory or practice. In the last few decades, the software industry has heavily relied on the wisdom of *coverage* techniques [29, 30] to maintain the high quality of software systems. Coverage techniques partition the behavior space of an SUT into finitely many equivalence classes. Coverage techniques can be used to measure how much of an SUT has been tested. In general, the more coverage you get, the more confidence you have in your SUT.

One coverage techique is the *visited-state coverage* [2,22] that uses the ratio of the number of visited states over the number of reachable states to measure the progress of test execution. Visited-state coverage has great discerning power since each state records the information necessary for the reaction to all future events from the states. However for CRTS, visited-state coverage is neither practical since the number of states is usually infinite [1]. In this work, we investigate how to use state-based coverage techniques for CRTS. In [1], a very fine partition of dense-time state-spaces is proposed. In this partition, an equivalence class is called a *region*. Given a description of a CRTS, the number of regions in a state-space is usually exponential to the description size. It is usually infeasible to cover all regions in a testing task. Another partition is with the equivalence classes of *zones* [14]. Zones can be constructed with on-the-fly techniques and usually partition a state-space in a coarse but precise enough granularity for the model-checking of dense-time models. In this work, we investigate how to adapt the zone technology for the efficient generation of test plans for CRTS. Specifically, we propose the following techniques.

- *Communicating timed automata* (*CTA*) [23,31,32] for the modeling of CRTS. A CTA may have many processes that interact through message channels and measure event intervals with dense-time clocks.
- *Active zone coverage (AZC) for the progress estimation of testing tasks for CRTS.* We first propose *zone-coverage* (*ZC*) for the measurement of progress of testing tasks for CRTS. We then propose a technique, based on inactive variable analysis, to enhance ZC to *active ZC* (*AZC*) for the construction of efficient test plans. With AZC, an equivalence class is called an *active zone*.
- *Test plan generation with priority for gains in AZC.* We first present an algorithm to construct zone trees that symbolically represent the branching structure of a CRTS. Then we discuss how to convert paths in the zone trees to linear test cases. Then we present a heuristic algorithm that prioritizes the test cases and construct a test plan. We prove that the test plan can always be constructed and visits all active zones.

We have implemented our ideas and experimented with the Bluetooth L2CAP [11, 32]. The experiment result shows that comparing with non-prioritized test plans, the prioritized test plans indeed improve the efficiency for high AZC coverage.

The rest of this paper is structured as follows. We review related work in section 2. We present our modeling language CTA in section 3. We explain some basic techniques for dense-time space manipulation and for zone forest construction in section 4. We present AZC in section 5. We explain how to generate a set of test cases for full AZC coverage in section 6. We present an algorithm to generate test plans with priority for high gains in AZC in section 7.

We report our experiment in section 8. Finally, we present the conclusion and discuss future work in section 9.

2 Related Work

A popular coverage technique for software testing is the *statement coverage* [6] which measures the proportion of already-executed statements during testing. The *transition coverage* [15] measures that of the executed transitions of the control flow machines. The *visited-state coverage* [2, 22] from the VLSI industry measures that of the visited states.

For dense-time systems, ACM, TCM, and RCM has been introduced to measure the progress of model-checking [32].

For testing dense-time systems, the popular *timed automata (TA)* [1] and its variations have been used as the formal specification languages [25]. Test case generation algorithms for untimed systems can also be applied by first discretizing the dense state-space. In [17], the discretization is achieved through digital clock automata. In [25], a timed automaton is discretized into a finite grid automaton with granularity 2^{-n} where n is the number of regions. In [7], a timed automaton is discretized at a sampling frequency $\frac{1}{|X|+2}$ where X is the set of clocks. These methods encounter the state space explosion problem even with small systems since they partition the state spaces with fine granularities.

In [12], the fastest diagnostic trace facility of UPPAAL [21] (a model checker for real-time systems) is used to generate time-optimal test cases. Three coverage techniques: *edge coverage*, *location coverage*, *definition-use pair coverage* for untimed systems are used.

In [20], real-time systems are modeled as event-recording automata. Then symbolic techniques are used to construct the reachability graph out of the equivalence-class graph. Finally, test cases are generated to cover all equivalence classes. In [12, 17, 20], test cases are generated to guarantee traditional coverage techniques, like arc coverage, domain analysis, location coverage, data-flow analysis.

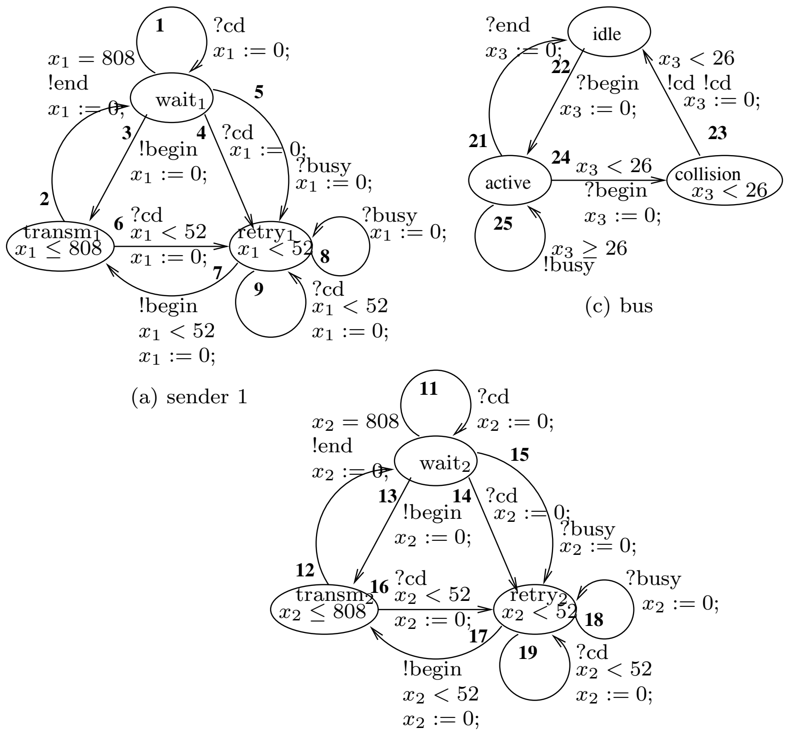
In [3], a state space exploration algorithm is proposed for efficiently computing the minimum cost of reaching a goal state in the model of *uniformly priced timed automata*. They used the cost metric to guide the state space exploration.

In [8], test cases are prioritized according to the rate of fault detection in regression testing. Such a technique has been implemented in Echelon for regression testing [24].

Huang and Wang [16] used symbolic techniques to automatically generate test cases, with coverage annotations for dense-time systems, with the CRD data-structures [28].

3 Communicating Timed Automata (CTA)

We first need to introduce our modeling language for CRTS. A *communicating timed automaton (CTA)* [23, 31, 32] is a set of *process timed automata (PTA)*,



(b) sender 2

Fig. 1. Specification of a bus-contending protocol

equipped with a finite set of dense-time clocks and synchronization channels. A PTA is structured as a directed graph whose nodes are *modes (control locations)* and whose arcs are *transitions*. The modes are labeled with *invariance conditions* while the transitions are labeled with *triggering conditions* and a set of clocks to be reset during the transitions. The invariance conditions and triggering conditions are Boolean combinations of inequalities comparing clocks with integers. At any moment, each PTA can stay in only one *mode* (or *control location*).

In the operation of a CTA, a minimal subset (called a *global transition*) of the PTA transitions can be triggered when the corresponding triggering conditions are satisfied and their input/output events are synchronized. Upon being triggered, all PTAs participating in the global transition instantaneously transit from one mode to another and resets some clocks to zero. In between transitions, all clocks in the CTA increase their readings at a uniform rate.

In figure 1, we have one bus process PTA and 2 sender PTAs for the modeling of a bus-contending protocol. The circles represent modes while the arcs represent transitions, which may be labeled with input/output events (e.g., `!begin`, ?`end`, ...), triggering conditions (e.g., $x < 52$), and assignments (e.g., $x := 0;$). For convenience, we have labeled the transitions with numbers. In the system, a sender process may synchronize through channel `begin` with the bus to start sending messages on the bus. While one sender is using the bus, the second one may also synchronize through channel `begin` to start placing messages on the bus and corrupting the bus contents. When this happens, the bus then signals bus collision (`cd`) to all the senders.

Due to page limit, we only give a brief definition of CTA. For detailed definition, please refer to [31,32]. For convenience, given a set Q of modes and a set X of clocks, we use $B(Q, X)$ as the set of all Boolean conjunctions of inequalities of the forms q and $x \sim c$, where $q \in Q$, $x \in X$, '$\sim$'$\in \{\leq, <, =, >, \geq\}$, and c is an integer constant. An element in $B(Q, X)$ is called *conjunctive* if the only Boolean operators in the element are conjunctions. $\mathbb{R}^{\geq 0}$ is the set of nonnegative real numbers.

Definition 1. **Process timed automata (PTA):** A PTA P is a tuple $\langle X, \Sigma, Q, I, \mu, E, \delta, \lambda, \tau, \pi\rangle$. X is a finite set of clocks. Σ is a finite set of synchronization channels. Q is a finite set of modes. $I \in B(Q, X)$ is the initial condition. $\mu : Q \mapsto B(\emptyset, X)$ defines the conjunctive invariance condition of each mode. E is the set of process transitions. $\delta : E \mapsto (Q \times Q)$ defines the source and destination modes of each process transition. $\lambda : (\Sigma \times E) \mapsto \mathbb{Z}$ defines the number of events sent and received at each process transition. When $\lambda(\sigma, e) \leq 0$, it means that process transition e receives $|\lambda(\sigma, e)|$ events through channel σ. When $\lambda(\sigma, e) > 0$, it means that process transition e sends $\lambda(\sigma, e)$ events through channel σ. $\tau : E \mapsto B(\emptyset, X)$ and $\pi : E \mapsto 2^X$ respectively define the conjunctive triggering condition and the clock set to reset of each transition. ■

Definition 2. **Communicating timed automata (CTA):** A CTA A of m processes is a tuple $\langle \Sigma, P_1, P_2, \ldots, P_m\rangle$, where Σ is the set of synchronization channels and for each $1 \leq p \leq m$, $P_p = \langle X_p, \Sigma, Q_p, I_p, \mu_p, E_p, \delta_p, \lambda_p, \tau_p, \pi_p\rangle$ is the PTA for process p and for each $1 \leq p < p' \leq m$, $Q_p \cap Q_{p'} = \emptyset$. ■

A *valuation* of a set is a mapping from the set to another set. Given an $\eta \in B(\bigcup_{1\leq p\leq m} Q_p, \bigcup_{1\leq p\leq m} X_p)$ and a valuation ν of $\bigcup_{1\leq p\leq m}(X_p \cup Q_p)$, we say ν *satisfies* η, in symbols $\nu \models \eta$, iff η is evaluated *true* when the variables in η are interpreted according to ν.

Definition 3. **States:** Suppose we are given a CTA $A = \langle \Sigma, P_1, P_2, \ldots, P_m\rangle$ such that for each $1 \leq p \leq m$, $P_p = \langle X_p, \Sigma, Q_p, I_p, \mu_p, E_p, \delta_p, \lambda_p, \tau_p, \pi_p\rangle$. A state ν of A is a valuation of $\bigcup_{1\leq p\leq m}(X_p \cup Q_p)$ with the following constraints.

- For each $q \in \bigcup_{1\leq p\leq m} Q_p$, $\nu(q) \in \{\mathit{false}, \mathit{true}\}$. Moreover for each $1 \leq p \leq m$, there is exactly a $q \in Q_p$ such that
$$\nu(q) \wedge \forall q' \in Q - \{q\}(\neg\nu(q')).$$
Given $q \in Q_p$, if $\nu(q)$ is true, we denote q as $\mathtt{mode}_p(\nu)$.
- For each $x \in \bigcup_{1\leq p\leq m} X_p$, $\nu(x) \in \mathbb{R}^{\geq 0}$ such that $\nu \models \bigwedge_{1\leq p\leq m} \mu_p(\mathtt{mode}_p(\nu))$.

For any $t \in \mathbb{R}^{\geq 0}$, $\nu + t$ is a state identical to ν except that for every clock $x \in \bigcup_{1\leq p\leq m} X_p$, $(\nu + t)(x) = \nu(x) + t$. ■

A *global transition* γ of a CTA is a mapping from process indices p, $1 \leq p \leq m$, to $E_p \cup \{\perp\}$, where $\perp$ means no transition (i.e., a process does not participate in γ).

A legitimate global transition has to be *synchronized*, that is, each output event from a process is received by exactly one unique corresponding process with a matching input event. In arithmetic, that is $\forall e \in E, \sum_{1 \le p \le m; \gamma(p) \neq \perp} \lambda(e, \gamma(p)) = 0$. Moreover, to be compatible with the popular interleaving semantics, we require that a global transition must be minimal, i.e., it cannot be broken down to two non-empty synchronized global transitions. For example, in figure 1, we may have a legitimate global transition γ with $\gamma(1) = 1$, $\gamma(2) = \perp$, and $\gamma(3) = 6$. In contrast, another global transition γ' with $\gamma'(1) = \perp$, $\gamma'(2) = \perp$, and $\gamma'(3) = 6$ is not. In the following, whenever we say "global transition", we actually mean "legitimate global transition" for briefness. Given a CTA A, we let $\Gamma(A)$ be the set of legitimate global transitions of A.

Given two states ν, ν' and $t \in \mathbb{R}^{\ge 0}$, $\nu \xrightarrow{t} \nu'$ iff $\nu' = \nu + t$. Also $\nu \xrightarrow{\gamma} \nu'$ iff

- $\nu \models \bigwedge_{1 \le p \le m; \gamma(p) \neq \perp} \tau_p(\gamma(p))$, and
- ν is identical to ν' except that for all $1 \le p \le m$,
  - if $\gamma(p) = \perp$, $\mathtt{mode}_p(\nu) = \mathtt{mode}_p(\nu')$ and $\forall x \in X_p(\nu(x) = \nu'(x))$.
  - if $\gamma(p) \neq \perp$, then
    - $\delta(\gamma(p)) = (\mathtt{mode}_p(\nu), \mathtt{mode}_p(\nu'))$,
    - $\forall x \in (X_p \cap \pi_p(\gamma(p)))(\nu'(x) = 0)$, and
    - $\forall x \notin (X_p \cap \pi_p(\gamma(p)))(\nu'(x) = \nu(x))$.

In the verification of a CRTS A, usually we are given a specification formula ϕ and want to check whether ϕ is violated in the test execution. We assume the framework of safety analysis, in which the specification is a safety predicate in $\phi \in B(\bigcup_{1 \le p \le m} Q_p, \bigcup_{1 \le p \le m} X_p)$.

4 Regions, Zones, and Zone Forests

In [32], *RCM (region coverage metric)* is proposed to measure the progress of reachability analysis. Specifically, RCM is an adaptation of the *visited-state coverage* technique from VLSI verification technology, which measures the visited states in an FSM. Instead of measuring the visited states directly, region-equivalence relation [1] can be used to partition the dense-time state-space into a finite set of equivalent classes. Each such equivalence class is called a *region*. Let $C_{A:\phi}$ be the biggest timing constant used in a CTA A and a TCTL specification ϕ. Given a real number t, we let $\mathit{frac}(t) = t - \lfloor t \rfloor$ be the fractional part of t. Two states ν, ν' are in the same region if and only if they satisfy the following constraints.

- Every discrete variable has the same value in the two states.
- For every clock variable x, if either $\nu(x) \le C_{A:\phi}$ or $\nu'(x) \le C_{A:\phi}$, then $\lfloor \nu(x) \rfloor = \lfloor \nu'(x) \rfloor$.
- For every two clocks x_1 and x_2, if $\nu(x_1) \le C_{A:\phi}$ and $\nu(x_2) \le C_{A:\phi}$, then
$$\mathit{frac}(\nu(x_1)) \le \mathit{frac}(\nu(x_2)) \text{ if and only if } \mathit{frac}(\nu'(x_1)) \le \mathit{frac}(\nu'(x_2)).$$

It can be shown that for any subformula ϕ' of ϕ and two states ν, ν' in the same region, ν satisfies ϕ' if and only if ν' satisfies ϕ' [1]. With RCM, we measure the visited regions instead of the concrete states. It is shown that RCM has better discerning power in timing bugs than both TCM (trigger coverage metric) and ACM (arc coverage metric) [32].

However, the number of regions is tremendous for any non-trivial CRTS. It is not infeasible to trace through all the reachable regions. In this work, we propose a new coverage technique, called *zone coverage* (*ZC*), for the testing of CRTS. For efficient verification of dense-time systems, *zones*, instead of regions, have very often been used as the basic unit in state-space manipulation [14]. A zone ζ is a state-space characterizable with a conjunction of atoms (positive or negative literals) in either of the following two forms.

- q_p, for some $1 \leq p \leq m$ and $q_p \in Q_p$.
- $x - x' \sim c$ for clock differences, where x and x' are clocks or 0, '$\sim$' $\in \{\leq, <\}$, and c is an integer.

A region is a smallest zone that is not properly contained by a different region. A zone may contain many regions. Although the number of zones can be of the same complexity as that of regions, in practice verification tools implemented with zones perform much better than those with regions [21, 27, 34].

Our symbolic trace computation is based on a well-discussed procedure, called `post()`, to compute a symbolic post-condition of a zone after a global transition and time-progress [14]. Given a zone ζ and a global transition γ,

$$\texttt{post}(\zeta, \gamma) = \{\nu' | \exists \nu \in \zeta \exists t \in \mathcal{R}^{+} \exists \nu'' (\nu \xrightarrow{\gamma} \nu'' \wedge \nu'' \xrightarrow{t} \nu')\}.$$

Note that the results of the post-condition procedure can also be represented as zones. With relation `post()`, we can construct a *zone forest* for a CTA and a TCTL specification. A zone forest is a set $\langle V, R, K \rangle$ of trees such that V is a set of zones, R is a set of initial zones, and K is a set of triples (ζ, γ, ζ') with $\zeta \in V$, $\zeta' \in V$, and $\zeta' = \texttt{post}(\zeta, \gamma)$.

We can use the following procedure to construct a zone forest for a CTA and a TCTL specification.

```
(1)  ZoneForest($A, \phi$) {
(2)    Rewrite the initial condition of $A$ in DNF $\zeta_1 \vee \ldots \vee \zeta_n$.
(3)    $R := \{\zeta_1, \zeta_2, \ldots, \zeta_n\}$; $V := R$; $\Phi := R$; $K := \emptyset$; $\psi := \bigvee_{\zeta \in V} \zeta$.
(4)    While $\Phi \neq \emptyset$, {
(5)      Pick a zone $\zeta$ from $\Phi$; Let $\Phi := \Phi - \{\zeta\}$.
(6)      For each $\gamma \in \Gamma(A)$, {
(7)        Let $\zeta' :=$ post($\zeta, \gamma$).
(8)        If $\zeta' \wedge \neg\psi$ is satisfiable, {
(9)          $\psi := \psi \vee \zeta'$; $\Phi := \Phi \cup \{\zeta'\}$;
(10)         $V = V \cup \{\zeta'\}$; $K := K \cup \{(\zeta, \gamma, \zeta')\}$.
(11)   } } }
(12)   return $\langle V, R, K \rangle$.
(13) }
```

Note that at line (5), we randomly pick a zone from Φ. In practice, generating a complete zone forest can still be very expensive since the number of zones can be exponential to the sizes of the given CTA and specification. So sometimes we may want to compromise with a partial zone forest. In that case, it is important for us to know that the zones in the forest cover an appropriate portion of the behaviors of the SUT. In section 8, we report experiment with various strategies in expanding the frontier of the zone forest construction.

Lemma 1. *For a CTA A and a CTL formula ϕ, if* $\texttt{ZoneForest}(A, \phi)=\langle V, R, E\rangle$*, then for every state ν that can be reached from an initial state of A, there is a $\zeta \in V$ such that $A, \nu \models \zeta$.* ■

5 Active Zone Coverage (AZC) for CRTS

Lemma 1 suggests that we can use the zones in a zone forest to estimate the coverage of equivalent state classes in the testing of a CRTS. Straightforwardly, we can count the zones in $\texttt{ZoneForest}(A, \phi)$ to estimate the coverage of a testing task. However, each zones in $\texttt{ZoneForest}(A, \phi)$ may include different number of regions. Based on the techniques discussed in [32] for estimating the number of regions included in a zone, we can define *zone coverage* (*ZC*) that estimates the ratio of number of regions in visited zones over those in reachable zones. Specifically, given a set Z of visited zones, we use $\texttt{RCM}_{A:\phi}(\bigvee_{\zeta\in Z}\zeta)$ to denote the estimation of coverage for those regions in zones in Z for CTA A and TCTL formula ϕ. Thus $\texttt{RCM}_{A:\phi}(\bigvee_{\zeta\in Z}\zeta)$ seems a reasonable definition for zone coverage estimation.

