1st French Singaporean Workshop on Formal Methods and Applications

FSFMA'13, July 15–16, 2013, Singapore

Edited by

Christine Choppy

Jun Sun

Editors

Christine Choppy
Université Paris 13, Sorbonne Paris Cité,
LIPN, CNRS UMR 7030, France
christine.choppy@lipn.univ-paris13.fr

Jun Sun
Singapore University of Technology and Design
Singapore
sunjun@sutd.edu.sg

ACM Classification 1998
D.2.4 Formal methods

ISBN 978-3-939897-56-9

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-939897-56-9.

Publication date
July, 2013

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Digital Object Identifier: 10.4230/OASIcs.FSFMA.2013.i

ISBN 978-3-939897-56-9 **ISSN 2190-6807** **http://www.dagstuhl.de/oasics**

OASIcs – OpenAccess Series in Informatics

OASIcs aims at a suitable publication venue to publish peer-reviewed collections of papers emerging from a scientific event. OASIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 2190-6807

www.dagstuhl.de/oasics

Contents

1st French Singaporean Workshop on Formal Methods and Applications (FSFMA'13).
Editors: C. Choppy, J. Sun
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

We are pleased to present the proceedings of the 1st French Singaporean Workshop in Formal Methods and Applications (FSFMA) which will take place on July 15–16th, 2013 in Singapore, as a satellite event before the Eighteenth International Conference on Engineering of Complex Computer Systems (ICECCS 2013).

FSFMA 2013 aims at sharing research interests and launching collaborations in the area of formal methods and their applications. The scientific subject of the workshop covers areas such as formal specification, model checking, verification, program analysis/transformation, software engineering, and applications in major areas of computer science, including aeronautics and aerospace. It is hoped that this workshop will help to establish links between academic and industry scientists interested in methods and techniques for constructing reliable systems using formal methods.

The workshop consists in two keynote talks, the presentation of peer-reviewed papers (there are regular papers and PhD papers), and two panel discussions.

The keynote speakers are Laurent Fribourg (LSV, France), and Wei-Ngan Chin (NUS, Singapore). Laurent Fribourg is head of the Laboratoire de Spécification et Vérification (ENS de Cachan & CNRS, France), and his talk is about the use of formal techniques to analyse properties of dynamic systems. Wei-Ngan Chin is Associate Professor in the Department of Computer Science, National University of Singapore, and his talk is about improving the specification and verification processes.

The panel discussions will be on applications of formal methods, and on collaborations between academic and industry, in different countries.

We received 22 submissions (3 were removed, 10 regular papers and 9 PhD session papers), and selected 10 papers (3 regular papers, and 7 PhD session papers).

FSFMA 2013 is funded by the Merlion programme of the French Institute in Singapore of the French embassy in Singapore. The workshop is partially funded by the French Institute and the Singapore University of Technology and Design, with the additional support of the Laboratoire d'Informatique de Paris Nord (Sorbonne Paris Cité, Université Paris 13 & CNRS UMR 7030, France), the Nanyang Technology University (Singapore), the National University of Singapore (Singapore), and the kind participation of the Laboratoire de Spécification et Vérification (ENS de Cachan & CNRS, France).

We would like to thank all program committee members, subreviewers, authors and participants for their involvement to the success of the workshop. We would also like to warmly thank Aurélie Martin, Florent Beau and Pascal Loubière, from the French Institute in Singapore, for their helpful support.

Christine Choppy and Jun Sun
Chairs of FSFMA 2013

1st French Singaporean Workshop on Formal Methods and Applications (FSFMA'13).
Editors: C. Choppy, J. Sun
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

List of Authors

Étienne André
Université Paris 13, Sorbonne Paris Cité,
LIPN, CNRS UMR 7030
93430 Villetaneuse, France
Etienne.Andre@univ-paris13.fr

Philippe Baufreton
Sagem - SAFRAN Electronics
91344 MASSY Cedex France

Patrice Carle
ONERA — The French Aerospace Lab
91123 Palaiseau, France
patrice.carle@onera.fr

Wei-Ngan Chin
School of Computing
National University of Singapore
chinwn@comp.nus.edu.sg

Christine Choppy
Université Paris 13, Sorbonne Paris Cité,
LIPN, CNRS UMR 7030
93430 Villetaneuse, France
Christine.Choppy@lipn.univ-paris13.fr