However we can further improve the above-mentioned ZC techniques without sacrificing its discerning power. We present an adaptation of ZC, called *active ZC* (*AZC*), to cut down the number of regions. The idea is to use inactive variable analysis. A variable x is *inactive* in a state if along all computations from the state, x is never read before being written to. Thus the values of inactive variables in a state do not affect the behaviors of a system. There are model-checkers that use inactive variable analysis for state-graph reduction [27]. To check if a variable x is inactive in a state ν, we can construct a CTL formula to characterize those states in which x is active. First, we let $\texttt{Read}(x)$ be the disjunction of the following two constraints.

- The disjunction of invariance conditions (of modes of PTAs) in which x is referenced.
- The disjunction of triggering conditions of all process transitions in which x is read before being writing to. Note that if x appears in both the $\tau()$ and the $\pi()$ components of a process transition, then it is considered read before being written to.

Then we also define $\texttt{PureWrite}(x)$ as the disjunction of the triggering conditions of all transitions e with $x \in \pi_p(x)$ for some process p. With these two conditions, we can now construct the following CTL formula [5] to characterize the states in which x is active.

$$\texttt{active}(x) \equiv \exists(\neg\texttt{PureWrite}(x))\mathcal{U}\texttt{Read}(x).$$

Thus we can use $\neg\texttt{active}(x)$ to characterize those states in which x is inactive.

Now we define *active zones* as new equivalence classes. Given a zone ζ and a variable x, we say that x is inactive in ζ if and only if for x is inactive in every state in ζ. A zone ζ is active if and only if no inactive variable x in ζ appears in the representation of ζ. Given a zone ζ with inactive variables $x_1, \ldots, x_n$, the smallest active zone that contain ζ can be constructed as $\exists x_1 \exists x_2 \ldots \exists x_n(\zeta)$. For convenience, we let $\texttt{active}(\zeta)$ be the smallest active zone that contains ζ. We can establish the following lemma.

Lemma 2. *Given a CTA A, a subformula ϕ' of a TCTL specification ϕ, and a zone ζ, all states of A in ζ satisfies ϕ' if and only if all states in* $\texttt{active}(\zeta)$ *of A satisfies ϕ'.* ■

It is clear that $\texttt{active}(\zeta)$ can be larger than ζ. To cover a whole reachable state-space, we may need fewer active zones than regular zones. Thus given a set Z of visited zones, we define

$$\texttt{AZC}_{A:\phi}(Z) = \texttt{RCM}_{A:\phi}(\textstyle\bigvee_{\zeta \in Z} \texttt{active}(\zeta))$$

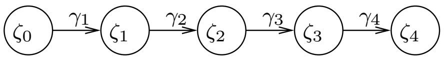
as the value of Z in the new *active zone coverage* (*AZC*).

6 Test Cases from a CTA

Suppose we are given a path through zones $\zeta_0, \zeta_1, \ldots, \zeta_n$ in a zone forest such that for each $0 \leq i < n$, $\zeta_{i+1} = \texttt{post}(\zeta_i, \gamma_{i+1})$. We can represent the path with the following notations.

$$\zeta_0 \xrightarrow{\gamma_1} \zeta_1 \xrightarrow{\gamma_2} \cdots \xrightarrow{\gamma_n} \zeta_n$$

We call a path in this representation a *symbolic trace*. In figure 2, we show such a symbolic trace of the bus-contending protocol (in figure 1). The trace describes that the two senders both send messages to the bus. After detecting the collision, A_3 retransmits its message.


$$\begin{tikzcd}\zeta_0 \arrow[r, "\gamma_1"] & \zeta_1 \arrow[r, "\gamma_2"] & \zeta_2 \arrow[r, "\gamma_3"] & \zeta_3 \arrow[r, "\gamma_4"] & \zeta_4\end{tikzcd}$$

$\zeta_0 : \texttt{mode}_1 = \texttt{idle} \wedge \texttt{mode}_2 = \texttt{wait} \wedge \texttt{mode}_3 = \texttt{wait}$

$\zeta_1 : \texttt{mode}_1 = \texttt{active} \wedge \texttt{mode}_2 = \texttt{wait} \wedge \texttt{mode}_3 = \texttt{transmit} \wedge x_1 = x_3$

$\zeta_2 : \texttt{mode}_1 = \texttt{collision} \wedge \texttt{mode}_2 = \texttt{transmit} \wedge \texttt{mode}_3 = \texttt{transmit} \wedge$
$x_1 < 26 \wedge x_2 < 26 \wedge x_1 = x_2 \wedge x_3 < 52 \wedge x_3 - x_1 < 26 \wedge x_3 - x_2 < 26$

$\zeta_3 : \texttt{mode}_1 = \texttt{idle} \wedge \texttt{mode}_2 = \texttt{retry} \wedge \texttt{mode}_3 = \texttt{retry} \wedge x_2 < 52 \wedge x_3 < 52 \wedge x_2 = x_3$

$\zeta_4 : \texttt{mode}_1 = \texttt{active} \wedge \texttt{mode}_2 = \texttt{wait} \wedge \texttt{mode}_3 = \texttt{transmit} \wedge$
$x_1 < 52 \wedge x_2 < 52 \wedge x_3 < 52 \wedge x_1 - x_2 \leq 0 \wedge x_1 = x_3 \wedge x_3 - x_2 \leq 0$

$\gamma_1(1) = 1\text{‘}\gamma_1(2) = \perp\text{‘}\gamma_1(3) = 6$

$\gamma_2(1) = 4\text{‘}\gamma_2(2) = 6\text{‘}\gamma_2(3) = \perp$

$\gamma_3(1) = 5\text{‘}\gamma_3(2) = 11\text{‘}\gamma_3(3) = 11$

$\gamma_4(1) = 1\text{‘}\gamma_4(2) = \perp\text{‘}\gamma_4(3) = 12$

Fig. 2. A path of the bus-contending protocol

Given a symbolic trace $\zeta_0 \xrightarrow{\gamma_1} \zeta_1 \xrightarrow{\gamma_2} \cdots \xrightarrow{\gamma_n} \zeta_n$, we may write $\zeta_0 \rightsquigarrow \zeta_n$. We borrow the techniques of generating a test case from the a symbolic trace from [16]. Given a symbolic trace of a CTA, we can extract a test case in TTCN format [9] for testing the CTA. A test case consists of a sequence of input events and expected output events annotated with symbolic timing-constraints between events. The global transitions in a symbolic trace correspond to the events in the test case. The zone descriptions in between transitions correspond to timing constraints between events. For the second sender process in figure 2, we can convert the path in figure 2 to the following test case in TTCN format.

(1)	START Time	/*start a clock called Time*/
(2)	?begin	/*receive a "begin" message*/
(3)	READTIMER Time(t_1)	/*store current reading of Time in t_1*/
(4)	!collision	/*send a "collision" message*/
(5)	READTIMER Time(t_2)	/*store current reading of Time in t_2*/
(6)	$[t_2 - t_1 < 52]$	/*check if $t_2 - t_1 < 52$*/
(7)	?begin	/*receive a "begin" message*/
(8)	READTIMER Time(t_3)	/*store current reading of Time in t_3*/
(9)	$[t_3 - t_2 < 52]$	/*check if $t_3 - t_2 < 52$*/

There is a global clock called `Time`. There are two TTCN commands, `START` (to start the ticking of a clock from zero) and `READTIMER()` (to read the current reading of a clock and save the reading in a variable). The test case can be used to check if a sender retries the transmission within 52 time unit after a bus collision is detected. Given a symbolic trace ρ, we denote the test case constructed out of ρ as `testcase`(ρ). A symbolic trace in `ZoneForest`(A, ϕ) is *root-to-leaf* if its first zone is a root and its last zone is a leaf.

7 Prioritized Test Plan Generation

Based on the materials in the last two sections, we can now define a set of test cases out of the `ZoneForest`. Given a CRTS A and a safety predicate ϕ, we denote the set of root-to-leaf symbolic traces in `ZoneForest`(A, ϕ) as `Rt2Lves`(A, ϕ). We let `TestCases`(A, ϕ) be the following set.

$$\texttt{TestCases}(A, \phi) = \{\texttt{testcase}(\rho) \,|\, \rho \in \texttt{Rt2Lves}(A, \phi)\,\}$$

A test plan that executes test cases $\theta_1, \ldots, \theta_n$ in sequence can be denoted as sequence $\theta_1 \ldots \theta_n$.

Different test plans may increase the AZC value with different efficiencies. In this section, we want to generate test plans that increase the AZC value with the maximum efficiency. Given a symbolic trace $\rho = \zeta_1\gamma_2\zeta_2\gamma_3 \ldots \zeta_n$, we let the disjunction of the zones in ρ be denoted as `PathCons`(ρ) $= \zeta_1 \vee \ldots \vee \zeta_n$. Also for convenience, given a symbolic trace ρ and a test case such that `testcase`(ρ) $= \theta$, we also write `PathCons`(θ) $= \zeta_1 \vee \ldots \vee \zeta_n$. Given two sequences Θ and Θ', their concatenation is denoted as $\Theta\Theta'$. We have the following procedure for test plan generation.

```
(1) TestPlan($A, \phi$) {
(2)    $\Phi :=$ Rt2Lves($A, \phi$); let $\Theta$ be an empty sequence; $\psi := \mathit{false}$.
(3)    While $\Phi \neq \emptyset$, {
(4)       Pick a $\rho \in \Phi$ such that for all $\rho' \in \Phi$,
(5)          AZC(PathCons($\rho$) $\vee \psi$) $\geq$ AZC(PathCons($\rho'$) $\vee \psi$).
(6)       $\Phi := \Phi - \{\rho\}$; $\Theta := \Theta$testcase($\rho$); $\psi := \psi \vee$ PathCons($\rho$).
(7)    }
(8)    return $\Theta$;
(9) }
```

We can prove that the returned test plan of `TestPlan`(A, ϕ) can reach all the active zones that are reachable from the initial states.

Lemma 3. *For any CTA A and safety predicate ϕ, procedure* `TestPlan`(A, ϕ) *terminates. When it terminates, it returns a test plan that traverses all active zones that are reachable from the initial states of A.*

Proof: Note that in procedure `ZoneForest`(A, ϕ), each node is added to the tree if it has some regions that are not covered by other nodes. This means that at each iteration of the loop at statement (3) in procedure `TestPlan`(), we cover some new regions that have not been covered before. Since the number of regions is finite, we know the loop at statement (3) eventually terminates.

Also when the loop terminates, we have added all test cases in `TestCases` (A, ϕ) to Θ. Since the test cases are derived from all root-to-leaf symbolic traces that together cover all reachable active zones, we know that the test plan also covers all reachable active zones. ■

Note that this procedure, `ZoneForest`(A, ϕ), may not generate the shortest test plan in achieving an AZC threshold. But it works with a greedy heuristic to try to get the best AZC gain in every decision point. The wisdom from NP problems [10] is that in practice, very often such a heuristics may still lead to acceptable solutions for difficult problems like set cover and bin packing.

8 Experiment

We have implemented our ideas in **RED** [26, 31, 28], a model-checker/simulator for CTAs and linear hybrid systems based on a BDD-like diagram called Clock-Restriction Diagram. We have experimented with the Philips Audio Protocol [13, 18] and Bluetooth L2CAP [11, 32]. Due to page-limit, we only report the experiment with Bluetooth L2CAP. Experiment data is collected on a 3.2 GHz PC running Red Hat Linux 9.0. In subsection 8.1, we first discuss the performance of procedure `ZoneForest`() for AZC gain with respect to the following three exploration strategies: DFES (depth-first exploration strategy), BFES (breadth-first exploration strategy), and CBES (AZC coverge-based exploration strategy). Then in subsection 8.2, we report the improvement in AZC coverage growth rate with procedure `TestPlan`().

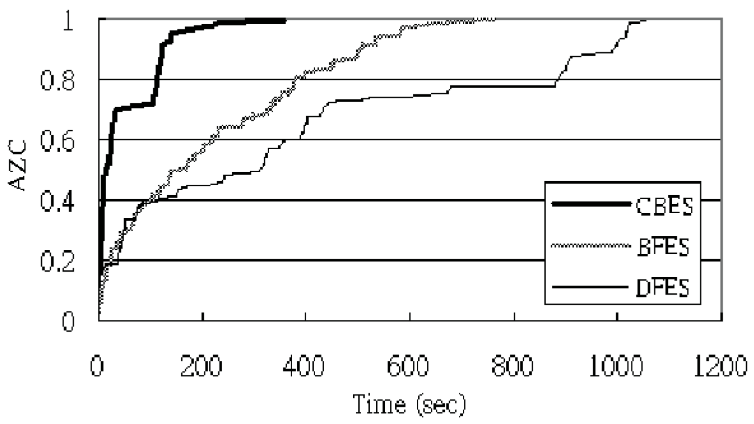


Fig. 3. AZC growth rate in symbolic state space exploration

Table 1. Experiment result of symbolic state space exploration

Strategy	CPU time(sec)	Memory(Kbyte)	# Symbolic trace
DFES	1055	7411	1957
BFES	762	6378	1815
CBES	359	4920	1610

8.1 Coverage Estimations for Strategies in Procedure `ZoneForest()`

We now report our experiment with procedure `ZoneForest`(). We applied DFES, BFES, and CBES to explore the whole active zone space of the Bluetooth L2CAP specification. In figure 3, we show the growth curves in AZC values with strategies DFES, BFES, and CBES. The AZC value grows much faster with CBES.

Table 1 shows the performance statistics of procedure `ZoneForest`(). "CPU time" is the time spent on exploring the whole active zone space in seconds. "Memory" is the memory used by the procedure in kilobytes. "# Symbolic trace" is the number of symbolic traces in the constructed trees of active zones. The data shows that CBES outperforms DFES and BFES against the L2CAP benchmark in the efficiency of AZC gain. Also, CBES uses the least resources (in both time and memory) and generated the fewest symbolic traces.

In summary, the experiment data shows that CBES could be promising in helping us achieving high coverage in state space exploration with limited resources.

8.2 Coverage Growth Rates in Procedure `TestPlan()`

In this subsection, we report our experiment with procedure `TestPlan`(). In table 2, we show six test plans, denoted as DFES_no, BFES_no, CBES_no, DFES_prior, BFES_prior, and CBES_prior. The six plans come from the combination of the choice of strategies in procedure `ZoneForest`() and the two options

Table 2. Six test plans and their strategy combinations

`TestPlan()` \ `ZoneForest()`	DFES	BFES	CBES
Not prioritized	DFES_no	BFES_no	CBES_no
Prioritized	DFES_prior	BFES_prior	CBES_prior

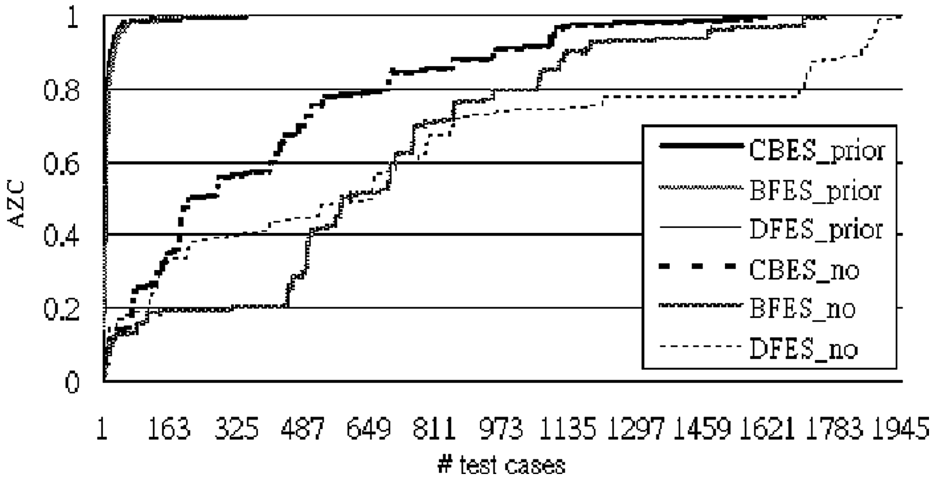


Fig. 4. AZC growth rate of the six test plans

in generating the test plans. DFES, BFES, and CBES are the choices for strategies in procedure `ZoneForest()`. With plans DFES_no, BFES_no, and CBES_no, we generate test plans in the 2nd step by simply enumerating the symbolic traces according to the depth-first order of the leaves of the traces in the zone forest. With DFES_prior, BFES_prior, and CBES_prior, we prioritized all symbolic traces in the 2nd step according to their AZC gain estimations.

To reduce the testing cost, it is desirable to achieve high coverage with few test cases. In figure 4, we show the growth curves of AZC values for the six test plans. DFES_prior, BFES_prior, and CBES_prior achieved 100% in AZC respectively after the 393rd, 413th, and 407th test cases were applied. The growth curves in AZC of the prioritized test plans with DFES_prior, BFES_prior, and CBES_prior, are similar in shape. In fact, in the chart, their curves almost coincide and reach the full coverage quickly.

On the other hand, with the non-prioritized test plans, including DFES_no, BFES_no, and CBES_no, the coverage curves grow much slower. Moreover, all test cases have to be executed in order to reach the full AZC. Since there are 1957 test cases for DFES_no, 1815 for BFES_no, and 1610 for CBES_no, significant testing time could be saved by using the prioritized test plans instead of the non-prioritized plans. The experiment data shows good promise of our techniques for the testing of CRTS.

Finally, test plan DFES_prior uses the least number of test cases because the test cases in DFES_prior tend to be longer than those in the other test plans.

9 Conclusion

In this paper, we propose techniques to model CRTS, to evaluate the progress of test execution with AZC, and test plan generation that gives priority to AZC gain. The experiment shows that the prioritized test plans can achieve high coverage more efficiently than the non-prioritized ones. Knowledge obtained in this work does point out some new directions for future research. For example, we assumed that the execution of every test case incurs the same cost (for examples, budget or time). In practice, some test cases may cost more than the others. It would be interesting to see how the factor of test case execution cost could be incorporated into our framework so that our test plan generation procedure could help in the accurate management of the verification budget.

Acknowledgment

The authors would like to thank the reviewers of TESTCOM/FATES 2008 for their helpful suggestions for improving this paper.

References

1. Alur, R., Courcoubetis, C., Dill, D.L.: Model Checking for Real-Time Systems. In: IEEE LICS (1990)
2. Bening, L., Foster, H.: Principles of Verifiable RTL Design: a Functional Coding Style Supporting Verification Processes in Verilog, 2nd edn. Kluwer Academic Publishers, Dordrecht (2001)
3. Behrmann, G., Fehnker, A., Hune, T.S., Larsen, K.G., Pettersson, P., Romijn, J.: Efficient Guiding Towards Cost-Optimality in UPPAAL. In: Margaria, T., Yi, W. (eds.) ETAPS 2001 and TACAS 2001. LNCS, vol. 2031, Springer, Heidelberg (2001)
4. Bochmann, G.V., Petrenko, A.: Protocol Testing: Review of Methods and Relevance for Software Testing. In: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis (1994)
5. Clarke, E., Emerson, E.A.: Design and Synthesis of Synchronization Skeletons using Branching-Time Temporal Logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, Springer, Heidelberg (1982)
6. Cheng, K.-T., Krishnakumar, A.S.: Automatic Functional Test Generation Using the Extended Finite State Machine Model. In: Proceedings of the 30th Design Automation Conference, June 1993, pp. 86–91 (1993)
7. En-Nouaary, A., Dssouli, R., Khendek, F.: Timed Wp: Testing Real-time Systems. In: IEEE Transactions on Software Engineering, vol. 29(11), IEEE Computer Society, Los Alamitos (2002)
8. Elbaum, S., Malishevsky, A.G., Rothermel, G.: Test Case Prioritization: A Family of Empirical Studies. IEEE Transactions on Software Engineering archive 28(2) (2002)

9. ETSI: Methods for Testing and Specification (MTS) The Testing and Test Control Notation version 3. ETSI ES 201873, parts 1 to 7, v3.1.1, (2005-06)
10. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman, New York (1983)
11. Haartsen, J.: Bluetooth Specification, version 1.0. `http://www.bluetooth.com/`
12. Hessel, A., Larsen, K.G., Nielsen, B., Pettersson, P., Skou, A.: Time-Optimal Real-Time Test Case Generation Using Uppaal. In: Petrenko, A., Ulrich, A. (eds.) FATES 2003. LNCS, vol. 2931, Springer, Heidelberg (2004)
13. Ho, P.-H., Wong-Toi, H.: Automated Analysis of an Audio Control Protocol. In: Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, Springer, Heidelberg (1995)
14. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic Model Checking for Real-Time Systems. In: IEEE LICS 1992 (1992)
15. Hoskote, Y., Moundanos, D., Abraham, J.A.: Automatic Extraction of the Control Flow Machine and Application to Evaluating Coverage of Verification Vectors. In: Proceedings of the Int'l Conference on Computer Design, October 1995, pp. 532–537 (1995)
16. Huang, G.-D., Wang, F.: Automatic Test Case Generation with Region-Related Coverage Annotations for Real-time Systems. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, Springer, Heidelberg (2005)
17. Krichen, M., Tripakis, S.: Black-box Conformance Testing for Real-Time Systems. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, Springer, Heidelberg (2004)
18. Larsen, K.G., Pettersson, P., Yi, W.: Diagnostic Model-Checking for Real-Time Systems. In: Proceedings of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems, New Brunswick, New Jersey, October 22-24 (1995)
19. Lee, D., Yannakakis, M.: Principles and Methods of Testing Finite State Machines - A Survey. Proceedings of The IEEE 84(8), 1090–1123 (1996)
20. Nielsen, B., Skou, A.: Automated Test Generation from Timed Automata. International Journal on Software Tools for Technology Transfer (STTT) 4 (2002)
21. Pettersson, P., Larsen, K.G.: UPPAAL2k. Bulletin of the European Association for Theoretical Computer Science 70, 40–44 (2000)
22. Rashinkar, P., Paterson, P., Singh, L.: System-on-a-chip Verificatoin, Methodology and Techniques. Kluwer Academic Publishers, Dordrecht (2001)
23. Shaw, A.: Communicating Real-time State Machines. IEEE Transactions on Software Engineering 18(9) (September 1992)
24. Srivastava, A., Thiagarajan, J.: Effectively prioritizing tests in development environment. In: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis (2002)
25. Springintveld, J., Vaandrager, F., D'Argenio, P.R.: Testing Timed Automata. Theoretical Computer Science 254(1-2) (2001)
26. Wang, F.: Symbolic Verification of Complex Real-Time Systems with Clock-Restriction Diagram. In: Proceedings of FORTE, Cheju Island, Korea (August 2001)
27. Wang, F.: Efficient Verification of Timed Automata with BDD-like Data-Structures. STTT (Software Tools for Technology Transfer) 6(1) (June 2004); special issue for the 4th VMCAI, LNCS. vol. 2575. Springer, Heildelberg (January 2003)

28. Wang, F.: Symbolic Parametric Safety Analysis of Linear Hybrid Systems with BDD-like Data-Structures. In: IEEE Transactions on Software Engineering, vol. 31(1), pp. 38–51. IEEE Computer Society, Los Alamitos (2005); A preliminary version is in proceedings of 16th CAV, LNCS vol. 3114. Springer, Heildelberg (2004)
29. Weyuker, E.J.: In Defense of Coverage Criteria. In: Proceedings of the 11th ACM/IEEE International Conf. on Software Engineering (ICSE), Pittsburgh, Pa (May 1989)
30. Weyuker, E.J.: How to Judge Testing Progress. Journal of Information and Software Technology 45(5), 323–328 (2004)
31. Wang, F., Huang, G.-D., Yu, F.: Symbolic Simulation of Real-Time Concurrent Systems. In: Chen, J., Hong, S. (eds.) RTCSA 2003. LNCS, vol. 2968, Springer, Heidelberg (2004)
32. Wang, F., Huang, G.-D., Yu, F.: Numerical Coverage Estimation for Dense-Time Systems. In: König, H., Heiner, M., Wolisz, A. (eds.) FORTE 2003. LNCS, vol. 2767, Springer, Heidelberg (2003)
33. Wong-Toi, H.: Symbolic Approximations for Verifying Real-Time Systems. Ph.D. dissertation, Stanford Univ., Stanford, CA (1995)
34. Yovine, S.: Kronos: A Verification Tool for Real-Time Systems. International Journal of Software Tools for Technology Transfer 1(1/2) (October 1997)

Applying Model-Based Testing to HTML Rendering Engines – A Case Study*

Jens R. Calamé[1] and Jaco van de Pol[2]

[1] Centrum voor Wiskunde en Informatica,
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
jens.calame@cwi.nl
[2] University of Twente, Faculty EEMCS
P.O. Box 217, 7500 AE Enschede, The Netherlands
vdpol@cs.utwente.nl

Abstract. Conformance testing is a widely used approach to validate a system correct w.r.t. its specification. This approach is mainly used for behavior-oriented systems. BAiT (Behavior Adaptation in Testing) is a conformance testing approach for data-intensive reactive systems. In this paper, we validate the applicability of BAiT to systems, which are not behavior-oriented (reactive) but document-centered.

In particular, we apply BAiT to the test of the HTML rendering engine Gecko, which is used by Mozilla Firefox. In order to do so, we formally specify a part of the CSS box model in the specification language μCRL and implement a wrapper for the Gecko renderer. Then, we automatically generate test cases and run tests with BAiT in a controlled experiment in order to demonstrate our approach on the relevant part of Gecko.

1 Introduction

Testing as a dynamic approach to software quality assurance is widely accepted in industry and is a well-studied field in academia. State-of-the-art testing approaches are model-based [4], i.e. tests are not generated in an ad-hoc manner, but founded on the specification of the software product under consideration. One of the most rigorous model-based test approaches is conformance testing, which tests whether an implementation conforms its specification focusing on functional requirements. These notions are made precise in the formal methods community [12].

In this paper, we want to apply conformance testing to rendering engines of web browsers. In state-of-the-art webdesign, content and design are kept separate from each other. Content is defined in the Hypertext Markup Language (HTML), while the design is specified in a Cascading Style Sheet (CSS). When a web page is rendered, the information from the CSS is used to position elements of content on the rendered page. If a web document has a complex structure,

* Part of this research has been funded by the Dutch BSIK/BRICKS project.

rendering algorithms can turn out to be erroneous, leading to "broken" web pages with mispositioned elements. Rendering a modern web application, whose appearance is dynamically changed on the client side using script languages, like web applications based on *Asynchronous JavaScript and XML* (AJAX, [16]), is even more demanding for rendering engines. Performing a sufficient conformance test in this context is tedious, so that an automated solution is preferable. In this paper, we present a feasibility study for automated testing of rendering engines using the test tool BAiT.

Here, we validate the applicability of BAiT [5,6], a blackbox test execution tool for non-deterministic, data-oriented reactive systems, to test the rendering engine of a web browser w.r.t. the positioning of boxes in the CSS box model. Boxes in HTML are entities like for instance a complete HTML document (`body` element) or a paragraph (elements `p` or `div`), which contain content or other boxes and which are positioned either absolutely or relative to each other. The box model is part of the W3C CSS Specification [15, Sect. 8].

We present a feasibility study, applying conformance testing using BAiT to the rendering engine *Gecko*[1], which is used by the open source web browser *Mozilla Firefox*[2]. In order to perform the tests, we formalize the CSS specification and design test purposes. Furthermore, we implement a wrapper component between the tester and *Gecko* in order to achieve a mapping between an action-oriented specification and the document-oriented rendering engine. For several setups of web pages, we then automatically generate parameterizable test cases. Those test cases can be instantiated with varying data settings for the positions of boxes, so that they are reusable for different page layouts. Then, the test cases are executed automatically against the test wrapper and the results retrieved from *Gecko* are interpreted in order to automatically assign verdicts.

1.1 Related Work

There exists a number of static test suites for the rendering capabilities of web browsers. Each of those test suites consists of a set of HTML and CSS documents with different page layouts. The most well-known one is probably the *ACID 2 Test*[3]. It tests web browsers for their full compliance to the actual version of CSS by rendering a web page with a vast amount of CSS features enabled.

Another set of test suites for the standard compliance of web browsers are the *W3C Cascading Style Sheets Test Suites*[4]. Here, again, we have a set of static documents, which test rendering capabilities for distinct features of CSS. Finally, Mozilla Firefox itself provides a set of static layout regression tests[5], which can be run in debug builds of the software.

Most of the named test suites are, however, not automated. In fact, the files in the test suites, i.e. the test cases, have to be loaded into the browser and then

[1] `http://www.mozilla.org/newlayout/`
[2] `http://www.mozilla.com/en-US/firefox`
[3] `http://acid2.acidtests.org/`
[4] `http://www.w3.org/Style/CSS/Test/`
[5] `http://www.mozilla.org/newlayout/doc/regression_tests.html`

the result of rendering the page has to be visually assessed. This process is not automatic at all, neither on the level of test case generation, nor on that of test execution. This means, that a certain amount of test cases has to be designed and executed manually and the results have to be visually evaluated. This process is time-consuming compared to an automated test process, where test cases – or at least test data – is generated for a number of standard and critical cases. In this case, the number of test cases to be generated and executed can be optimized in order to reduce the absolute number of test cases. The regression tests of Mozilla Firefox are at least automated on the level of test execution, however, they are still founded on a static set of test cases.

The approach, which we propose in this paper, provides not only an automated test execution and evaluation of rendering results for a fragment of the CSS features, the box model, but also an automatic generation and variation of tested web page layouts. We chose this fragment, because rendering results, i.e. the position of a box, can be objectively measured (in pixels) rather than having to be visually assessed. The approach of a fully automatic test case generation and execution has the advantage regarding the other named approaches, that the executed test cases can cover more variation w.r.t. data parameters (i.e. the position of boxes), but also regarding the structure of the rendered web pages. The first issue enables us to reuse equally structured test cases (i.e. web pages) and by that to reduce the number of generated test cases. The latter allows us to test rendering web pages with a different interrelation of elements and by that cover a larger variety of possible failures in the IUT.

We are aware of several case study reports of model-based testing, concerning topics like the Conference Protocol [2], the Storm Surge Barrier in the Netherlands [9], smart card applications [8], the telecommunication sector [3] or – vertical to our work – the generation of test purposes for the Session Initiation Protocol [1]. To the best of our knowledge, however, this is the first application of model-based automatic test generation and testing techniques to document-centered applications, esp. to HTML rendering engines.

2 The Test Environment

The test environment for our case study consists of two main components. On the one hand, we have the tester, which controls the run of the experiment. The tester will be discussed in Sect. 2.1. On the other hand, we have the implementation under test (IUT), which we will discuss in Sect. 2.3. This is the object under consideration, which we will actually be testing throughout the case study. In Sect. 2.2, we will give an introduction to the CSS box model.

2.1 Firefox and Cascading Style Sheets

Firefox. Mozilla Firefox is a stand-alone web browser, which has its roots in the Netscape Communicator from the 1990s. Most of its code was put under an open-source license in 1998 and founded the basis for the Mozilla Suite, from which Firefox arose as a stand-alone browser in 2004.

Web page content (HTML):

```
...
<div class="warning">
  ...
  <div class="warning"
       id="warning1">...</div>
  ...
</div>
...
```

Web page layout (CSS):

```
div.warning {
  border-style: solid;
  border-color: red;
  color:red;
}
div.warning#warning1 {
  font-style: italic;
}
```

Fig. 1. Two differently formatted boxes

A web browser reads and interprets HTML structured data to display web pages. The task of actually displaying is carried out by a rendering component. While loading a web page, this component incrementally builds up a Document Object Model (DOM) tree from the HTML document to be displayed together with declarative layout information. Mozilla Firefox uses the renderer *Mozilla Gecko*, whose version 1.8.1 we consider in this paper as the IUT.

Cascading Style Sheets (CSS). In the 1990s, a declarative stylesheet language was developed for structured documents in order to properly divide the content of a web page from its design. Currently, the de-facto standard for CSS is in version 2.1 [15]. This version is currently not fully supported by all web browsers, including Mozilla Firefox. This issue, however, does not affect our case study.

The CSS design definition for a web page can be provided in three different ways: as an external CSS file, which is linked to the HTML file of the web page, inline the HTML web page and inline a particular element of the web page. In the first two cases, a stylesheet is a collection of blocks of the following form:

```
element.class#id {property_1: value_1; ...; property_n: value_n; }
```

The literal `element` denotes one of the possible elements of HTML [14], like – for simple boxes – `div` or `span`. The literal `class` denotes a user-defined specialization of an HTML element, while `id` denotes a user-specific identifier for a particular occurrence of this element in an HTML document. For each of these elements, we can define pairs of properties and values.

Fig. 1 shows a small example: Boxes of class *warning* are rendered with a red border and red text. The one warning box with the identifier *warning1* additionally has the text in *italic*. Since this box is a warning box, too, it also takes over all properties from the warning boxes (red border and red text). As a result, the shown HTML code fragment is rendered as two red boxes, embedded into each other, of which the inner box has italic text.

If a CSS design definition is provided in a separate file, it is linked to the web page by using the `link` element of HTML in the following way:

```
<link rel="stylesheet" type="text/css" href="mycss.css" />
```

CSS information can also be provided inline the HTML document by nesting it in a `style` element. Inline CSS on HTML element level, which we will be using in our case study, omits the block structure from Fig. 1. The definition of the outer box from the figure using CSS inline the element itself looks as follows:

```
<div style="border-style:solid;border-color:red;color:red;"></div>
```

2.2 The CSS Box Model

In our case study, we will regard the positioning of `div`-boxes by the Gecko rendering engine. Therefore, we have to regard both the dimensions of a box as well as other parameters, which determine the box's position or its distance to other elements on a web page. The interrelation of boxes on a web page is defined by the CSS box model [15, Sect. 8], which we will briefly introduce here.

The dimensions of a `div`-box are essentially determined by two CSS properties: `width` and `height`. Furthermore, a minimum width and height can be defined as well as their maximum counterparts.

In addition to its width and height, a box also has a number of distances to contained or surrounding content. Those settings are displayed in Fig. 2. First of all, this is the distance to surrounding content (CSS properties `margin` or `left-/top-/right-/bottom-margin`, resp.). Furthermore, the distance of the box's border to any contained content can be defined (properties `padding` or `left-/top-/right-/bottom-padding`, resp.). Finally, the width of a box's border is defined by the properties `border` or `left-/top-/right-/bottom-border`, resp. We will later come back to these settings.

Boxes can be positioned in a variety of possibilities. The positioning mode is set in the CSS property `position`, which can have one of the four values `static`, `absolute`, `fixed` and `relative`. The default setting is `static`, which does not affect the standard element flow (top to bottom on the web page). Boxes can furthermore be positioned absolutely to either the HTML document under consideration (`absolute`) or the viewport, i.e. the browser window or a page in print (`fixed`). Finally, boxes can be positioned relative to each other, using the setting `relative`.

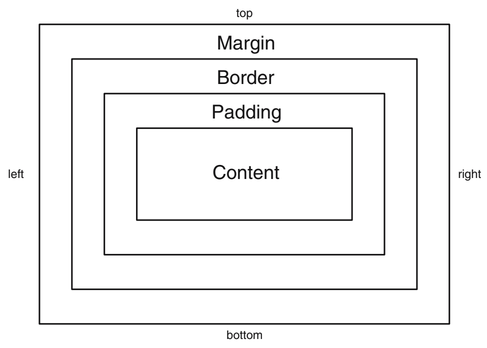


Fig. 2. CSS Box Model [15], box dimensions

An absolutely positioned box is provided with up to four additional parameters determining its position: a `left`, `right`, `top` and `bottom` offset. A box, which is positioned w.r.t. the upper left corner of the HTML document can, for instance, be determined by a `left` and a `top` offset in addition to its `width` and `height`. A box with the lower right corner as its fix point would accordingly be defined using the `bottom` and `right` offset parameters and leaving the other ones undefined. For boxes, which are overdefined, e.g. by defining a `left` and a `right` offset as well as a `width`, the W3C documents define the correct handling.

While the position of an absolutely positioned box is only determined by the given offsets, the position of a relatively positioned box must be computed regarding the other boxes on the same web page. One issue which determines the position of a relatively positioned `div`-box w.r.t. another one is its position in the DOM-tree of the HTML document. If a box A appears in the tree before another box B, then A is rendered above or left of B. If A appears after B, then it is rendered either right of B or below B. Furthermore, A can enclose B, if B is a child node of A in the DOM tree.

The absolute position of the box is then computed as the summation of the other boxes' measurements. Assume, the box under consideration is anchored to the upper left corner of the web page. Then, its top offset is the sum of all top and bottom margins, widths of the top and bottom borders and the heights of all boxes *above* the one under consideration. The box's left offset is computed as the sum of all left and right margins, widths of the left and right borders and the widths of all boxes *left* the one under consideration. The right and bottom offsets are left undefined. The padding of the one box, which surrounds all the mentioned boxes, is taken into account by assuming, that the top margin of the top-most box and the left margin of the left-most box is at least as wide as the padding of the surrounding box.

2.3 BAiT: Testing and Test Data Selection

Behavior-Adaptation in Testing (BAiT, [5,6,7]) is a toolset, that implements the test generation and execution process displayed in Fig. 3. The process starts from a specification of the IUT. In our case, such a specification is given in the formal specification language μCRL [10], which is based on a process algebra with abstract datatypes. In a first step, behavior and data are separately extracted from the specification. The behavior part forms the abstract specification, which is further used for the generation of parameterizable test cases. Data relations in the regarded system form the test oracle, which is later used to actually parameterize the generated test cases.

The test generation branch of the process regards behavior after having abstracted data using a so-named chaotic data abstraction [6,7], by which variable parameters are replaced by a distinct constant *chaos*. In doing so, we avoid the problem of state space explosion during the test generation process. The generation of parameterizable, abstract test cases is performed by the tool *Test Generation with Verification Techniques* (TGV, [11]), which performs a guided search over the whole state space of the abstract specification. The test engineer

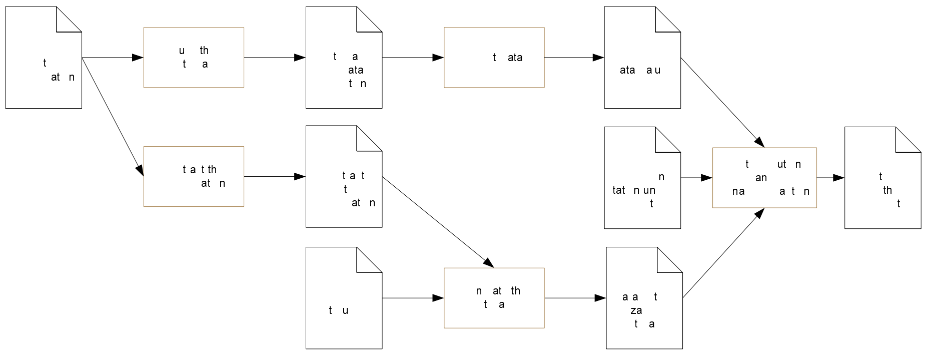


Fig. 3. Test Generation and Execution Process

can guide this process by providing test purposes, which sketch the focus of the test cases to be generated. The result of test case generation are parameterizable test cases, which can be instantiated with test data for test execution.

In parallel to test case generation, BAiT also prepares the later instantiation of the test cases with test data. Therefore, it generates a constraint logic program (CLP) from the system specification. This CLP holds all data interdependencies within the tested system and serves as a test oracle during test execution. Using the test oracle, data can be selected in order to instantiate parameters for the generated test cases.

Once both the parameterizable test cases and the test oracle have been generated, the test can be performed against an IUT. Depending, whether the IUT reacts as expected or not, the test ends with a verdict *Pass*, *Fail* or *Inconc*.

In our setting, TGV does not generate a single test case, i.e. a single trace through the system's behavior, but a so-named complete test graph (CTG). Such a CTG provides a set of traces through the system, whose final states can be marked as *Pass* or *Inconc* states. If during test execution a *Pass* state is reached in the CTG, then the test *passes*, i.e. no failure could be found in the system. If an *Inconc* state is reached, then the system behaved according to its specification, however, the test purpose was not met. This can happen, if the test purpose disallows a particular behavior which is allowed in the system's specification. In this case, it is not clear, whether a failure has been found in the test run, or not. Finally, a test can lead to the verdict *Fail*, if the IUT shows behavior, which is not allowed according to the specification.

For test execution, BAiT first selects a single abstract trace to a *Pass* verdict from the CTG. This trace is then instantiated and executed in a step-wise manner. If the IUT diverts from the precomputed trace by sending unexpected data to the tester, the trace under consideration is pruned. Then, BAiT checks, whether the IUT is still in an allowed system state according to the test oracle. If this is not the case, the test run ends with a verdict *Fail*. If it is the case, then BAiT tries to find an alternative trace to a *Pass* verdict. If this attempt does not succeed, the test ends with verdict *Inconc.* If a trace to *Pass* could be successfully executed to its end, then the test run ends with verdict *Pass*.

3 Objective of the Case Study

The objective of our case study is to apply BAiT to testing the implementation of the *Gecko* rendering engine. BAiT has originally been designed as a tool for the test of data-oriented reactive systems in general. In this paper, we report on a feasibility study to validate the applicability of BAiT to HTML rendering engines. These systems (or system components, resp.) are not reactive systems in the original sense.

Reactive systems are based on events being sent forth and back between several systems. These events can be parameterized with data. A rendering engine works differently: It is document-centered, i.e. it receives a document and renders it. While sending the document to the rendering engine is still comparable to the reactive systems described above, rendering this document is not. It is no reaction in the original sense, since no evaluable events are sent back from the IUT. The only event sent back from the renderer states, that rendering is finished, but it does not contain the actual result of rendering. In many cases, the result of rendering must even be evaluated visually, while for some aspects, the relevant information can be retrieved from the rendering engine and can be computer-processed.