Martin Elshuber
Institute of Computer Engineering
Vienna University of Technology
1040 Wien, Austria
martine@vmars.tuwien.ac.at

Laurent Fribourg
LSV, CNRS & ENS de Cachan
94235 Cachan, France
Laurent.Fribourg@lsv.ens-cachan.fr

Susanne Kandl
Institute of Computer Engineering
Vienna University of Technology
1040 Wien, Austria
susanne@vmars.tuwien.ac.at

Romain Kervarc
ONERA — The French Aerospace Lab
91123 Palaiseau, France
romain.kervarc@onera.fr

Ulrich Kühne
University of Bremen
Bremen, Germany

Yan Liu
National University of Singapore
yanliu@comp.nus.edu.sg

Bruno Monsuez
UIIS, ENSTA ParisTech
91762 Palaiseau Cedex, France
surname@ensta-paristech.fr

Vladimir-Alexandru Paun
UIIS, ENSTA ParisTech
91762 Palaiseau Cedex, France
surname@ensta-paristech.fr

Ariane Piel
ONERA — The French Aerospace Lab
91123 Palaiseau, France
ariane.piel@onera.fr

Jonathan Protzenko
INRIA
Rocquencourt, France
jonathan.protzenko@ens-lyon.org

Peter Puschner
Institute of Computer Engineering
Vienna University of Technology
1040 Wien, Austria
peter@vmars.tuwien.ac.at

César Rodríguez
LSV, ENS Cachan & CNRS, INRIA Saclay
94235 Cachan Cedex, France
cesar.rodriguez@lsv.ens-cachan.fr

Yanti Rusmawati
School of Computer Science, The University of Manchester
Oxford Road, Manchester M13 9PL, UK
rusmaway@cs.man.ac.uk

David Rydeheard
School of Computer Science, The University of Manchester
Oxford Road, Manchester M13 9PL, UK
david@cs.man.ac.uk

1st French Singaporean Workshop on Formal Methods and Applications (FSFMA'13).
Editors: C. Choppy, J. Sun
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Stefan Schwoon
LSV, ENS Cachan & CNRS, INRIA Saclay
94235 Cachan Cedex, France
stefan.schwoon@lsv.ens-cachan.fr

Romain Soulat
LSV, CNRS & ENS de Cachan
94235 Cachan, France
Romain.Soulat@lsv.ens-cachan.fr

Somsak Vanit-Anunchai
School of Telecommunication Engineering
Institute of Engineering
Suranaree University of Technology
Muang, Nakhon Ratchasima, Thailand
somsav@sut.ac.th

Program Committee

Chairs

Christine Choppy
Université Paris 13, Sorbonne Paris Cité,
CNRS UMR 7030, France

Jun Sun
Singapore University of Technology and
Design, Singapore

PhD Session Chairs

Étienne André
Université Paris 13, Sorbonne Paris Cité,
CNRS UMR 7030, France

Yang Liu
Nanyang Technological University, Singapore

Program Committee

Étienne André
Université Paris 13, Sorbonne Paris Cité,
CNRS UMR 7030, France

Christine Choppy
Université Paris 13, Sorbonne Paris Cité,
CNRS UMR 7030, France

Jörg Desel
Fernuniversität in Hagen, Germany

Jin-Song Dong
National University of Singapore, Singapore

Maritta Heisel
University of Duisburg-Essen, Germany

Romain Kervarc
ONERA, France

Kaïs Klaï
Université Paris 13, Sorbonne Paris Cité,
CNRS UMR 7030, France

Lars M. Kristensen
Bergen University College, Norway

Ulrich Kühne
University of Bremen, Germany

Charles Lakos
University of Adelaide, Australia

Yang Liu
Nanyang Technological University, Singapore

Hadj-Alouane Nejib
ENIT Tunis, Tunisia

Geguang Pu
East China Normal University, China

Shengchao Qin
University of Teesside, Middlesbrouq, U.K

Gianna Reggio
DIBRIS, Genova, Italy

Jun Sun
Singapore University of Technology and
Design, Singapore

Naijun Zhang
Chinese Academy of Sciences, China

1st French Singaporean Workshop on Formal Methods and Applications (FSFMA'13).
Editors: C. Choppy, J. Sun
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