Such an aspect are the positions of `div`-boxes to which we will restrict the test. Rules for positioning such boxes are defined in the CSS Box Model, which has been described in Sect. 2.2. We will, however, not consider the full box model, but restrict to a fragment of it.

First of all, we concentrate on the settings `absolute` and `relative` with a binding to the top left corner of the web page for the possible positioning of boxes. Secondly, we consider empty boxes of an explicitly defined width and height only in order to keep the results of rendering predictable by the test oracle. Boxes filled with content may lead to overflowing content which results in a correction by the rendering engine based on information about the viewport dimensions and the used font dimensions. Treating those details in this feasibility study would not be purposeful and furthermore would have required arithmetic division operations, which would complicate the specification in μCRL. For this reason, we do not regard (overflowing) content in this case study.

Thirdly, we limit the possible scales used in the design definition of a `div`-box. Normally, the position and size of a `div`-box is determined by distances, which can be defined in a variety of scales. Some of those some are absolute (pixels, didot points, pico points, inches, millimeters and centimeters) and other are relative to either the actual font setting (scales em and ex) or the rest of the page layout (percentages or the `auto` setting). In this paper, we only consider absolute lengths of scale `px` (pixel) in order to avoid scale conversions.

Finally, we use a "flat" model in our case study rather than one, which resembles the whole nested structure of a web page. This means, that we regard only a distinct box *testbox* and its absolute position on the web page. When we add another box to the web page, then we recompute the position of *testbox* as it has changed due to the newly added other box. This means for instance, that, if we add another box above the regarded *testbox*, the top offset of *testbox* is recomputed as the summation of the previous top offset, the top and bottom margins of the new box, the top and bottom border width of the new box and the new box's height. By doing so, we can easily keep track of the position of the regarded *testbox* without having to keep the whole HTML document structure in our model.

Apart from the applicability of BAiT to HTML rendering engines in particular, we also aim at two other targets with this case study. On the one hand, we want to test the adaptability of BAiT to nondeterministic behavior of the IUT further by introducing some artificial nondeterminism w.r.t. to the system's feedback about the rendered boxes. On the other hand, we want to regard the feasibility of μCRL as a language for the design of test purposes.

4 Realizing the Test Environment

In order to test Gecko, we first had to create a test environment. This environment, as schematically depicted in Fig. 4, consists of a tester and an implementation under test (IUT). The tester is in our case the tool BAiT. The IUT is a component named Gecko Wrapper, which wraps Gecko internally. Both the tester and the IUT are Java components which communicate with each other using bidirectional procedure-based communication.

In order to generate and run the tests following the schema from Fig. 3, we also need a system specification of the CSS box model for Gecko and a test purpose to sketch out the later test cases. While the design of the tests and hence also that of the test purposes will be the topic of Sect. 5, we will in the remainder of this section discuss the specification of the boxmodel. Furthermore, we will give some details on the Gecko Wrapper.

4.1 Modelling CSS in μCRL

We modeled a fragment of the CSS box model with the limitations from Sect. 3 in μCRL [10]. The modeled fragment of CSS allows to position boxes relative to

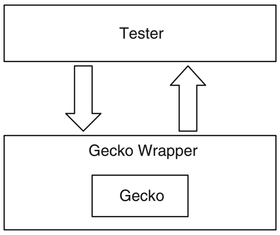


Fig. 4. Test Environment

Table 1. Actions for the CSS box model

Action	**Functionality**
Input actions:	
`resetBoxes`	Wipes all boxes and starts again with a fresh document.
`setupTestbox`	Defines the distinctly regarded test box.
`putBoxRelative`	Puts a box relatively to the other boxes yet defined in the actual HTML document. It can be defined, whether this box appears left of, right of, above, beneath or around all yet defined boxes. Finally, the measurements can be defined.
`putBoxAbsolute`	Puts a box with an absolute position.
`render`	Renders the actually defined document and starts returning results (offsets, see below)
Output actions:	
`offsetLeft`	Returns absolute left offset of test box.
`offsetRight`	Returns absolute right offset of test box.
`offsetTop`	Returns absolute top offset of test box.
`offsetBottom`	Returns absolute bottom offset of test box.

each other or absolutely. In our model, recursive structures of boxes are flattened by regarding only one distinct box and its position, rather than a structure of boxes. Whenever a box is added, only the consequences on the position of the regarded box are computed and applied.

While rendering a web page is in principle a document-centered task, our specification of the box model is behavior-oriented. Hence, we defined a set of input actions, which allow us to put boxes into a box structure. Furthermore, we defined some output actions, which provide information about the current offset of the regarded box to the tester. The actions are defined in Table 1.

The system behavior for the CSS box model is specified as follows: As a first step, a testbox must be set up (`setupTestbox`). The action `setupTestbox` accepts parameters, which determine the box's width and height. Other boxes can be put in the vicinity of this testbox using the actions `putBoxRelative` and `putBoxAbsolute` in any order. The action `putBoxRelative` accepts 15 parameters: The first one determines, whether the box appears above, below, left, right or around the other boxes, which have yet been inserted into the web page. This parameter is named "position", but is actually not related the the `position`-property of CSS. The next two parameters determine the box's width and height. The last 12 parameters, finally, define the width of the box's padding, border and margin as depicted in Fig. 2. The action `putBoxAbsolute` only accepts seven parameters. The first three are identical with those from `putBoxRelative`, while the remaining four parameters define the box's absolute position on the page w.r.t. the four margins of the web page.

Any of these three actions can be followed by an action `resetBoxes` in order to delete all boxes and start from scratch, or by an action `render`. In this case, the IUT renders the defined structure of `div`-boxes. Afterwards, the different actual values for the offsets of the testbox (left, right, top, bottom offset) are returned by the IUT in an arbitrary order.

As we described in Sect. 3, we only regard a distinct box *testbox* in the model, whose position we recompute each time another box is added to the HTML test document. The actions `putBoxAbsolute` and `putBoxRelative` change this position in the described way. Below, we give the μCRL code, which relates to the behavior of `putBoxRelative`:

```
...
PrepareRendering(position:PositionType,relation:RelationType,
   width:Nat,height:Nat,offsetLeft:Nat,offsetRight:Nat,
   offsetTop:Nat,offsetBottom:Nat) =
...
 + sum(pposition:PositionType,sum(pwidth:Nat,sum(pheight:Nat,...,
   putBoxRelative(pposition,pwidth,pheight,pmarginleft,
       pmarginright,pmargintop,pmarginbottom,pborderleft,...).
   PositionBoxRelative(position,relation,width,height,...)...)))
...
PositionBoxRelative(position:PositionType,relation:RelationType,
   width:Nat,height:Nat,...) =
   tau.PrepareRendering(...,offsetLeft+pmarginLeft+
       pborderLeft+ppaddingLeft+pwidth+ppaddingRight+
       pborderRight+pmarginRight,...)
    <|and(eq(relation,relative),eq(pposition,left))|>delta
 + tau.PrepareRendering(...,offsetLeft,0,offsetTop,offsetBottom)
     <|and(eq(relation,relative),eq(pposition,right))|>delta
 + tau.PrepareRendering(...,offsetTop+pmarginTop+pborderTop+
        ppaddingTop+pheight+ppaddingBottom+pborderBottom+
        pmarginBottom,offsetBottom)
     <|and(eq(relation,relative),eq(pposition,top))|>delta
 + tau.PrepareRendering(...,offsetLeft,offsetRight,offsetTop,0)
     <|and(eq(relation,relative),eq(pposition,bottom))|>delta
...
```

The above fragment of our specification shows the definition of the action `putBoxRelative` within a process `PrepareRendering`. When `putBoxRelative` has been invoked, the system enters another process, `PositionBoxRelative`. After a τ-step, the system leaves `PositionBoxRelative` and goes back to `Prepare-Rendering`. While doing so, the new position of the test box is computed depending on the value of the variable `pposition` of the newly positioned box. This leads to a case distinction depending on the position of the new box.

4.2 Wrapping Mozilla Gecko

Mozilla Gecko can be embedded into custom applications as a component, which can be programmed using its XPCOM interfaces. The Cross Platform Component Object Model (XPCOM, [13]) is an unmanaged component framework, which is used for the Mozilla software products. Gecko can be embedded into Java applications. In order to do so, one can either instantiate it directly via its XPCOM interfaces or embed it indirectly via the `Browser` component of the Standard Widget Toolkit (SWT)[6].

We implemented a wrapper for Gecko in Java using SWT. The wrapper receives all actions which place boxes and builds up an internal structure for a test web page. On action `render`, the wrapper generates actual HTML and CSS code and sends this code to the renderer. A window is opened for rendering.

When rendering is finished, the renderer is queried for the offsets. For this procedure, we followed an existing code example[7]. In order to query an offset, a short piece of JavaScript code is generated and executed within the web browsing component. This piece of code internally queries for the respective offsets and writes the result to an (invisible) status bar. When this has happened, the wrapper can read the value from this status bar in order to store it, and the next piece of JavaScript is generated and executed (one execution per one of the four offset parameters). After all offsets have been queried, the tester is informed by the actions `offsetLeft`, `offsetRight`, `offsetTop` and `offsetBottom` in a random order. We chose for a random order in order to test the adaptation of BAiT to nondeterministic behavior of the IUT, as we have described in Sect. 3.

5 Running the Tests

5.1 Design of the Test Cases

In the BAiT approach, test generation is based on the tool TGV [11]. This tool takes as input the system specification in the form of an LTS as well as a test purpose. In a first step, prior to test case generation, we applied data abstraction on the specification of the system, in order to avoid space explosion induced by the many unrestricted numerical parameters of the input actions `setupTestbox`, `putBoxAbsolute` and `putBoxRelative`.

The second step was to design test purposes. We designed two test purposes, of which one traditionally directly as an LTS, while the second one was specified in μCRL as was the system itself. According to the first test purpose, we set up a testbox, put at least one more (relatively positioned) box in its vicinity and render the resulting HTML document. Having done so, we expect the system to return at least the top offset of the testbox. The second test purpose is designed a bit differently, since we still wanted to experiment more with BAiT's capability of behavior-adaptation during a test run. For this purpose, the test purpose was designed to expect at least the left offset of the testbox and to refuse an action `offsetBottom` following directly on the `render` action. This refusal in combination with the absolutely random order of `offset`-events from the IUT leads to more situations in which BAiT will be led into a trace to an *Inconc*

[6] `http://www.eclipse.org/swt`

[7] `http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/org.eclipse.swt.snippets/src/org/eclipse/swt/snippets/Snippet160.java`

Test purpose in μCRL:

```
% datatype definitions and action definitions skipped
proc
  PASS = ACCEPT.PASS
  FAIL = REFUSE.FAIL

  PutBox = render + putBoxRelative.PutBox
  TP = setupTestbox.putBoxRelative.PutBox.
        (offsetLeft.PASS + offsetBottom.FAIL)

init TP
```

Resulting test purpose as LTS before adding placeholders for action parameters:

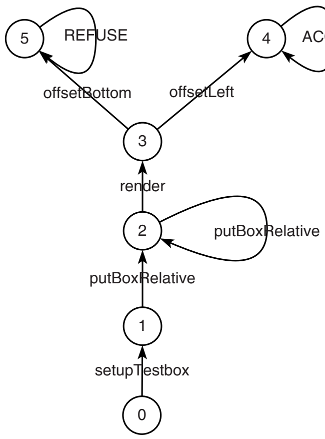


Fig. 5. A test purpose both in μCRL and as an LTS

verdict, from which it will try to find an alternative trace to a *Pass* verdict. BAiT will, however, never find such a trace and it will have to give up terminating with verdict *Inconc*. Since the generated test cases can contain loops, BAiT might search for a trace to *Pass* without ever terminating. This issue has been solved by introducing a configuration option for BAiT, which defines the maximum amount of traces to search for before giving up and assigning *Inconc*. This second test purpose is shown in Fig. 5 as a μCRL specification and an LTS.

In a third step, we generated complete test graphs (CTGs) with TGV. The abstracted system specification as input to TGV was quite manageable with its 17 states and 57 transitions, so that the generation process took place within negligible time. For the first test purpose, generation resulted in a CTG with 25 states and 70 transitions. The second test purpose put more restrictions on the behavior of the IUT during the test, so the number of transitions in the resulting CTG was reduced to 59; the number of states increased slightly to

28. In general, these numbers are relatively low, a circumstance which does not astonish if one keeps in mind, that we regard only the behavior of a highly data-intensive system after data abstraction. The main work, as we had already remarked earlier, is the data selection during test execution.

5.2 Test Execution

Based on the generated CTGs, we ran some tests with BAiT and the Gecko wrapper. We used the default trace search algorithm of BAiT in order to select test traces through the CTGs. This algorithm searches only for traces to *Pass*, using a breadth-first search. Having automatically selected a trace to *Pass*, we then selected data for the different parameters of the box positioning actions and executed the trace. During the different test runs, we found a few failures. However, those failures were induced by faults in the used model rather than by the IUT itself. After having eliminated the faults, we did not find any more failures in the IUT.

As expected, the test runs based on the first CTG always ended in a *Pass* verdict, after we had corrected the model. The test runs based on the second CTG, randomly went to a *Pass* or an *Inconc* verdict. This behavior was dependent on whether the wrapper returned an `offsetBottom` event before (*Inconc*) or after (*Pass*) the `offsetLeft` event (cf. the description of the second test purpose). Since the order of events was implemented in the wrapper to be random, the assignment of verdicts was also as expected a priori.

6 Conclusion

In this paper, we presented the application of an existing approach for conformance testing, Behavior-Adaptation in Testing (BAiT), to the test of the HTML rendering engine Mozilla Gecko. This is a new application for BAiT, since the toolset was originally designed to treat nondeterministic reactive, and thus action-based, systems with data. We modeled a fragment of the CSS box model in the formal specification language μCRL, implemented a wrapper for Mozilla Gecko in order to be able to apply the BAiT for test execution, and designed some test cases following the BAiT approach. In a controlled experiment, we validated the applicability of BAiT to document-centered HTML rendering.

We designed the case study as a feasibility study in order to do a first evaluation for further experiments with BAiT and Gecko. Focusing on a subset of aspects and leaving out things like handling of boxes with overflowing content, decreased the design effort of the model of our fragment of the CSS box model and the implementation effort w.r.t. the wrapper for Gecko significantly.

Our experiences with the design of test purposes were twofold. On the one hand the behavior-oriented part of the design was very easy. On the other hand, this means that most of the test design is induced by data and thus actually happens during the test run itself and can be computed automatically. We did not encounter any problems at this point. The main issue was the fact, that the data parameters for the configuration of the `div`-boxes were mostly independent of each other. This did not constitute a problem by itself, however, the capabilities of BAiT guiding test data selection by precomputing the ranges of data parameters could not be beneficial in this case. Running the second test case, we encountered, that BAiT's behavior adaptation can easily lead to test runs, which do not terminate. In this case, a careful configuration of the search threshold of BAiT was necessary to prevent infinite test runs while at the same time avoiding superfluous *Inconc* verdicts. In real life testing of Gecko, this would not be an issue, since the superfluous *Inconc* verdicts are induced by the nondeterminism, which we artificially built into the IUT.

The feasibility study described in this paper was successful and forms an important step towards a fully automated model-based test approach for HTML rendering engines using BAiT. Compared to static test suites, which serve the document-centered HTML rendering process, our behavior-oriented approach has the advantage of a higher flexibility w.r.t. data parameters and the tested document structure. With a wrapper for Gecko and a behavior-oriented approach to test the IUT, it is far easier to generate a variety of different HTML test documents, which cover different aspects of the IUT. Special expertise is only necessary in order to provide BAiT with a formal specification of the CSS box model. Test data of a sufficient quality can – also for critical values causing overflowing boxes – can, for instance, easily be selected by the developers of the rendering engine themselves.

From the reached state, we can now extend the work in some respect. In order to do a full model-based test of Gecko (or other rendering engines) w.r.t. the CSS box model, it is necessary to extend the formal specification to the whole model. Since the model in [15] is given in natural language rather than a formal notation, it might quite likely be incomplete, ambiguous and may contain semantic variation points. Such aspects would complicate an attempt to formalize the model. Another, technical, issue concerning the conformance test with the full box model is the treatment of floating point datatypes rather than integers. Not only, that the specification language used in this case study does not directly support floating point datatypes, also some aspects of test data selection must be considered. At the moment, we consider output from the IUT being exactly equal to the expected output in order to pass the test. In a floating point setting, we would have to consider the output to be not necessarily exactly equal to the expected output, but to be within a particular margin around our expectations. A last issue to consider w.r.t. a full formal model of the CSS box model is, that this extended model may not be flattened anymore like the one we used in this case study. It rather has to reflect the nested structure of boxes on the rendered web page in its system status. This issue also affects the test data selection approach.

References

1. Aichernig, B.K., Peischl, B., Weiglhofer, M., Wotawa, F.: Test Purpose Generation in an Industrial Application. In: Proc. of the 3rd Intl. Workshop on Advances in Model-based Testing, pp. 115–125. ACM, New York (2007)
2. Belinfante, A., Feenstra, J., de Vries, R., Tretmans, J., Goga, N., Feijs, L., Mauw, S., Heerink, L.: Formal Test Automation: A Simple Experiment. In: Csopaki, G., Dibuz, S., Tarnay, K. (eds.) 12th Intl. Workshop on Testing of Communicating Systems, pp. 179–196. Kluwer Academic Publishers, Dordrecht (1999)
3. Born, M., Schieferdecker, I., Gross, H.-G., Santos, P.: Model-driven Development and Testing – A Case Study. In: van Sinderen, M.J., Pires, L.F. (eds.), 1st European Workshop on Model Driven Architecture with Emphasis on Industrial Application, number TR-CTIT-04-12 in CTIT Technical Report, Enschede, pp. 97–104 (2004)
4. Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.): Model-Based Testing of Reactive Systems. LNCS, vol. 3472. Springer, Heidelberg (2005)
5. Calamé, J.R.: Adaptive Test Case Execution in Practice. Technical Report SEN-R0703, Centrum voor Wiskunde en Informatica (June 2007)
6. Calamé, J.R., Ioustinova, N., van de Pol, J.C.: Automatic Model-Based Generation of Parameterized Test Cases Using Data Abstraction. In: Romijn, J., Smith, G., van de Pol, J. (eds.) Proc. of the Doctoral Symposium affiliated with the 5th Intl. Conf. on Integrated Formal Methods (IFM 2005). Electronic Notes in Computer Science, vol. 191, pp. 25–48. Elsevier, Amsterdam (2007)
7. Calamé, J.R., Ioustinova, N., van de Pol, J.C., Sidorova, N.: Data Abstraction and Constraint Solving for Conformance Testing. In: Proc. of the 12th Asia-Pacific Software Engineering Conf. (APSEC 2005), pp. 541–548. IEEE Press, Los Alamitos (2005)
8. Clarke, D., Jéron, T., Rusu, V., Zinovieva, E.: Automated Test and Oracle Generation for Smart-Card Applications. In: Attali, S., Jensen, T. (eds.) E-smart 2001. LNCS, vol. 2140, pp. 58–70. Springer, Heidelberg (2001)
9. Geurts, W., Wijbrans, K., Tretmans, J.: Testing and Formal Methods – Bos Project Case Study. In: EuroSTAR 1998. 6th European Intl. Conf. on Software Testing, Analysis & Review, pp. 215–229 (1998)
10. Groote, J.F., Ponse, A.: The Syntax and Semantics of μCRL. In: Ponse, A., Verhoef, C., van Vlijmen, S. (eds.) Algebra of Communicating Processes, Workshops in Computing, pp. 26–62. Springer, Berlin (1994)
11. Jard, C., Jéron, T.: TGV: Theory, Principles and Algorithms. Intl. Journ. on Software Tools for Technology Transfer 7(4), 297–315 (2005)
12. Tretmans, J.: Test Generation with Inputs, Outputs, and Repetitive Quiescence. Software - Concepts & Tools 17(3), 103–120 (1996)
13. Turner, D., Oeschger, I.: Creating XPCOM Components (2003)
14. W3C. XHTML 1.0 The Extensible HyperText Markup Language (2nd edn.) (August 2002) W3C Recommendation, `http://www.w3.org/TR/2002/REC-xhtml1-20020801`
15. W3C. Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification (July 2007). W3C Candidate Recommendation, `http://www.w3.org/TR/2007/CR-CSS21-20070719`
16. Zakas, N.C., McPeak, J., Fawcett, J.: Professional AJAX. Wrox Press (2006)

Model-Based Generation of Testbeds for Web Services

Antonia Bertolino[1], Guglielmo De Angelis[1], Lars Frantzen[1,2], and Andrea Polini[1,3]

[1] Istituto di Scienza e Tecnologie della Informazione "Alessandro Faedo"
Consiglio Nazionale delle Ricerche, Pisa, Italy
{antonia.bertolino,guglielmo.deangelis}@isti.cnr.it
[2] Institute for Computing and Information Sciences (ICIS)
Radboud University Nijmegen, The Netherlands
lf@cs.ru.nl
[3] Department of Mathematics and Computer Science
University of Camerino, Italy
andrea.polini@unicam.it

Abstract. A Web Service is commonly not an independent software entity, but plays a role in some business process. Hence, it depends on the services provided by external Web Services, to provide its own service. While developing and testing a Web Service, such external services are not always available, or their usage comes along with unwanted side effects like, e.g., utilization fees or database modifications. We present a model-based approach to generate stubs for Web Services which respect both an extra-functional contract expressed via a Service Level Agreement (SLA), and a functional contract modeled via a state machine. These stubs allow a developer to set up a testbed over the target platform, in which the extra-functional and functional behavior of a Web Service under development can be tested before its publication.