List of Subreviewers

Florin Craciun
Babes-Bolyai University, Romania

Melanie Diepenbeck
University of Bremen, Germany

Hoang Le
University of Bremen, Germany

Jianwen Li
East China Normal University, China

Rene Meis
University of Duisburg-Essen, Germany

Truong Khanh Nguyen
National University of Singapore, Singapore

Yidong Sheng
Chinese Academy of Sciences, China

Tian Huat Tan
National University of Singapore, Singapore

Hengjun Zhao
Chinese Academy of Sciences, China

1st French Singaporean Workshop on Formal Methods and Applications (FSFMA'13).
Editors: C. Choppy, J. Sun

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Control of Switching Systems by Invariance Analysis (Invited Talk)

Laurent Fribourg

LSV, CNRS & ENS de Cachan
94235 Cachan, France
Laurent.Fribourg@lsv.ens-cachan.fr

Abstract

Switched systems are embedded devices widespread in industrial applications such as power electronics and automotive control. They consist of continuous-time dynamical subsystems and a rule that controls the switching between them. Under a suitable control rule, the system can improve its steady-state performance and meet essential properties such as safety and stability in desirable operating zones. We explain that such controller synthesis problems are related to the construction of appropriate invariants of the state space, which approximate the limit sets of the system trajectories. We present a new approach of invariant construction based on a technique of state space decomposition interleaved with forward fixed point computation. The method is illustrated in a case study taken from the field of power electronics.

This work is a joint work with Romain Soulat.

1998 ACM Subject Classification D.2.4 Formal methods

Keywords and phrases Control theory, Hybrid systems, Safety, Stability

Digital Object Identifier 10.4230/OASIcs.FSFMA.2013.1

Short Biography

Laurent Fribourg is a CNRS Senior Researcher working at École Normale Supérieure de Cachan (ENSC), France. Since 2007, he has been Scientific Coordinator of Institut Farman, which federates interdisciplinary projects between 5 Laboratories of ENSC. He has been Director of LSV, the Computer Science Lab. of ENSC since 2011. He has written more than 70 international publications in the field of Logic Programming, Program Testing, and Model Checking.

1st French Singaporean Workshop on Formal Methods and Applications 2013 (FSFMA'13).
Editors: Christine Choppy and Jun Sun; pp. 1–1

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Specification, Verification and Inference (Invited Talk)

Wei-Ngan Chin

School of Computing
National University of Singapore
chinwn@comp.nus.edu.sg

Abstract

Traditionally, the focus of specification mechanism has been on improving its ability to cover a wider range of problems more accurately, while the effectiveness of verification is left to the underlying theorem provers. Our work attempts a novel approach, where the focus is on designing good specification mechanisms that can achieve both better expressiveness and better verifiability. Moreover, we shall also highlight a unified specification mechanism that can be used for both verification and inference. Our framework allows preconditions and postconditions to be selectively inferred via a set of uninterpreted relations which are computed using bi-abduction, and modularly synthesized to support concise specification for program codes.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Expressive Specification, Automated Verification, Specification Inference

Digital Object Identifier 10.4230/OASIcs.FSFMA.2013.2

Short Biography

Wei-Ngan Chin is presently an Associate Professor in the Department of Computer Science, National University of Singapore. His research interests are in programming languages and software engineering. He has worked on various program analyses and verification techniques that are aimed at improving clarity, reliability and reusability of software.

1st French Singaporean Workshop on Formal Methods and Applications 2013 (FSFMA'13).
Editors: Christine Choppy and Jun Sun; pp. 2–2

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Analysis of Two-Layer Protocols: DCCP Simultaneous-Open and Hole Punching Procedures*

Somsak Vanit-Anunchai

School of Telecommunication Engineering
Institute of Engineering
Suranaree University of Technology
Muang, Nakhon Ratchasima, Thailand
email:somsav@sut.ac.th

Abstract

The simultaneous-open procedure of the Datagram Congestion Control Protocol (DCCP), RFC 5596, was published in September 2009. Its design aims to overcome DCCP weaknesses when the Server is behind a middle box, such as Network Address Translators or firewalls. The original DCCP specification, RFC 4340, only allows the Client to initiate the call. The call request cannot reach the Server behind the middle box. A widely used solution to address this problem is called the "hole punching" technique. This technique requires the Server to initiate sending packets. Using Coloured Petri Nets (CPN) this paper models and analyses the DCCP procedure specified in RFC 5596. However, the difficulty is that detailed modelling of the address translation is also required. This causes state space explosion. We alleviate the state explosion using prioritized transitions and the sweep-line technique. Modelling and analysis approaches are discussed in the hope that it is helpful for others who wish to analyse similar protocols. Analysis results are also obtained for the simultaneous-open procedure specified in RFC 5596.