1 Introduction

The emergence of the Service Oriented Architecture (SOA) paradigm is changing the way in which software applications are developed [17]. Two trends, seemingly in contradiction with each other, are witnessed. On the developer's side, we face a drop in control capability: a service-oriented application consists of the dynamic composition of autonomous services. Therefore, a service-oriented application commonly depends on the services provided by external, autonomous services, and its development process is not anymore under the full control of a single stakeholder/organization.

On the consumer's side, instead, users (i.e., the service clients), who can choose among many available services, are becoming more and more exigent concerning the properties expected from a service, and ask for precise specifications of the offered service functionality and performance. Based on those they will make their "purchase" decision. Such specifications establish a *contract* between the service provider and the client.

Thus, a service provider needs to manage and reconcile these contrasting trends: binding client requirements against limited control capability. A key approach to solve this conflict stays in propagating the formalization of contracts among all the services invoked by a composite service. The interdependency with other services is laid on precise specifications of service interfaces, which also establish contracts on the interactions between services.

A contract can deal with functional and extra-functional properties. When dealing with the former, beside the mere signatures of the provided operations, the offered functionality of services is often subject to given conditions, like e.g. *always invoke the authentication operation providing a valid password before invoking the booking operation.* This is especially true for stateful services. Such contracts are prevalently specified via state machines. Extra-functional contracts, i.e., the delivered Quality of Service (QoS) levels, are made explicit in Service Level Agreements (SLAs in the following) [21]. SLAs also provide the basis on which the cost of using the service is quantified, and the penalties, in case the contract is violated, are defined.

When developing a new service, its interaction with the external services it uses must be tested, to validate that it obeys the functional contracts in place, and to evaluate its offered quality level, which is affected by the quality levels of the invoked services. In an ideal case, all the external services are available, and can be arbitrarily accessed at development time for testing purposes. Unfortunately, this ideal case is seldom offered. Commonly, at least some of the external services are either not available at all (for instance simply not implemented, yet), or their usage comes along with unwanted side-effects (for instance utilization fees or database modifications). But what is usually available are the contracts of the external service interfaces. The availability of models specifying extra-functional behavior for the interacting services could also suggest the application of analytical techniques [16] to derive the needed extra-functional properties. As discussed in [8], when interaction happens through complex middlewares, such an option is not always feasible, since the modeling of such infrastructures is particularly difficult and error prone.

At the core of the approach presented in this paper is a model-based stub generator, called Puppet [23]. We assume that for each external service not available or suited for testing, both a functional contract in terms of a state machine, and an extra-functional contract in terms of an SLA, are present. Based on these two contracts, Puppet automatically generates a stub for the external service, which respects the contracts. For the functionality this means that the stub behaves conforming to the functional contract (for instance, it never violates any guard of the corresponding state machine). Furthermore, when someone is using the stub, it is able to detect a violation of its functional contract by that user. Respecting the extra-functional contract means that the stub meets the quality levels specified in the corresponding SLA.

Using this generator the developer of a new composite service can replace all external services which are not fully available at development time with the stubs generated by Puppet. The automatically obtained stubs make up a testbed in which the service under development can be tested within the target platform before it is published. Carrying on the testing also entails the identification of a suitable test suite. This is a challenging issue, but outside the scope of the present paper (though we hint at possible directions to pursue in future work).

In a preliminary work [4] we described how stubs respecting an extra-functional contract can be generated, but in that work the generated stubs did not consider functional aspects (i.e., they provided a correct quality-related behavior, but the messages were not built to be semantically meaningful). This paper's original contribution is the integration of functional contracts as part of the generated testbed. We will motivate (see Sect. 5) that functionality and extra-functionality are not independent from each other. Having functionally correct stubs can reveal extra-functional behavior of the service under development, which may not be observable in a purely extra-functional testbed. Vice versa, having stubs respecting given quality levels may disclose functional behavior which does not show up in a purely functional testbed. Thus, here the whole is more than the sum of its parts; combining both kinds of contracts strictly increases the testing power of the testbed compared to taking both a purely extra-functional and a purely functional testbed. While related proposals exist for functional verification or extra-functional evaluation of services (see Sect. 6), to the best of our knowledge, no such framework supports an integration of both aspects.

The next section provides an overview of Puppet, and introduces the case study we will use throughout the paper to illustrate the approach. Section 3 explains how the SLAs are modeled and simulated, while Sect. 4 provides the background for functional modeling, and introduces the formal testing relation we exploit to simulate correct functional behavior of the generated stubs. Section 5 illustrates the testing power of the testbed generated by Puppet. Finally, related work is overviewed in Sect. 6, while conclusions and further interesting features, that Puppet could offer in future work, are given in Sect. 7.

2 Overview

We demonstrate our approach by referring to an exemplary case study, which is introduced in Sect. 2.1. Section 2.2 presents the logical approach of the Puppet stub generator.

2.1 Motivating Scenario

We have implemented a simplified version of the scenario presented in [1], in which three services (the customer, the supplier, and the warehouse) cooperate to achieve the task of a trade. The customer service is interested in buying a certain amount of a given product, and queries the supplier service for a quote for the product of interest. Having received the request, the supplier queries one or more warehouse services to check if the requested quantity is in stock. The information provided by the warehouses is then collected by the supplier service, and returned to the customer service. If satisfied with one of the provided quotes, the customer can then proceed with the order. The case study also handles further interactions,

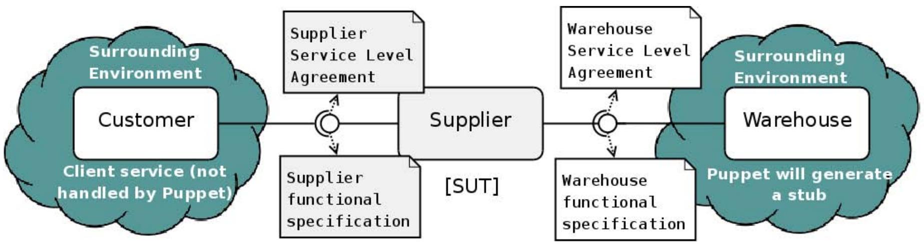

Surrounding Environment
Customer
Client service (not handled by Puppet)
Supplier Service Level Agreement
Supplier functional specification
Supplier
[SUT]
Warehouse Service Level Agreement
Warehouse functional specification
Surrounding Environment
Warehouse
Puppet will generate a stub

Fig. 1. The Customer-Supplier-Warehouse Case Study

namely supplier authentication and bonus accounting schemes, which will be discussed in Sect. 4 and Sect. 5.

We can realistically assume that the three services are implemented and provided by different stakeholders, and that their interactions are governed by message exchange protocols under agreed levels of QoS, as shown in Fig. 1.

For illustration purposes, we put ourselves in the role of the *developer of a supplier service*, which requires a testbed to a) test the compliance of the supplier service with the functional contracts of the warehouse services, and b) needs to derive reliable values for the QoS of the supplier service by taking into account the QoS of the given warehouse services. The latter is particularly important if the developer him/herself needs to publish a contract for potential customers, which may involve the provision of the services under agreed level of QoS parameters (such as for instance throughput, latency, reliability).

The behavior of the supplier (both functional and extra-functional) depends on the behavior of the warehouse services going to be accessed at run-time. The problem the developer faces here is that he/she does not want to invoke the real warehouse services during development time for testing purposes (e.g., really buying goods).

To address this issue, the PUPPET tool can automatically generate stubs for the warehouses that do not only reproduce the specified functional behavior encoded in corresponding state machines, but also perform according to the QoS parameters defined in the warehouses SLAs. The idea is, that such a stub can then be deployed on a Web Service platform (such as Axis [3]), potentially reproducing different distribution settings, and being accessed during testing by the supplier service for taking experimental measures. For simplicity the considered case study only deals with one warehouse service to be simulated; in general PUPPET can handle an arbitrary number of services, generating one stub for each externally accessed service.

2.2 Logical Approach of PUPPET

PUPPET generates stubs which exhibit meaningful extra-functional and functional behaviour. The generation is structured in three main steps. The first step defines the stub structure by converting the abstract part of a WSDL [6] representation into a collection of Java classes and interfaces. This transformation is performed exploiting the Apache-Axis *WSDL2Java* utility [3]. The second generation step extends the stub with the code simulating the extra-functional behavior. We assume that the desired QoS properties of the service to be stubbed are expressed according to a WS-Agreement specification [13]. Section 3 will show how the WS-Agreement statements are mapped to parametric portions of Java code. The final step of the generation process inserts into the stub parametric code able to emulate the intended functionality. Our assumption is that a functional specification of the service to be stubbed is available (possibly derived from a global view such as a choreography specification) via a state machine. Such a specification defines which are the correct values expected for the service invocation parameters, what is the legal message ordering accepted by the service, and, what are the characteristics of an answer to be provided by the service. In particular we adopt the *Symbolic Transition System* (STS) notation (see Sect. 4). Having all desired stubs generated by Puppet, the developer is able to mock-up the environment by deploying the generated stubs on any Axis platform.

Note that the generated functional and the extra-functional parts may interoperate at run-time. As described in the following, Puppet uses pre-defined Java patterns in order to emulate the functional and extra-functional properties. Puppet combines the generated patterns according to a fixed order. Defining a new pattern or changing an existing one has to be validated with respect to both the possible dependencies with the other patterns, and the order in which Puppet combines them. For example, as detailed in Sect. 3.2, the definition of the Java pattern dealing with latency constraints takes into account the temporal dependency on other computational tasks.

3 Model-Based Extra-Functionality

This section firstly introduces the language we use to express SLAs (Sect. 3.1), and then shows how the referred extra-functional properties are processed by Puppet (Sect. 3.2). Generally speaking, SLAs describe the agreements that a service commits to accomplish when processing a request from a client, starting from the moment it receives the request until the moment it replies [22]. QoS properties are defined only as a provider constraint, and do not include any kinds of events that the client may experience, for example network failures or traffic congestion problems. We point at ways to consider network issues in Sect. 6.

3.1 WS-Agreement

WS-Agreement [13] is a language defined by the Global Grid Forum aiming at providing a standard layer to build agreement-driven SOAs. The main ingredients of the language concern the specification of domain-independent elements of a simple contracting process. Such generic definitions can be augmented with domain-specific concepts. The top-level structure of a WS-Agreement is expressed via an XML document comprising the agreement descriptive information, the context it refers to and the definition of the agreement items. It includes the involved parties as well as other aspects such as its expiration date.

An agreement can be defined for one or more contexts. The defined consensus, or obligations, of a party core in a WS-Agreement specification are expressed by means of terms, organized in two logical parts. The `Service Description Terms` part specifies the involved services. It describes the reference to a description of a service, rather than describing it explicitly into the agreement. The second part of the terms definition specifies measurable guarantees associated with the other terms in the agreement. Such guarantees can be fulfilled or violated. A `Guarantee Term` definition consists of the obliged parties (i.e, *Service Consumer* and *Service Provider*), the list of services this guarantee applies to (`Service Scope`), a boolean expression that defines the condition under which the guarantee applies (`Qualifying Condition`), the actual assertions that have to be guaranteed over the service (`Service Level Objective` - SLO), and a set of business-related values (`Business Value List`) of the described agreement (i.e., importance, penalties, preferences). In general, the information contained in the fields of a `Guarantee Term` is expressed by means of domain-specific languages.

3.2 Defining Extra-Functional Annotations

The approach implemented in PUPPET associates concepts in the WS-Agreement (i.e., `SLO`, `Qualifying Condition`, `Service Scope`) with an interpretation by means of a given operational semantics. This can be a quite complex and effort-prone task, but given a specific language and an intended interpretation of the concepts, it has to be done only once and for all.

Precisely, PUPPET defines a mapping from the declarative XML descriptions of the supported QoS properties to composable Java code segments. The mapping is specified in a parametric format that is instantiated each time one occurrence of the concept appears. Within the scope of this paper, we deal with two QoS properties: latency and reliability. The remainder of this section introduces their characteristics. Please note that the specifications of such QoS properties conform to the definitions adopted within the PLASTIC Project [9]. Nevertheless, also other definitions can be adopted (e.g. as in [20]).

```
...
<wsag:ServiceLevelObjective>
 <puppetSLO:PuppetSLO>
  <puppetSLO:Latency>
   <value>25000</value>
    <puppetSLO:Distribution>
      <Gaussian>10</Gaussian>
    </puppetSLO:Distribution>
  </puppetSLO:Latency>
 </puppetSLO:PuppetSLO>
</wsag:ServiceLevelObjective>
...
```

–A–

```
...
Density D = new Density();
long funcElapsedTime = puppet.ambition.Naturals.asNatural
      (aMbItIoNinvocationTime - System.currentTimeMillis()
      );
long maxSleepingPeriod = 25000 - funcElapsedTime;
Double sleepingPeriod = D.gaussian(maxSleepingPeriod,10);
try {
  Thread.sleep(sleepValue.longValue());
} catch (InterruptedException e) {}
...
```

–B–

Fig. 2. SLO Mapping for Latency

Latency is defined as a server-side constraint, and does not concern (just ignores) other kinds of delays that the client may experience, for example due to network failures or traffic congestion problems. Conditions on latency are simulated in PUPPET by introducing *delay* instructions into the operation bodies of the services stubs. For each `Guarantee Term` in a WS-Agreement document, information concerning the maximum service latency is defined as a `Service Level Objective`. As an example, Fig. 2.A reports the XML code for a maximum latency declaration of $25000msec$ normally distributed, and Fig. 2.B shows the corresponding Java code that is automatically generated by PUPPET.

When dealing with latency constraints, PUPPET also has to deal with other computational tasks, like generating a functionally correct return message, taking care of reliability constraints, etc. Since these tasks also consume time, PUPPET has to adapt the generated latency sleeping period. For example, consider that the term in Fig. 2.A comes in combination with some functional computation statements. If at run time these computations take $2sec$, the delay of the service is adjusted to the range of $[0 \div 23000]msec$. In case the calculation of the functionally correct return message takes more than what is allowed by the latency constraint, the stub raises an exception and has failed its purpose. Since SLA latency constraints for services are commonly in the order of seconds, the computational tasks needed to generate the return messages only miss such deadlines in quite rare cases.

Reliability constraints are declared in the `Service Level Objective` of a `Guarantee Term`, stating the maximal admissible number of failures a service can raise in a given time window. Such kinds of QoS attributes can be reproduced introducing code that simulates a service failure. PUPPET realizes a reliability

```
...
<wsag:ServiceLevelObjective>
 <puppetSLO:PuppetSLO>
  <puppetSLO:Reliability>
   <Reliabilitywindow>
    120000
   </Reliabilitywindow>
   <MaxFailures>
    3
   </MaxFailures>
   <puppetSLO:Distribution>
    <Gaussian>
     10
    </Gaussian>
   </puppetSLO:Distribution>
  </puppetSLO:Reliability>
 </puppetSLO:PuppetSLO>
</wsag:ServiceLevelObjective>
...
```

–A–

```
...
long winSize = 120000;
int maxFault = 3;
long currentTimeStamp = System.currentTimeMillis();
for (int i=0; i<faultBuffer.size();i++){
  if (currentTimeStamp - faultBuffer.get(i) >= winSize){
    faultBuffer.remove(i);
  }
}
if (faultBuffer.size() < maxFault){
  Density d = new Density();
  double dv = d.gaussian(100);
  if (dv > 50) {
    String fCode = "Server.NoService";
    String fString = "PUPPET␣EXCEPTION␣:␣No␣target␣
          service␣to␣invoke!";
    org.apache.axis.AxisFault fault = new org.apache.
          axis.AxisFault(fCode, fString, "", null);
    aMbItIoNsim.undo();
    faultBuffer.add(currentTimeStamp);
    throw fault;
  }
}
...
```

–B–

Fig. 3. SLO Mapping for Reliability

failure via an exception raised by the platform hosting the Web Service stub. An example of the Puppet transformation for reliability constraints is shown in Fig. 3. Part A shows the XML code specifying a maximum allowed number of three failures over an observation window of 2 minutes; part B gives the corresponding Java translation, assuming that the Apache-Tomcat/Axis [3] platform is used.

As described in Sect. 3.1, a guarantee in a WS-Agreement document could also be stated under an optional condition expressed by means of some `Qualifying Condition` elements. Usually such optional constraints are defined in terms of accomplishments that the service consumer must meet. For instance, the latency of a service may depend on the value of some parameters provided at run-time. In these cases, the Puppet transformation function wraps the simulating code obtained from the `Service Level Objective` part within a conditional statement. As mentioned, the scope for a guarantee term describes the list of services to which it applies. In these cases, for each listed service, the transformation function adds the behavior obtained from the `Service Level Objective` and `Qualifying Condition` transformations only to those operations declared in the scope.

4 Model-Based Functionality

The functional behavior of a service is modeled using an automata model called *Symbolic Transition System* (STS). STSs are a well studied formalism in modeling and testing of reactive systems [12]. We understand, though, that they could sound unfamiliar and difficult for practitioners. However, STSs can be seen as a formal semantics for a variant of UML 2.0 state machines [18]. We have developed a library called Minerva [23], which transforms the output generated by MagicDraw (`http://www.magicdraw.com`) – a commercial UML modeling tool – into an STS representation understood by Puppet. Thus, a developer can use this visual tool to model the functionality of service interfaces in the common formalism of UML 2.0 state machines. We do not describe this mapping here, but present instead directly the STS formalism. We will use a dedicated testing relation called *eco* [11] which is specifically appropriate for the creation of stubs like the ones we are dealing with in our setting. This section introduces STSs and the *eco* relation.

4.1 Symbolic Transition Systems

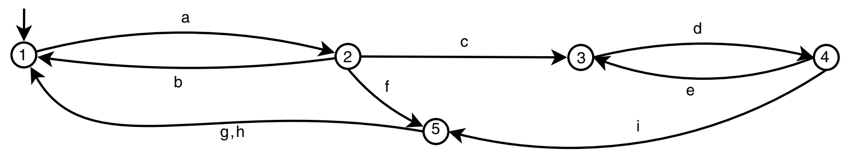
In our setting, STSs specify the functional aspects of a service interface. Firstly, there are the static constituents like types, messages, parameters, and operations. This information is commonly denoted in the WSDL [6]. Secondly, there are the dynamic constituents like states, and transitions (also called arcs) between the states. STSs can be seen as a dynamic extension of a WSDL. They specify the legal ordering of the message flow at the service interface, together with constraints on the data exchanged via message parameters (called *parts* in the WSDL).

An STS can store information in STS-specific variables. Every STS transition corresponds to either a message sent to the service (input), or a message sent from the service (output). Furthermore, a transition can be guarded by a logical expression. After a transition has fired, the values of the variables can be updated. Due to its extent and generality we do not give here the formal definition of STSs, which can be found in [12]. Instead, we exemplify the concepts in the setting relevant for this paper.

We assume here that data types in the WSDL are specified via XML Schema types, as commonly done. Let us consider a WSDL operation `checkAvail` with an input message `?checkAvail` and an output message `!checkAvail`. The input message has a part `r` of type `QuoteRequest`; the output message has a part `q` of type `Quote`. The `QuoteRequest` type is a complex type sequence with the elements `product` of type `String` and `quantity` of type `Integer`. The `Quote` type is a complex type sequence with the elements `status` of type `Integer`, `product` of type `String`, `quantity` of type `Integer`, `price` of type `double`, and `refNumber` of type `Integer`. This WSDL operation could for instance correspond to a Java method with signature: `Quote checkAvail(QuoteRequest r)`, together with the classes `Quote` and `QuoteRequest`. A message in an STS corresponds to a message in the WSDL. Hence, we model the call of the `checkAvail` operation in the STS by two succeeding transitions. The first one with message `?checkAvail(r:QuoteRequest)` represents the operation invocation, the second one represents the returned value via the `!checkAvail(q:Quote)` message.

Regarding the case study, which we introduced in Sect. 2.1, the `checkAvail` operation is one of the four operations offered in the WSDL specification of the warehouse service. The remaining three operations are `auth`, `cancelTransact`, and `orderShipment`. The `auth` operation has an input message `?auth(pw:String)` and an output message `!auth(q:Quote)`. Just an input message exists for the `cancelTransact` and `orderShipment` operations, no value is returned via an output message. The input message of the `cancelTransact` operation is called `?cancelTransact`, and has a part `ref` of type `Integer`. The input message of the `orderShipment` operation is called `?orderShipment`, and has a part `ref` of type `Integer`, and a part `adr` of type `Address`. The `Address` type is a complex type sequence with elements necessary for identifying an address (the concrete elements are not relevant, here).

Figure 4 shows an STS specifying the warehouse service. Initially, the warehouse is in state `1`. Now a user of the service (in our case study the Service Under Test (SUT), i.e. the supplier) can invoke the `checkAvail` operation by sending the `?checkAvail` message. This corresponds to the transition *a* from state `1` to state `2`. The guard of the transition restricts the attribute `quantity` of parameter `r` to be greater than zero. After the transition has fired, `r` is saved in the variable `qr` (which is also of type `QuoteRequest`). Next, the warehouse has to return a `Quote` object via the return parameter `q`. Three things can happen. Firstly, the requested product may not be on stock with the requested quantity. In this case a `Quote` object is returned with the status attribute being `SOLDOUT` (transition *b*). Secondly, if the product is on stock and the requested quantity is less than or



ARC	MESSAGE	GUARD	UPDATE
a	`?checkAvail<r:QuoteRequest>`	`r.quantity > 0`	`qr = r;`
b	`!checkAvail<q:Quote>`	`q.status == SOLDOUT`	
c	`!checkAvail<q:Quote>`	`qr.quantity > MAXQ && q.status == AUTHREQ`	
d	`?auth<pw:String>`		`spw = pw;`
e	`!auth<q:Quote>`	`q.status == PWINVAL && !valid(spw)`	
f	`!checkAvail<q:Quote>`	`qr.quantity <= MAXQ && q.price > 0.0 && q.status == VALIDQUOTE && q.refNumber > 0 && q.product == qr.product && q.quantity == qr.quantity`	`qi = q;`
g	`?cancelTransact<ref:Integer>`	`ref == qi.refNumber`	
h	`?orderShipment<ref:Integer, adr:Address>`	`ref == qi.refNumber`	
i	`!auth<q:Quote>`	`valid(pw) && q.status == VALIDQUOTE && q.product == qr.product && q.price > 0.0 && q.quantity == qr.quantity && q.refNumber > 0`	`qi = q;`

Fig. 4. The Provided Interface of the Warehouse as an STS

equal some limit `MAXQ`, a `Quote` object is returned with status `VALIDQUOTE`, the same `quantity` as being requested, and a `price` and `refNumber` greater than zero (transition f). We save here the issued quote in the variable `qi`. Thirdly, if the requested quantity exceeds `MAXQ`, a quote is returned with status `AUTHREQ` (transition c). This informs the user to provide a password string via the the `auth` operation (transition d). If the password in invalid, a quote with status `PWINVAL` is returned (transition e), and the user has to invoke the `auth` operation again. Given a valid password, a valid quote is returned (transition i).

Being in state `5`, again two things can happen. Either the user of the service decides to reject the quote. He/she invokes the one-way operation `cancelTransact` by sending the message `?cancelTransact` (transition g). Here he/she must refer to the correct issued reference number `refNumber`. Or he/she decides to accept the quote. In this case, in addition to the correct reference number, an address must be provided as a second parameter to the `?orderShipment` message (transition h).

4.2 Environmental Conformance

We show now how the above specification can be used to generate functionally correct responses within a generated stub. In model-based testing, a testing relation formally defines when a model representing the SUT conforms to a model constituting its specification [24]. A testing relation for the formalism of Labeled Transition Systems (LTS) is *eco* [11]. Since STSs have an underlying LTS semantics, we can use *eco* also for STSs.

The motivation of *eco* is to define what it means for a component C to correctly invoke a requested, or environmental, component E. Given a provided interface specification of E in terms of an LTS, an *eco* compliance checker for E

plays the role of E. While doing so it checks if C respects the specification of E when invoking it.

The main observation here is, that an *eco* compliance checker for the component E does exactly what we demand from a functionally correct stub for E. Taking the warehouse STS specification given in Fig. 4, a generated *eco* compliance checker (from hereon simply called *stub*) will play the role of the warehouse, and in doing so it can test if the supplier, while using the warehouse, does respect the STS specification. To render more precisely what that means, we show an example walk by the stub trough the STS, next. Initially, the stub is in state `1`. The only allowed call here is `checkAvail` with a quantity greater zero. If it receives a different call, it will alert a detected failure of the user. If the call is correct, it moves to state `2`. Now the stub can decide nondeterministically if it either returns a quote with status `SOLDOUT` (back to state `1`), or if it checks the quantity and proceeds to state `3` or `5`. This choice is made randomly, but may also be made according to certain coverage criteria, or other heuristics. Assuming, for instance, that the stub decides to check the quantity and that the quantity is greater `MAXQ`. It then constructs a quote with status `AUTHREQ`, returns it to the user, and moves to state `3`. Now it waits for the user in invoke the `auth` operation. Assuming that the user does provide a valid password here, the stub moves to state `4` and sees that the password is valid. Next it constructs a `Quote` object with status `VALIDQUOTE`. Also here the stub has many choices, every solution to the guard on transition i corresponds to a possible quote. The stub will choose one solution (again randomly, or according to some heuristics), return the corresponding quote to the user, and move to state `5`. Now it waits again for the user to either cancel the transaction, or order the shipment. And so on. Using this approach ensures that the generated stubs give always functionally correct answers, and can detect incorrect invocations from interacting components.

5 Verifying and Utilizing the Testbed

We have shown so far the approach to generate stubs for Web Services which respect both a given SLA contract, and a functional contract in terms of a state machine. This section firstly presents in Sect. 5.1 results of experiments which we carried out to verify the stubs themselves, i.e., we tested here if the stubs really behave as their protocols dictate, to gain confidence in the implementation environment. Having the stubs deployed, the developer can test the service under development in this testbed. Sections 5.2 and 5.3 exemplify and summarize the testing power of the generated testbed. We refer again to the customer, supplier, warehouse scenario introduced in Sect. 2.

5.1 Verifying the Generated Stubs

We have to verify that a stub which is generated by Puppet both respects its SLA, and its STS specification. To verify the former we specified several SLAs for the warehouse service, and checked if the behavior of the generated stubs is in line with what their SLA dictates. In order to verify the latency exhibited by the stubs, we instrumented the service container of the warehouses with a simple performance monitor logger, as described in [3]. To obtain meaningful measures, we iterated each interaction scenario over 1000 executions. We always verified that the mean response time elapsed was in line with the one dictated by the SLA. To monitor reliability constraints, we associated the agreement in Fig. 3 with the operation `orderShipment` of the warehouse, and developed a supplier which repeats any invocation of `orderShipment` when a remote exception was thrown by the warehouse. Furthermore, we instrumented the supplier with logging code, counting how many invocations it executed until the shipment of the order succeeded. As in the case of latency, we monitored the reliability constraints over 1000 executions, and verified that they were always respected.

To verify the functional conformance we generated several warehouse stubs with Puppet based on (variations of) the state machine specification from Fig. 4. To verify that they functionally conform to their specification, we automatically tested them with the model-based Web-Service testing tool Jambition [23], which is especially suited for this purpose since it also takes STSs as its specification formalism. No failures were observed after several hours of automatic testing. Secondly, we developed a supplier service which behaves conforming to the warehouse specification from Fig. 4, and made several non-conforming mutants of it, like *requesting a quote with a zero quantity*, or *ordering goods without having received a valid quote beforehand*. All mutants were immediately detected by the warehouse stub.

5.2 Detecting Extra-Functional Failures

This case refers to the task of the developer to derive reliable values for the quality levels of the newly developed service by taking into account the QoS of the external services. Having merely extra-functional correct stubs gives a testbed as the one we described in [4]. The added value for the approach of this paper comes into play when taking into account also the functionality. The main advantage of a functional stub here is, that it has a notion of state. In other words, it forces the user to respect the functional protocol. By so doing, the user automatically walks through the specified transactions, like *ordering products after having authenticated*. For instance, the enforcement of its functional protocol by the warehouse stub may reveal failures in the extra-functional behavior of the supplier. Going in detail, let us assume that the supplier has to meet a given SLA on latency regarding the interactions with its clients, namely processing each request within $40000msec$. As defined in the agreement with the warehouse shown in Fig. 2, each interaction between the warehouse and the supplier service can take up to $25000msec$. Take also into account, as described in Sect. 4, that the warehouse service requires an additional authentication step in case the product quantity exceeds `MAXQ` (see Fig. 4).

A potential extra-functional failure here is, that when the authentication of the supplier is required, the time needed by the supplier to fulfill a client request may violate its SLA. Even if the first password provided is correct, the response of the warehouses to the availability request (arc c), together with the response to the provided password (arc i), may in the worst case sum up to $50000 msec$, which respects for each invocation the SLA exposed by the warehouse, but breaks the supposed SLA between the supplier and the client. Given a warehouse stub which does not have any notion of the functional protocol might never notice the necessity of authentication for a supplier. Each request is considered stand-alone, and no relation to previous or following requests, including data interdependencies, exist. Thus, in a mere extra-functional testbed this extra functional failure can easily be invisible.

5.3 Detecting Functional Failures

We have already shown above (end of Sect. 5.1), that the generated stubs are able to detect functional violations of their contracts by the user. This refers to the task of testing the compliance of a service under development with the functional contracts of external services, and is on its own already of high value for the developer.

Taking into account also the extra-functional correctness of the stubs, can reveal further functional issues of their users. For instance, the warehouse stub generated by PUPPET shows an extra-functional behavior which can potentially affect the functional state of the supplier (its user). To exemplify this, assume a supplier offering a special *welcome discount* to new clients for their first five purchases. Furthermore, let us consider that the supplier behaves as depicted in Fig. 5. For a given client, the supplier associates a counter `FreeOrder`, initially being five, which is decremented each time the client places an order. To fulfill the order request, the supplier invokes the `orderShipment` operation of the warehouse stub. In case a reliability failure occurs now, this is propagated to the client. Let us recall that the interactions between the supplier and the warehouse service is governed by an SLA containing a reliability clause as specified in Fig. 3. The supplier service is not prepared to deal functionally properly with such a reliability failure in the sense that it does not increase again the

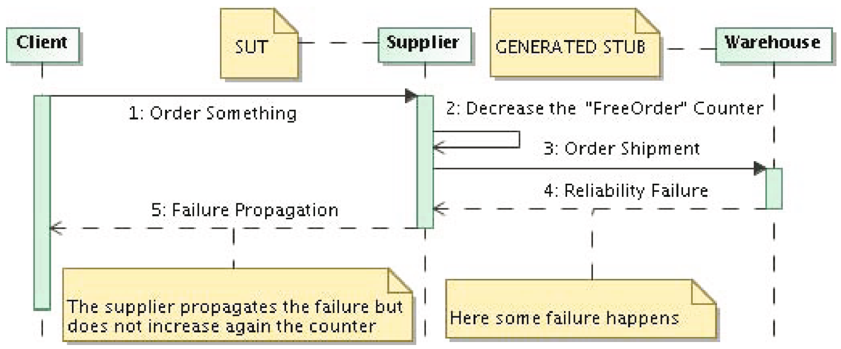

Client
SUT
Supplier
GENERATED STUB
Warehouse
1: Order Something
2: Decrease the "FreeOrder" Counter
3: Order Shipment
4: Reliability Failure
5: Failure Propagation
The supplier propagates the failure but does not increase again the counter
Here some failure happens

Fig. 5. Functional Fault Revealed by a Reliability Constraint

`FreeOrder` counter to its original value. This is necessary since the warehouse does not process the order due to its reliability failure - the products cannot be purchased by the client of the supplier. As a consequence, each reliability failure reduces the number of discounts by one, even though no goods have been purchased by the client. Such kinds of functional failures cannot be discovered using a testbed that only reproduces functionally correct behavior, ignoring the extra-functional specifications.

6 Related Work

An early interesting work exploring QoS testing is Grabowski and Walker [14], although they did not consider QoS relations among input and output, but only temporal relations among two streams, and did not target the automatic generation of testbeds. However, they provide an interesting classification of QoS requirements. In [10], Weyuker and Vokolos highlight the potential of using testing and synthetic workload to assess the performance behavior of complex distributed applications. Denaro and coauthors [8] provide a framework for early evaluation of performance characteristics of a distributed application, taking into account the influence of complex middleware.

Concerning our goal, i.e., testbed generation for deriving or validating the contracts of composite services, without accessing the invoked real services, to the best of our knowledge no similar approaches exist that can simulate both the functional and extra-functional behavior of the surrounding environment. Ramsokul and coauthors [19] discuss a testing strategy and provide a conformance relation to assess the cooperation among services against a globally specified protocol (they do not address SLA constraints). Closer to our approach are some QoS testbeds. Zhu and coauthors [26] develop an interesting model-driven approach to the generation of benchmarks reproducing clients' workload for evaluating the QoS provided by a Web Service. Grundy and coauthors [15] propose a performance test-bed generator which has much in common with what we propose here, in that a collection of service stubs is generated from a compositional model, and clients simulating a defined workload are also automatically developed. However, in the cited works, no contract or agreement specification is assumed, and the surrounding environment is not simulated. We believe that the above works could be used to complement PUPPET, providing the workload to solicit our generated stubs.

Another interesting issue is how the network affects the QoS. As said, the provider of a service is responsible of the agreed SLAs at his/her port. Possible network problems are not his/her business, so the customer should handle such issues with the network provider [22]. Nevertheless the QoS perceived by the user of a service is also affected by the network. Therefore, to reproduce likely distribution settings and network delay distributions, the tester should use a network in which it is possible to control the introduced delay. In this respect an interesting tool is *Weevil* [25], that supports the creation of a synthetic workload for distributed software and can reproduce realistic stub distribution over world-wide distributed platforms.

7 Conclusions and Future Work

It is well known that testing amounts for the most part of its effort to coding, and great part of this coding effort is needed for making test cases executable. Automation of the setup of the testing environment is routinely addressed in conventional software development. Such need also affects, or is even exacerbated, in the testing of service-oriented systems, as the provision of a service commonly depends on other surrounding services.

We have presented the Puppet environment for the automatic generation of stubs simulating the behavior of external services invoked by a service under test, which encapsulates both their functional specifications and their contractual SLAs. To achieve this, we assume that we have access to an extra-functional specification (SLA), and a functional specification (STS), of the surrounding services. Using these specifications, the Puppet tool automatically generates for each surrounding service a stub which respects the functional and extra-functional properties. These stubs constitute an environment which realistically simulates the runtime environment. The service developer can test the SUT within it, and obtain realistic QoS measures, without having the need to access the real surrounding services, that might not be available, or could generate unwanted side effects. We have illustrated why considering both functional and extra-functional aspects for the stub generation is a crucial means to raise potential failures and obtain realistic estimates for the QoS attributes.

The integration of the extra-functional and functional contracts is currently done in pragmatic way, aiming at a proof-of-concept of the presented approach. Further investigations are necessary to come up with a theoretical foundation of such an integration. One promising candidate here is the combination of functional models like STSs with concepts known from timed automata [2,5].

Having in place the Puppet testbed, a suitable test suite must be generated. This task is out of the scope of Puppet, but the framework already gives indications on how this could be achieved. Obtaining reliable values demands for the execution of many tests, thus test generation here is a strong candidate for automation. Model-based testing (MBT) has precisely this goal, i.e., the automatic generation, execution, and evaluation of test cases based on a (usually functional) formal model of the SUT. A natural choice would be to keep the model already used by Puppet, namely STSs, also for functionally modeling the provided service of the SUT. We could use some existing STS-based MBT tools for doing so (like [7,23]). Future research has to explore how to precisely combine functional MBT of the SUT with an effective derivation of the corresponding QoS properties. Also here the promising candidate is the integration with timed automata models.

Finally the experiments we conducted so far have been aimed at verifying that the generated stubs are correct with respect to the (functional and extra-functional) specifications. As a next step, we are currently involved in further experiments directed at validating Puppet in practice. Specifically, in collaboration with the industrial partners of the PLASTIC project [9], we are using the stubs generated by Puppet on wider and more complex case studies.

Acknowledgements

This research is carried on within the European Project PLASTIC (IST STREP 026955), and supported by the Marie Curie RTN TAROT (MRTN-CT-2004-505121) and the Netherlands Organisation for Scientific Research (NWO) under project STRESS.

References

1. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services–Concepts, Architectures and Applications. Springer, Heidelberg (2004)
2. Alur, R.: Timed Automata. In: Computer Aided Verification, pp. 8–22 (1999)
3. Jeelani Basha, S., Irani, R.: AXIS: the next generation of Java SOAP. Wrox Press (2002)
4. Bertolino, A., De Angelis, G., Polini, A.: A QoS Test-bed Generator for Web Services. In: Baresi, L., Fraternali, P., Houben, G.-J. (eds.) ICWE 2007. LNCS, vol. 4607, Springer, Heidelberg (2007)
5. Briones, L.B., Brinksma, E.: A Test Generation Framework for quiescent Real-Time Systems. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, Springer, Heidelberg (2005)
6. Christensen, E., et al.: Web Service Definition Language (WSDL) ver. 1.1 (2001), `http://www.w3.org/TR/wsdl/`
7. Clarke, D., Jéron, T., Rusu, V., Zinovieva, E.: STG: a Symbolic Test Generation tool. In: Katoen, J.-P., Stevens, P. (eds.) ETAPS 2002 and TACAS 2002. LNCS, vol. 2280, Springer, Heidelberg (2002)
8. Denaro, G., Polini, A., Emmerich, W.: Early Performance Testing of Distributed Software Applications. In: Proc. of WOSP 2004, pp. 94–103. ACM Press, New York (2004)
9. PLASTIC european project homepage, `http://www.ist-plastic.org`
10. Weyuker, E., Vokolos, F.: Experience with Performance Testing of Software Systems: Issues, and Approach, and Case Study. IEEE Transaction on Software Engineering 26(12), 1147–1156 (2000)
11. Frantzen, L., Tretmans, J.: Model-Based Testing of Environmental Conformance of Components. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Proc. of FMCO 2006. LNCS, vol. 4709, pp. 1–25. Springer, Heidelberg (2007)
12. Frantzen, L., Tretmans, J., Willemse, T.A.C.: A Symbolic Framework for Model-Based Testing. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) FATES 2006 and RV 2006. LNCS, vol. 4262, pp. 40–54. Springer, Heidelberg (2006)
13. Global Grid Forum. Web Services Agreement Specification (WS–Agreement), version 2005/09 edition (September 2005)
14. Grabowski, J., Walker, T.: Testing Quality-of-Service Aspects in Multimedia Applications. In: Proc. of PROMS 1995 (1995)

15. Grundy, J., Hosking, J., Li, L., Liu, N.: Performance Engineering of Service Compositions. In: Proc. of IW-SOSE 2006, pp. 26–32. ACM Press, New York (2006)
16. Haverkort, B.R., Niemegeers, I.G.: Performability modelling tools and techniques. Performance Evaluation 25(1), 17–40 (1996)
17. Huhns, M.N., Singh, M.P.: Service-Oriented Computing: Key Concepts and Principles. IEEE Internet Computing 9(1), 75–81 (2005)
18. Object Management Group. UML 2.0 Superstructure Specification, ptc/03-08-02 edition. Adopted Specification
19. Ramsokul, P., Sowmya, A., Ramesh, S.: A Test Bed for Web Services Protocols. In: Proc. of ICIW 2007, pp. 16–21 (2007)
20. Sahner, R.A., Trivedi, K.S., Puliafito, A.: Performance and Reliability Analysis of Computer Systems An Example-Based Approach Using the SHARPE Software Package. Kluwer Academic Publishers, Dordrecht (1995)
21. Skene, J., Lamanna, D.D., Emmerich, W.: Precise Service Level Agreements. In: Proc. of ICSE 2004, pp. 179–188. IEEE Computer Society Press, Los Alamitos (2004)
22. Skene, J., Skene, A., Crampton, J., Emmerich, W.: The Monitorability of Service-Level Agreements for Application-Service Provision. In: Proc. of WOSP 2007, pp. 3–14. ACM Press, New York (2007)
23. PLASTIC tools homepage, `http://plastic.isti.cnr.it/wiki/doku.php/tools`
24. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software—Concepts and Tools 17(3), 103–120 (1996)
25. Wang, Y., Rutherford, M.J., Carzaniga, A., Wolf, A.L.: Automating Experimentation on Distributed Testbeds. In: ASE 2005, pp. 164–173. ACM, New York (2005)
26. Zhu, L., Gorton, I., Liu, Y., Bui, N.B.: Model Driven Benchmark Generation for Web Services. In: Proc. IW-SOSE 2006, pp. 33–39. ACM Press, New York (2006)

Real-Time Testing with TTCN-3

Diana Alina Serbanescu[1], Victoria Molovata[2], George Din[3],
Ina Schieferdecker[3,4], and Ilja Radusch[1]

[1] DCAITI, Berlin, Germany
[2] Politechnica, Bucharest, Romania
[3] Fraunhofer FOKUS, Berlin, Germany
[4] TU Berlin, Berlin, Germany

Abstract. Reactive real-time software is used when safety is the issue and the margin for errors is narrow. Such software is used in automotive, avionics, air traffic control, space shuttle control, medical equipment, nuclear power stations, etc. As the timeliness of those systems is critical, it needs to be assured and tested. However, real-time properties require automated tests as manual tests are untimely and imprecise. This paper reviews Testing and Test Control Notation Version 3 (TTCN-3) as a means for real-time testing and proposes extensions to enable real-time test systems in TTCN-3. Small examples demonstrate the usage of the new constructs. Real-time operating systems are analyzed and reviewed, to enable the realization of real-time test systems based on TTCN-3.