1998 ACM Subject Classification C.2.2 Network Protocols, D.2.2 Design Tools and Techniques, D.2.4 Software/Program Verification

Keywords and phrases Network Address Translators, Coloured Petri Nets, Sweep-line Method, Prioritized Transitions.

Digital Object Identifier 10.4230/OASIcs.FSFMA.2013.3

1 Introduction

The Datagram Congestion Control Protocol (DCCP) [18] is a transport protocol that provides bidirectional flow of data for applications that prefer timeliness to reliability. It is a connection-oriented protocol operating over the Internet between two entities, the Client and the Server. Originally specified in RFC 4340, only the Client can initiate the connection while the Server passively listens to the incoming request. When the Server is located in a private network or behind a Network Address Translator (NAT[1]), the first incoming packet cannot reach the Server because address mapping in the NAT does not exist yet. To overcome this problem, a simple solution widely used with other transport protocols (UDP, TCP and SCTP) is known as the "hole punching" technique.

* This work is supported by Research Grant from the Thai Network Information Center Foundation and the Thailand Research Fund.

[1] NAT is a middlebox that maps private (IP addresses - port number) to public (IP addresses - port number) and allows many hosts behind NAT to share the same public IPv4 address.

1st French Singaporean Workshop on Formal Methods and Applications 2013 (FSFMA'13).
Editors: Christine Choppy and Jun Sun; pp. 3–17

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

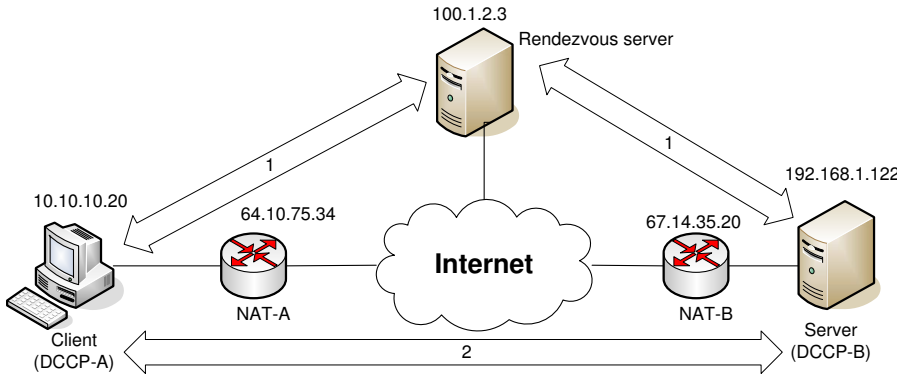


Figure 1 Peer-to-peer communication with rendezvous server.

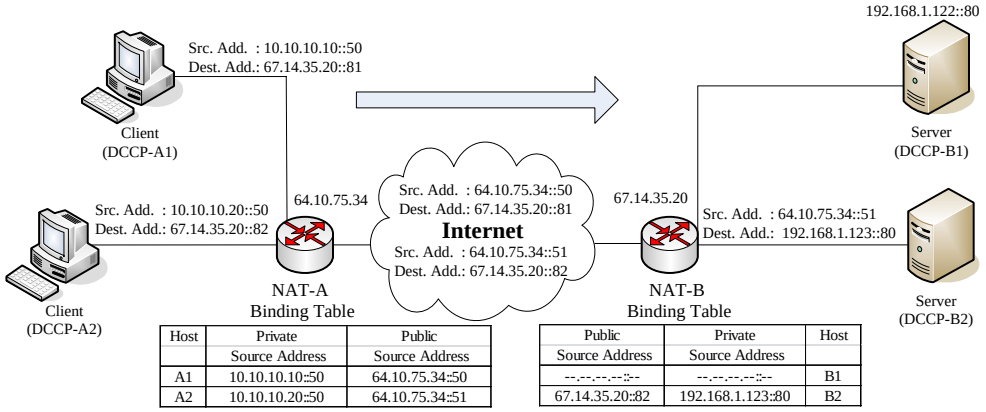


Figure 2 Binding tables in NAT-A and NAT-B.