1 Introduction

As the software industry undergoes a high and rapid development process, the software products become increasingly diverse and complex. Therefore, the demands regarding performance and reliability enlarges. An important part in the industry is the field of embedded real-time systems, which operate in environments with strict timing constraints. Embedded real-time systems find their applicability in a variety of domains where consideration of these timing requirements is critical important or even crucial (for example in automotive, avionics and robotic controllers). Dealing with an increased level of complexity of the software products, the possibility of errors occurring during the development process is an unavoidable fact. Software testing can be costly, but avoiding the testing may generate more expensive risks, especially in places where human lives are at stake. Together with the emergence of real-time embedded technologies, the demand for developing suitable means for testing those systems has increased, so as to systematically achieve the desired level of reliability. Compared to classical systems, embedded real-time applications require even more powerful testing techniques because they not only have to provide a certain functionality, but they have to provide that functionality in a predefined, well-determined amount of time, sometimes this amount of time being very short. Therefore, the test system (TS) should be well instrumented for measuring the timing attributes of the system under test (SUT) and for providing test stimuli in appropriate time, as required by the test procedures.

This paper discusses, in Section 2, the properties a test system should have in order to test real-time embedded applications and provides further, a solution for designing such test systems. This solution is based on the TTCN-3. This language for designing and specifying tests is an internationally standardized testing language, constantly developed and maintained by the European Telecommunications Standards Institute (ETSI). It gained popularity during the last years for its successful applications, especially in the telecommunication domain. Although TTCN-3 is an expressive and flexible language for writing tests, it does not fully provide notions for describing real-time aspects. Therefore, we took the liberty to enhance the language with new concepts necessary for testing real-time features. Such extensions to the language and their usage are presented in Section 3 of this study.

Our target is a *real-time test system* that tests real-time applications not only remotely, by way of some form of dedicated communication links, but also within the proximity of the tested system itself, as it is embedded on a board physically attached to the system. This allows to deploy test systems even where interconnectivity is hard to achieve, as for example, on online tests of electronic control units (ECUs) in automotive and avionics. For achieving that, the test system should be able to define abstract real-time test constraints for the tested system and for itself. After its abstract definition, the test system should be low level implemented on a specific embedded platform, which consists of both the real-time operating system and real-time hardware. As embedded systems dispose of minimal hardware resources, the selection of the right real-time operating system (RTOS), constitutes an important step. The criteria used for selecting an appropriate real-time operating system is presented in Section 4.

For the execution of the tests, one has to fill in the gap between the high level test specifications and the real-time executable code. In Section 5, a practical example is presented, consisting of a simple real-time application for automotive and a demonstrative test designed for the example. The test is implemented both using the enhanced TTCN-3 notation and the application programmer interface (API) provided by the operating system. It is interesting to present the two perspectives, one abstract and the other very specific, in order to emphasize the correlations between them. While the TTCN-3 test specification is more concise and simple to use, the system implementation is rather complex. Hence, the aim is to have this code generated, so that a future goal is to develop a compiler for automated transformation of tests into executable code, customized for the specific platform, as discussed in Section 6.

2 Real-Time Testing

Software testing is the process of executing a program or system with the intent of assuring its quality [1] and of finding errors [2]. Testing involves any activity aimed at evaluating attributes or capabilities of programs or systems, and finally, determining if they meet all of the requirements.

Both white-box testing and black-box testing approaches are used. White-box testing denotes testing with the knowledge of the internal structure of the SUT. In contrast, black-box testing, which is also called functional testing, is purely based on the requirements of the SUT. The internal structure of the SUT is considered to be unknown. Black-box testing is mainly used in the higher levels of system design, that is, for integration, system or acceptance testing. Different to white-box testing, test cases for black-box testing are often written by using special test specification and test implementation languages. TTCN-3 is such a test language and its primary domain is black-box testing. There exist several types of black-box testing which focus on different kinds of requirements or on different test goals, such as conformance testing, interoperability testing, performance testing, system testing, acceptance testing and so forth. Conformance testing is functional black-box testing where the functionality of the SUT is tested. This paper considers mainly conformance testing of real-time embedded systems.

A simple conformance test system is that where the SUT is seen as a black-box, where only the inputs and outputs are visible. The outputs of the SUT are reactions of certain stimuli induced to it by the TS. These outputs are captured by the TS and matched against some predefined templates and if they coincide, the requirements are considered to be satisfactory, and the test is passed. Otherwise, the test fails.

In addition, real-time systems have to respect some special requirements, for which the matching of outputs is insufficient and the timings at which those outputs were received, are also relevant. This means that functionality must be accomplished within a certain time interval, its starting or ending should be marked by precisely defined points timely, but allowing some tolerance. An example of a test logic with real-time requirements for the TS and SUT is given in Figure 1. Regarding time, there are two critical sections in this test example: first, there is *t_max_1*, a time constraint for the SUT that indicates the interval in which the reaction to the first stimuli should be received by the TS; the second is *t_max_2*, which indicates the time that should elapse at the TS side, between the receiving of the first reactions from the SUT and the sending of the second stimuli to the SUT.

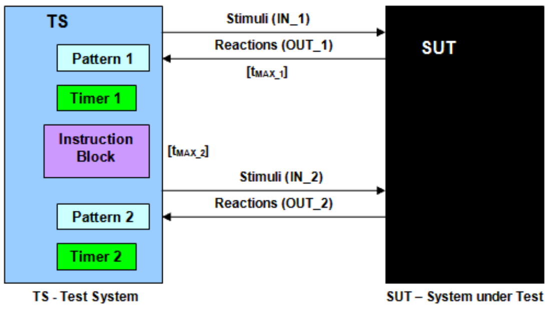


Fig. 1. Reactive real-time TS

3 Real-Time Language Extensions for TTCN-3

Following the presentation in the previous section of the requirements that a real-time test system should follow, the most important demands for a real-time language are now introduced. After that, we try to fit the TTCN-3 language in this perspective, analyzing it as a real-time language for writing real-time test systems. Also, a discussion follows, concerning insufficiencies of TTCN-3 in defining real-time specifications and possible solutions for some of them are provided.

3.1 Requirements of a Real-Time Language

It is well known that a real-time software must be guaranteed to meet its timing constraints [3]. Therefore, one of the most important requirement of a real-time language is that it must be designed so that its *programs can be guaranteed to meet their deadlines at compile-time* (that means they should be calculability analyzable). Real-time applications must also be very reliable and have a long life expectancy. Therefore, the language should provide *strong typing*, *structured constructs* and *be modular*. To reduce the costs allocated for maintenance, real-time programs should be easy to understand, be readable and simple to modify. This general maintainability requirement is greatly aided if the language is *readable*, *well-structured* and *not very complex*. Moreover, a real-time language should provide error handling mechanisms. Because many real-time programs involve multiprogramming, a real-time language should include *process definitions* and *process synchronization mechanisms*.

TTCN-3 is an expressive testing language providing all the necessary features for writing reliable and maintainable test systems for a wide range of application domains. It is a modular and well-structured language for testing not only conformance, but also other qualitative and quantitative features of the targeted systems. TTCN-3 allows multiprogramming and distribution due to the concept of test component which can be associated with a task.

Despite its advantages, TTCN-3 is not expressive enough for designing real-time tests. There are several problems when an atempt is made to design such a test. The first problem is *the precision* of time when it is recorded or checked by the TS or when it is associated with certain events. There is the semantic of timers that was not intended for suiting real-time properties, but conceived only for catching (typically longterm) timeouts. When using timers, the measurement of durations is influenced by the TTCN-3 snapshot semantics and by the order in which receive and timeout operations are ordered in the alt statement. TTCN-3 makes no assumptions about the duration for taking and evaluating a snapshot. Thus, exact times cannot be measured using ordinary timers. As they are now, timers shall be used for detecting or provoking the absence of signals and to take care that a test case eventually terminates if it is blocked for some reason, but not for specifying real-time requirements. Furthermore, the process of matching the received feedback from the SUT against the expected templates may take arbitrary long, though finite in time, as it depends on the structure and size of the templates [4]. Having no restriction on the number of snapshots, on the structure of test data and on templates, introduces time *nondeterminism* and thus, the matching time cannot be properly estimated. Therefore, it would not be real-time. Nondeterministic *delays* are also introduced at the implementation level of different TTCN-3 primitives, for example, at the adaptor layer. For certain critical operations it is very important to have the possibility to impose *limits for the execution time*, upper and lower limits, and TTCN-3 language lacks in providing instruments for achieving measurements for these actual times of execution, or for imposing those time limits. Another problem is that timers are always local to a test component. They cannot be made global variables, and thus it is impossible to test real-time properties which are imposed on events that occur at different test components. This is also a matter of time *synchronization* and time *consistency* for different components, possibly running on different machines. When dealing with distributed systems it is important to have mechanisms for time synchronization.

Before tackling some of those problems of TTCN-3, by introducing new concepts to overcome the inconveniences, an overview of the existing work in the area is performed.

3.2 Related Work

There exist already several approaches that have been proposed in order to extend TTCN (Tree and Tabular Combined Notation, an earlier version of TTCN-3) and TTCN-3 for real-time and performance testing.

- *PerfTTCN* (*Performance TTCN*) [5] extends TTCN with concepts for performance testing, such as: *performance test scenarios* for the description of test configurations, *traffic models* for the description of discrete and continuous streams of data, *measurement points* as special observation points, *measurement declarations* for the definition of metrics to be observed at measurement points, *performance constraints* to describe the performance conditions that should be met, *performance verdicts* for the judgement of test results. The PerfTTCN concepts are introduced mainly on a syntactical level by means of new TTCN tables. Their semantics is described in an informal manner and realized by a prototype.
- *RT-TTCN* (*Real-Time TTCN*) [6] is an extension of TTCN in order to test *hard* real-time requirements. On the syntactical level, RT-TTCN supports the annotation of TTCN statements with two timestamps for earliest and latest execution times. On the semantical level, the TTCN snapshot semantics has been refined and, in addition, RT-TTCN has been mapped onto timed transition systems.
- *TimedTTCN-3* [7] is a real-time extension for TTCN-3, which covers most and *RT-TTCN* features while being more intuitive in usage. Moreover, the TimedTTCN-3 extensions are more unified than the other extensions by making full use of the expressiveness of TTCN-3. Therefore only a few changes to the language are needed. It introduces the following features:

A new test verdict to judge real-time behavior; *Absolute time* as a means to measure time and to calculate durations and this is the reason for using the operation **now** at the current local time retrieval; *Delays* to postpone the execution of statements as the new statement **resume** provides the ability to delay the execution of a test component; *Timed synchronization* for test components; TimedTTCN-3 supports the *timezones* concept, by which those test components can be identified and must be synchronized in time; *Online and offline evaluation* of real-time properties.

These new concepts are all useful and are utilized additionally for real-time test specifications. However, they are means of verifying the real-time properties of the SUT in particular and do not guarantee that the TS in itself is real-time, or that the TS is able to stimulate and respond timely to the SUT queries. In order to impose a real-time execution to the TS, a further introduction of control mechanisms at semantical and syntactical level of the language is described.

– *ContinuousTTCN-3* [8] introduces basic concepts and means for handling continuous real world data in digital environments. TTCN-3 is enhanced with concepts of stream-based ports, sampling, equation systems, and additional control flow structures to be able to express continuous behavior. In ContinuousTTCN-3 time is also very important, and the problem of imprecise timers is mentioned. The concept of global time is overtaken from TimedTTCN-3 and it is enhanced with the notion of sampling and sampling time.

In paper [4] some of the presented limitations of the existing treatment of time within TTCN-3 are illustrated as follows: the problems with snapshot semantics, with assignment of test verdict and with synchronization of distributed test configurations. The proposed approach is that of giving general guidelines for a more accurate measurements of the real-time using the actual capabilities of the language within its boundaries.

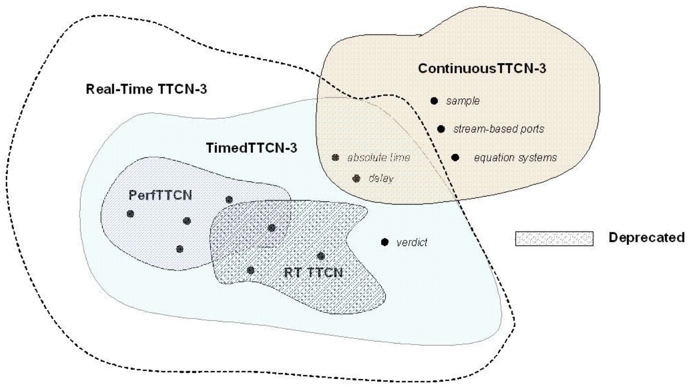


Fig. 2. TTCN-3 overview of proposed extensions

3.3 New Concepts for a Real-Time TTCN-3

In the following, we discuss and introduce new instruments for dealing with real-time requirements in order to solve the problems presented before. Carefully selected new additions to the language were developed in a minimal set of concepts build upon the predefined ones. The set is not fully developed yet and it only covers some aspects. The concepts are syntactically and semantically presented and integrated into the language.

Clocks And Timers. Precision timing and synchronization are key factors for real-time applications. Period clocks and timers are the basic instruments for achieving them. Therefore, we start the conceptual part with these elements, although they do not represent an original contribution of this paper. An equivalent definition for clocks is overtaken from TimedTTCN-3 [7] and timers are presented as they are already in the standard, relegating for the implementation the cumbersome work of making them real-time efficient.

A real-time clock is an incremental counter with fixed intervals, called the clock resolution. Generally, clocks are used for extracting absolute and local time values and for introducing intended delays into executions of test components. Absolute time is the time counted from a fixed point in time for the global system and the local time is the time counted on a local component. TTCN-3 does not provide the *clock*-concept, but thanks to the enhancements introduced by the TimedTTCN-3 [7], the functionality of a clock can be replaced. For extracting absolute and local time as a float value, representing the number of seconds from the established fixed point in time, primitive **now** is used. For local time retrieval the **now** operation should be applied to the **self** handle, for example, **self.now**. The primitive defined for introducing delays is **resume** which takes as argument an absolute time value, designating the moment for resuming its execution.

A timer is a complement of a clock. While a clock increments time, a timer decrements its value and generates a signal when that value reaches zero. Timers can be also used for counting relative time in a system where relative time is the time counted from a point in time, relative to the execution flow. In a general way, timers are used for controlling the sequence of event execution. There are situations when timers are used as standalone instructions, for example, when they are intended to delay processes or to indicate passage of time, and situations where they are used in dependence upon other statements, for example, associated to a receive event. In that case, the timer has the function to impose a time constraint on that event. Nevertheless, as discussed, it is not a reliable means for measuring the precise time. In order to keep consistence the old notion of timer is maintained as it is, but the implementation should overcome the limitations of snapshot semantics. We propose to implement each timer as a task that is programmed to trigger a signal after its expiration period of time.

In TTCN-3, events can be considered to be *dependent* or *independent* from each other. In Listing 1, *P.receive* and *T*1*.timeout* are dependent events because the execution of one influences the other, timer *T*1 imposes a time

Listing 1. Dependent and independent receive-timeout events

```
...                                                     1
port P;
timer T1, T2;                                           3
alt {
    [] P.receive(response){ ... }                       5
    [] T1.timeout{ ... }
}                                                       7
...
T2.start(100);                                          9
T2.timeout;
P.send(message);                                       11
...
```

constraint on the *P.receive* operation. In order to control this relation, tasks associated to these events, in fact to the statements, synchronize with each other. The way in which the alt instruction is exited depends upon which of the two operations is executed first.

In the case of independent events, presented in Listing 1, timer $T2$ functions as a delay for the entire process and is not a time constraint to determine the arrival of another event in a timely manner, for example, the sending of a message to happen in a given time. In Listing 1, lines 14 and 15 can be replaced by the **resume(self.now + 100)** statement.

The Try-Catch Statement. Time constraints are considered to be central elements to describe the real-time properties of a system. They are needed to control both the timing of the TS and the SUT, depending on the instructions to which the time constraints apply. When applied to the communication statements for receiving events, they indicate the responsiveness of the SUT. When applied to other simple or compound statements grouped together in blocks or standalone, those constraints indicate whether the TS itself respects certain time requirements.

To assure that a sequence of statements, or instructions, executes in a specified period of time, the instruction block can be protected by a **try-catch** construct. This wraps an instruction block by a time constraint together with an exception handling if the time constraint is not met. As an argument to the **try-catch** we have a float value which is preceded by the ***absolute*** or ***relative*** keyword. The float value represents a time value which can be absolute or relative. If dealing with an absolute value of the time it means that when reaching that point in time the execution of the enclosed instructions should have been terminated. Otherwise a real-time exception is generated. Whereas dealing with a relative time, the float value indicates the time units in which the block of statements should execute. If the time overpasses, the real-time exception is raised.

If the float value is negative or zero, a real-time exception is generated immediately. The exception can be handled immediately and the handling behavior is described in its following brackets. The instructions contained inside a **try-catch** statement can be simple or compound. They can be *ConfigurationStatements*, *TimerStatements*, *AltConstruct*, *RepeatStatements* or *CommunicationStatements* [9].

Listing 2. Generic try-catch statement bounding time execution for a set of statements

```
try (absolute|relative FloatValue) {

   Statement_1;
   Statement_2;
   ...
   Statement_N;

} catch (rtexception) {
   // exception handler
}
```

Real-time Exceptions. Time guards for instruction blocks that do not contain communication with the SUT designate the time that should elapse on the TS side. If the time constraint is not respected, the TS is responsible for a test failure and an exception handling mechanism for protecting against TS errors should be activated. A new type of exception shall be introduced, **rtexception**, which will occur only when a deadline is missed by the TS. Also, special exceptions should be introduced for the delays of the SUT. The exception handling mechanism can define alternative actions depending on the test requirements for hard, firm, soft, or real real-time systems:

– the test will be stopped and a verdict will be set to a value corresponding to the error;
– the test can continue if the missed deadlines were not critical for the system;
– the test can continue by repeating the statements for a specified number of times.

Special Instructions. The statements that require special attention are sending/receiving to/from a communication port. They are special because they enable the communication and interaction with the SUT. In many cases, it is necessary that a **send** operation must be executed in a timely manner. Another case is that sending to a port should be done every x time units where a time unit can vary from microseconds to hours, days or even more, depending on the test requirements. The construction in Listing 3 is very similar to the previous diagrams. However, a dedicated time constraint for a **send** statement is given. When encountering this construction, a separate thread is created for sending the message to the SUT. The first parameter represents the maximum duration of the **send** operation. The second parameter represents the interval between two consecutive send operations. The third parameter of the **try-catch** construction should be a integer value, indicating the number of times the message should be send to the SUT.

If the first parameter is negative or zero, an exception will be generated. If the second parameter is negative or zero or if the third parameter is zero, the sending of the package will not be repeated.

In this example, real-time exceptions could be obtained in three situations: (1) if the send instruction takes longer to execute than the time indicated by the first parameter (2) or if the send operation cannot be scheduled in time due to the overloading of the system (3) or an inappropriate scheduling politic. If

Listing 3. Time constraints imposed on a **send** operation

```
try (relative 0.1, 0.2, 3) {
   P.send(message);
} catch (rtexception rte) {
   // exception handling behavior
}
```

Listing 4. Time constraints imposed on a **receive** event

```
alt {
   try(relative 0.001) [] P.receive(message1) {
       Statement_1;
       Statement_2;
       ...
       Statement_N;
   } catch (rtexception rte){
       // exception handling behavior
   }
   try(relative 0.01)  [] P.receive(message2) {
       Statement_1;
       Statement_2;
       ...
       Statement_N;
   } catch (rtexception rte){
       // exception handling behavior
   }
}
```

the first parameter is preceded by absolute keyword, indicating absolute time, it will be automatically increased with the interval value at every cycle. The **try-catch** statement can also be used in the previous indicated manner with just one parameter, and then, the instruction will be executed only once.

In case of a **receive** statement, the constraint is put as part of the alternative statement and refers to the timed response of the SUT plus the time for matching. When a new message arrives, it will be matched conforming to the classical matching mechanism, and if it enters a branch, it verifies also if the timing requirement was respected or not. Depending on this it will execute further the block of statements contained inside the branch or the exception handling behavior.

The proposed **try-catch** statement is intended to be a powerful construct and therefore, the relation between this statement and other statements of TTCN-3 should be carefully analyzed. Other important example will be the combination with a **default** behavior. The **defaults** will be treated as a normal branch for an alternative. It is supposed that the **try-catch** statement guards the behavior defined for the associated **default**. All the situations will be approached when the operational semantic for all the introduced new concepts will be defined.

Code Spanning Limitations. We have introduced the concepts *time guards* and *real-time exceptions* in order to enable TTCN-3 to stipulate real-time test specifications. These test specifications can be used to check real-time properties of the SUT and to make the test system itself have a time-deterministic behavior. Nevertheless, there are elements in TTCN-3 which increase the code complexity and affect time determinism. Those elements are, for example, the **alt** statement which can have a large number of alternative branches, or a large number of nesting with other **alt** statements inside, **altstep** invocations, and similar. The type structure and nesting of templates are an additional load factor for the test system and in particular critical during the matching operation. There are other statements such as **for** loops or function invocations which enable an infinite number of executions. Therefore, in addition to the new concepts, we have to introduce a means to limit those aspects of a test in order to increase the chance of the TS to conform with its real-time requirements. For example, the test designer will be able to attach to an **alt** statement an upper limit for the number of branches and for the number of snapshots being taken into consideration (see Listing 5). Those limits are represented as integer numbers attached to the statements. Therefore, at compile time, a static prediction for the timings of the system can be obtained. At runtime, those timings will be also influenced by the load and other factors of the current state of the test system. This mechanism is extremely useful for the specialist to rapidly regulate the timing behavior of the test system without making great modifications into the code. Those limitations could be regarded as annotations introduced to bring in additional information into the system. This information could be used by the compiler for optimization purposes. Code spanning limitations could be gracefully manipulated by the test designer in order to deliberately separate important aspects of the functional behavior from the ones of real-time behavior. An alternative solution, using the existing resources of the language, would be to define a global boolean variable as for example **with_rt**. The variable can be used in an **if** statement with a **true** value to guard the behavior relevant for real-time evaluation and with a **false** value to guard the behavior important for functional evaluation, but which can be dropped when real-time tests are performed. Nevertheless, the solution proposed with the new constructs is more elegant and increases the flexibility and configurability of the tests. Based on the indication incrusted into the code, a great challenge would be the compiler itself, which should be designed in such a manner that it would be able to generate a fully optimized code, suitable for embedded systems.

In Listing 5 the first parameter of the **alt** statement indicates the number of the branches that should be taken into consideration and the second parameter indicates the number of snapshots that should be taken into account. The branches are considered in top down oder. The parameters are natural numbers.

4 Real-Time Platforms

The market of operating systems (OS) is continuously developing due to multiples and more sophisticated requirements. One of the key necessities is to support embedded real-time applications in which the OS must guarantee the timeliness and the correctness of the code processing. Many OS claim to be real-time operating systems (RTOS), but often only by reviewing the OS specifications, or arriving detailed information, can one truly identify those operating systems that enable real-time applications.

Listing 5. Limitations

```
// only the first 10 branches and
// only the first 2 snapshots
// are taken into consideration

alt (10, 2){
          [] Branch1{
                for(var integer i:=1; i<3; i++){
                    ...
                }
          }
          [] Branch2{...}
           ...
          [] Branch10{...}
          [] Branch11{...}
          [] Branch12{...}
}
```

The process of selecting the right RTOS is important and, at the same time, critical. It involves knowing all the specifications of different real-time operating systems, in an abundant market of available real-time operating systems, from micro kernels to commercial ones. The design space available to an RTOS is very broad. Selecting the RTOS based on specific features is a multidimensional search problem where each dimension corresponds to a RTOS characteristic. This requires an exhaustive research quest, tremendous computing resources and valuable time.

A wide variety of real-time operating systems are available to suit most projects and pocketbooks [10], [11]. Our search revealed sixteen real-time operating systems that deserve worth further investigation. These were merely the ones that could be applied to construct a real-time test system. We were specially interested in currently maintained open source projects. This consideration left four RTOS candidates ([12], [13], [14], [15]) to be evaluated in detail and ranked conforming to our specific requirements.

Based on specific requirements from the automotive domain and on obvious need for performance, reliability and cost-effectiveness common to every real-time project, we have divided the selection criteria in two parts. First, one envelopes general points of view such as supported languages, portability, latest update, commercial status, available API and information about development and support (Table 1). Secondly, it includes more specific features of real-time operating systems such as scheduling algorithms, type of RT (soft of hard), priority levels, kernel ROM size, kernel RAM size, multi-process support, interrupt latency, task switching time, type of interprocess communication (IPC) mechanism, memory management, task management and so forth.

Based on our specific requirements, we selected FreeRTOS as the most suitable RTOS. FreeRTOS is a very small, simple and concise operating system, making it suitable for small applications on small platforms. Since the majority of code is written in C language, it is highly portable and has been ported to many platforms. A strong advantage of FreeRTOS is that the code includes a demo project for each supported platform, demonstrating how to use the code on that specific platform. Unfortunately, this feature was not found at the other systems, where installation, configuration and development required more effort.

Table 1. A General RTOS Selection

	eCos	**FreeRTOS**	**RTAI**	**RTEMS**
Languages Support	Assembly, C, C++, Ada95	C, Assembly	C	C, C++, Ada95
Target CPUs Support	x86, PowerPC, ARM, MIPS, Altera NIOS II, Calmrisc16/32, Freescale 68k ColdFire, Fujitsu FR-V, Hitachi H8, Hitachi SuperH, Matsushita AM3x, NEC V850, SPARC	ARM architecture (ARM7, Cortex-M3) , AVR, AVR32, HCS12, MicroBlaze, MSP430, PIC microcontroller (PIC18, PIC24, dsPIC), Renesas H8/S, x86, 8052	X86 (with and without FPU and TSC), PowerPC, ARM (StrongARM; ARM7: clps711x-family, Cirrus Logic EP7xxx, CS89712, PXA25x), MIPS	ARM, Blackfin, ColdFire, Texas Instruments C3x/C4x DSPs, H8/300, x86, 68K, MIPS, Nios II, PowerPC, SuperH, SPARC
Development Status	eCos 2.0, May 2003	FreeRTOS 4.4.0, July 2007	RTAI 3.5, February 2007	RTEMS 4.6.6, April 2006
Source Model/ License	Open source/ eCos License (GPL with exceptions)	Open source/ Modified GPL	Open source	Open source/ Modified GPL
API	POSIX (1003.1b), ITRON, "classic/native" API in C and Ada	Well written custom API, based on "classic" API in C	Custom API derived from RTLinux V1 API	uITRON 3.0 API, POSIX 1003.1b, BSD standards
Development Information/- Support	Books, papers/ Mailing list	Web tutorials/ Forum	Incomplete documentation/ Web support, mailing list	Wiki/ Contractual support

Its strength is in its small size, making it possible to run where most other operating systems would not fit.

5 A Practical Example

The actual mapping of the basic and new introduced concepts of TTCN-3 language to a RTOS platform is explained by a simple test case for an example taken from the automotive domain. The chosen example consist of an embedded system on a MC9S12NE64 demo board [16] attached to a car door. The system is controlling various basic units of the door, for example, the window lifter, flashing light indicator, electrical central locking system. The system changes its internal states when receiving signals via RS232 serial interface. Those signals are in fact, some control strings with the role of triggering a special basic functionality of the door such as driving up the window, turning on the flash indicator and so forth. When receiving such a string, the system enters a different

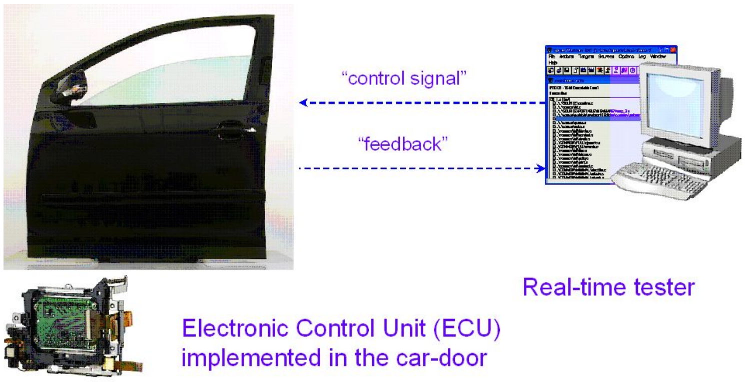


Fig. 3. Test Setup

state, executes the function associated with that state, and sends back a response string to the serial interface. The basic functions of the door could be combined so as to form safety applications such as when an accident occurs, the window should be driven down, the flash indicator should blink and the door should be automatically unlocked. Therefore, it is important to assure that the basic functionality is happening real-time. One can put several real-time constraints on such as: "the window should be driven down within ten milliseconds; the flash signal should be turned on within one millisecond; it should remain on for ten milliseconds then turn off."

In Figure 3 the test settlement is presented. On the left side we have the embedded system connected to the door, and on the right there is the TS consisting of a PC upon which is installed the FreeRTOS. The PC is an Intel Pentium 4 CPU with 3.20GHz and 1.00 GB of RAM. The operating system installed on the PC is Windows XP Professional version 2002. FreeRTOS has been optimized for an embedded system environment, having lack of resources. Although our aim is to develop a TS based on an embedded platform, in the development phase we prefer to use a PC with a port for FreeRTOS that runs on an integrated environment from the WATCOM open source project [17] (the distribution is for Windows). The PC is connected to the board through a serial cable. The real-time requirement that one wishes to test can be formulated in this manner: "The flash signal should be turned on within 30 milliseconds." For testing this requirement only, the signal is sent several times instructing for a repeated flashing. After sending one signal, and the flash is on, it is assumed that it automatically turns off after a fixed period of time. After expiring this period of time one can send another signal for turning it on again. This example epitomizes the new introduced concepts.

5.1 TTCN-3 Test Specification

The TTCN-3 partial code of the behavior of the real-time test component is shown in Listing 6. The mapping between the port of the test component and

Listing 6. TTCN-3 Test Specification Sample Code

```
testcase rtTest() runs on RTComponent system System {
    ...                                                                      2
    map(self.P, system.P);
    // sending the string for bringing the system into initial state        4
    P.send(SYS_INIT_CODE);
    try(relative 0.03){                                                      6
        P.receive(feedbackInit);
    }catch(var rtexception rte){                                             8
        log(rte);
        setverdict(fail);                                                   10
    }
    // periodically sending the receive statement                           12
    try(relative sendDelay, sendPeriod, nrOfTimes){
        P.send(BLINK_ON_CODE);                                              14
    }catch(var rtexception rte) {
        log(rte);                                                           16
        setverdict(fail);
    }                                                                       18
    for(var int i:= 0; i<=nrOfTimes; i:=i+1){
        try(relative 0.03){                                                 20
            P.receive(feedbackBlinkOn){
                finished := finished + 1;                                   22
                if(finished == nrOfDatasets) {
                    setverdict(pass);                                       24
                    stop;
                }                                                           26
            }
        }catch(var rtexception rte){                                        28
            log(rte);
            setverdict(fail);                                               30
        }
    } // end for                                                            32
} // end testcase
```

the port of the abstract SUT is performed (Line 3). This mapping should be implemented for the serial port. Then, the initialization string is sent to the SUT without any special time requirement (line 6). We wait for the SUT to move into its initial state and to send back a confirmation string which we capture (Lines 8-13). We introduce here the new **try-catch** statement for imposing time requirements to the receive operation. We should expect the feedback from the SUT no more than 30 milliseconds. This should be a real-time restriction for the SUT and therefore, the implementation of receive operation should introduce no delay. If there is no message received from the SUT within the indicated time, then a real-time exception will be captured and logged, and the verdict of the test is **fail**. After the initialization phase we send the command string for turning on the flash signaller for a number of times (Lines 15-20). We can observe here the new construct introduced for cyclic real-time restricted behavior. The first **send** should be triggered after *sendDelay* relative delay, and the next send operation should occur after every *sendPeriod* interval. The *sendPeriod* represents also the time required for the flash signaler to automatically turn off. We are expecting the feedback from the SUT for every send message in a **for** loop (Lines 23-37). For the **receive** operation inside the loop (Lines 24-36), we have the time constraint of 30 milliseconds expressed in the similar manner as previously described. This means that in 30 milliseconds from the **send** operation, we should receive confirmation from the SUT, which means that the function was executed and the flash signaler was turned on.

Listing 7. TTCN-3 FreeRTOS tSend Task Code

```
void v_tSend(void *pvParameters) {                                                          1
   ...
   for( ;; ) {                                                                              3
     if(i==0) vTaskDelay(tSendDelay); /* Suspend the current task for a given time */
     else vTaskDelay(tSendPeriod);                                                          5
     ...
     time1[i] = xTaskGetTickCount();  /* Send the string to the serial port */             7
     vSerialPutString( xPort, codes[codeId], strlen(codes[codeId]));
     /* Create the tExp task in order to wait for the response for this data set */        9
     xTaskCreate( v_tExp, tExpName, STACK_SIZE, &i, mainMTC_TASK_PRIORITY, &tExpHandle[i] );
     i++;  //increment the counter i                                                        11
   }
}                                                                                           13
```

Listing 8. TTCN-3 FreeRTOS tExp Task Code

```
void v_tExp(void *pvParameters) {                                                           1
  ...
  for(;;) {                                                                                 3
      vTaskDelay(tExpConst);
      ...                                                                                   5
      vTaskEndScheduler();
  }                                                                                         7
}
```

5.2 Test System Implementation at the FreeRTOS Level

The implementation using the FreeRTOS API is complex and not very easy to read. Therefore, we intent to illustrate only a few samples representing the basic implementation of *tExp* timers, the implementation of the thread associated with the *send* instruction, as well as, the implementation of the thread associated to the *receive* instructions. These aspects are illustrated in Listings 7, 8 and 9 respectively. The synchronization between the processes is made using primitives $vTaskDelay$, for delaying a task for a defined period and $vTaskEndScheduler$, for interrupting the execution of other threads, which were provided by FreeRTOS.

It is important to note that these tasks are dependent on each other and therefore they are synchronized accordingly. Each time the *tSend* task sends a message to the serial port (Line 10, Listing 7), it creates also a *tExp* task (Line 15 and 15, Listing 7) corresponding to a timer. It can be observed that the first *send* operation is delayed with *tSendDelay* and the following are sent after each *tSendPeriod* expiration (Lines 5 and 6, Listing 7).

Each timer task contains a $vTaskDelay$ instruction (Line 4, Listing 8) which delays the running of the task for a period $tExpConst$ that represents the timer's expiration time. If the given time elapses, the timer task becomes active. By using of the primitive function $vTaskEndScheduler()$ all the other processes are killed (Line 6,Listing 8) and the test execution is finished. However, this is not done before assigning the verdict $fail$ to the current test (Lines 11 and 17, Listing 8). From the behavior of the receive thread we can observe that if a message was received (Line 5, Listing 9) and the message corresponds to the expected one (line 8, Listing 9), the timer thread associated with that message is killed (Line 12, Listing 9). The time difference between the sending of

Table 2. The Results For The Presented Example

Sent_at (ticks)	Received_at (ticks)	Interval (ticks)	Constraint (ticks)	Verdict
2000	2002	2	3000	**pass**
3000	3001	1	3000	**pass**
4000	4001	1	3000	**pass**
5000	5118	118	3000	**pass**
6000	6002	2	3000	**pass**
7000	7001	1	3000	**pass**
8000	8031	31	3000	**pass**
9000	9001	1	3000	**pass**
10000	10001	1	3000	**pass**
11000	11541	541	3000	**pass**

Listing 9. TTCN-3 FreeRTOS tReceive Task Code

```
void v_pReceive(void *pvParameters){
   ...                                                                          2
   for( ;; ) {
      ...                                                                       4
      xGotChar = xSerialGetChar( xPort, &cRxedChar, xBlockTime );
      ...                                                                       6
      if(match(responseStr,responses[resId])) { // validate the response
         time2[i] = xTaskGetTickCount();                                        8
         timedif[i] = time2[i] - time1[i];
         ...                                                                    10
         vTaskDelete(tExpHandle[i]);
      }                                                                         12
   }
}                                                                               14
```

a message and the response of the message is calculated (Line10, Listing 9). Therefore that timer is deactivated.

We observe that the function for retrieving the time is *xTaskGetTickCount* (Line 9, Listing 7 and Line 9, Listing 9). For a tick rate of 10^5 Hz, we obtained the numbers listed in Table 2. All the tests were passed. The table contains the relative times, measured in numbers of clock ticks for sending and for receiving the message. The real-time constraint was 3000 ticks which means 30 milliseconds.

6 Conclusions and Future Work

This paper demonstrates a possibility of realizing real-time test systems by using the TTCN-3 testing language. The extended version of TTCN-3 presented in this paper has features for describing, analyzing and testing real-time properties of systems. The basic language concepts together with new concepts for defining time constraints, handling time-critical behavior and so forth are employed.

Although the project is in an early stage that covers only parts of an extended TTCN-3 mapping to a RTOS platform, the first objectives were already achieved:

– description and design of simple real-time tests using the existing and newly introduced TTCN-3 features;
– experiments with a selected RTOS platform;
– experiments on different ways of realizing TTCN-3 real-time concepts using the primitives provided by the real-time operating system platform;
– realization of a simple real-time test which demonstrates the power of the new concepts.

This paper provides only a validation of real-time tests through empirical results. A formal validation was not targeted in this paper, but it may represent the subject of a further research. Also, further work is needed for analyzing the full potential of enriching TTCN-3 with dedicated real-time features. The fundament of the work will be a specific real-time semantic for TTCN-3, which limits current unlimited elements of TTCN-3 executions, for example, by putting an upper number for the number of snapshots per alternative statement and so forth. Only this will allow to give execution time limits for TTCN-3 statements, making altogether TTCN-3 a real-time test language. Furthermore, our implementation of the new features and semantics on a real-time operating system will be continued. For that, this analysis and experimentation with the API's of other real-time operating systems such as RTAI or RTEMS will be continued. As long as the goal is to run TTCN3 inside an embedded operation system, a very advanced TTCN3 compiler should be developed, which is the greatest challenge of all. Although the derivation of code for embedded systems from abstract test specification may seem at first sight to be a cumbersome task, this goal can be achieved through efficient and well optimized mapping patterns.

References

1. Osterweil, L.: Strategic directions in software quality. ACM Comput. Surv. 28(4), 738–750 (1996)
2. Myers, G.J.: The Art of Testing, 1st edn. John Wiley & Sons, Chichester (1979)
3. Halang, W.A., Stoyenko, A.D.: Constructing Predictable Real Time Systems. Kluwer Academic Publishers, Norwell (1991)
4. Sinnott, R.: Towards more accurate real-time testing. In: The 12th International Conference on Information Systems Analysis and Synthesis (ISAS 2006) (2006)
5. Schieferdecker, I., Stepien, B., Rennoch, A.: Perfttcn, a ttcn language extension for performance testing, `citeseer.ist.psu.edu/243959.html`
6. Walter, T., Grabowski, J.: Real-time ttcn for testing real-time and multimedia systems (1997), `http://citeseer.ist.psu.edu/walter97realtime.html`
7. Dai, Z., Grabowski, J., Neukirchen, H.: Timed ttcn-3 - a real-time extension for ttcn (2002), `http://citeseer.ist.psu.edu/article/dai02timedttcn.html`
8. Schieferdecker, I., Gromann, J.: Testing embedded control systems with ttcn-3. In: Testing embedded control systems with TTCN-3, pp. 125–136. Springer, Berlin, Heidelberg (2007)
9. ETSZ ES201 873-1 V3.2.1, Methods for testing and specification (mts); the testing and test control notation version 3; part 1: Ttcn-3 core language (2007-02)
10. Dedicated systems encyclopedia rtos list, `www.dedicated-systems.com/encyc/buyersguide/products/Dir1048.html`

11. Frequently asked questions of the comp. realtime newsgroup, `http://www.omimo.be/encyc/publications/faq/rtfaq.htm`
12. Freertos — open source, mini real time kernel, `http://www.freertos.org`
13. Rtai — the realtime application interface for linux, `http://www.rtai.org`
14. ecos — open source real-time operating system intended for embedded applications, `http://www.ecos.sourceware.org`
15. Rtems — real-time operating system for multiprocessor systems, `http://www.rtems.com`
16. Mc9s12ne64 demonstration kit, `http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=DEMO9S12NE64`
17. Open source watcom c, c++, and fortran cross compilers and tools, `http://www.openwatcom.org`
18. Neukirchen, H.W.: Languages, tools and patterns for the specification of distributed real-time tests. Ph.D. dissertation, Mathematisch-Naturwissenschaftlichen Fakultten der Georg-August-Universitt zu Gttingen (2004)
19. Molovata, V.: Realizing real-time test systems using ttcn-3, diploma thesis (2007)
20. Grabowski, J., Hogrefe, D., Réthy, G., Schieferdecker, I., Wiles, A., Willcock, C.: An introduction to the testing and test control notation (ttcn-3). Comput. Networks 42(3), 375–403 (2003)

Author Index