The basic idea of hole punching consists of two phases, labeled 1 and 2, as shown in Fig. 1. Firstly, DCCP-A and DCCP-B, which are located behind NAT-A and NAT-B respectively, establish connections with a rendezvous server at a well-known public IP address (100.1.2.3). Because DCCP entities initiate the session via their NAT to the rendezvous server, the server can observe the public IP addresses and port numbers assigned for both sessions. The server then informs each entity of the public IP address and port number of its peer. After receiving this information, the connection between DCCP-A and DCCP-B can start. Illustrated in Fig. 2, when DCCP-A1 sends a packet to DCCP-B1, a "binding table" (or a hole) in NAT-A is created. However the packet is blocked by NAT-B because NAT-B has no binding for DCCP-B1 yet. Thus DCCP-B1 needs to send its packet to DCCP-A1 in order to create a binding table in NAT-B. After a hole is punched in NAT-B, the public address (67.14.35.20::81) associated with DCCP-B1 in the incoming packet will be translated to the private address of DCCP-B (192.168.1.122::80) so that the packet can be locally forwarded to DCCP-B1. In the hole punching scenarios, the Client and the Server initiate sending a packet at about the same time. This requires a new simultaneous open procedure as described in RFC 5596 [9].

1.1 Previous Work

Since 2003 we have constructed, refined and analysed Coloured Petri Net (CPN) [16] models of DCCP's connection management procedure according to RFC 4340, using Design/CPN [8].

In [24], we reported our experience with the incremental enhancement and iterative modelling of the connection management procedures as the DCCP specification was developed. Insight into the decisions behind the modelling choices can also be found in [24]. The full CPN specification of the connection management procedures can be found in Section 2 of [25]. Section 4 of [25] also explains the development of progress mappings for sweep-line state space analysis [21] of DCCP. We have published an enhanced version of [24] which also discusses a procedure-based model of DCCP's connection management procedures [5]. In [6], we discuss how to embed a parameterised channel into CPN models of protocols, using DCCP as an example.

1.2 Contributions

The contribution of this paper is two-fold. Firstly, we extend the Coloured Petri Net model and analysis of the DCCP connection management procedure (RFC 4340) in [5,23] to include the simultaneous open procedure (RFC 5596). Secondly, since embedding NAT with the hole punching procedure as a channel module [6] leads to significant state explosion, we demonstrate methods to circumvent the problem using prioritized transitions and the sweep-line technique [7].

1.3 Organisation

This paper is organised as follows. Section 2 provides an overview of the DCCP simultaneous open procedure. Modelling approach is discussed in Section 3. A description of the CPN model of DCCP's simultaneous open procedure is given in Section 4. Section 5 discusses analysis approach. Section 6 presents the experimental results, with Section 7 providing conclusions and future work.

2 DCCP Overview

2.1 Connection Management Procedures

The Datagram Congestion Control Protocol [17, 18] is a point-to-point transport protocol operating over the Internet between two DCCP entities, the Client and Server. It provides a bidirectional flow of data for applications, such as voice and video, that prefer timeliness to reliability. DCCP is designed to provide congestion control for these applications [11]. Its congestion control algorithms require statistics on packet loss because loss is related to the level of congestion in the network. DCCP uses sequence and acknowledgement numbers in packets to detect and report loss, and includes state variables in each protocol entity to keep track of these numbers. State variables on both sides must be synchronised, otherwise DCCP may misinterpret loss information. Thus DCCP needs mechanisms to set up, synchronise and clear state variables in both the Client and Server. We refer to these mechanisms in general as connection management (CM) procedures.

The CM procedures require packets to be exchanged between the Client and Server. RFC 4340 defines 10 different packet types for this purpose: Request, Response, Data, DataAck, Ack, CloseReq, Close, Reset, Sync and SyncAck. Figure 3 is a state diagram illustrating DCCP's connection establishment and release procedures for both the Client and Server. It is derived by combining the state diagrams in RFCs 4340 and 5596, with the dashed parts of the diagram being added by RFC 5596. Ellipses in Fig. 3 represent states while arrows represent state transitions. CLOSED is both an initial and a final state. The inscription on each arrow describes the input and output actions, if any. For instance, the inscription

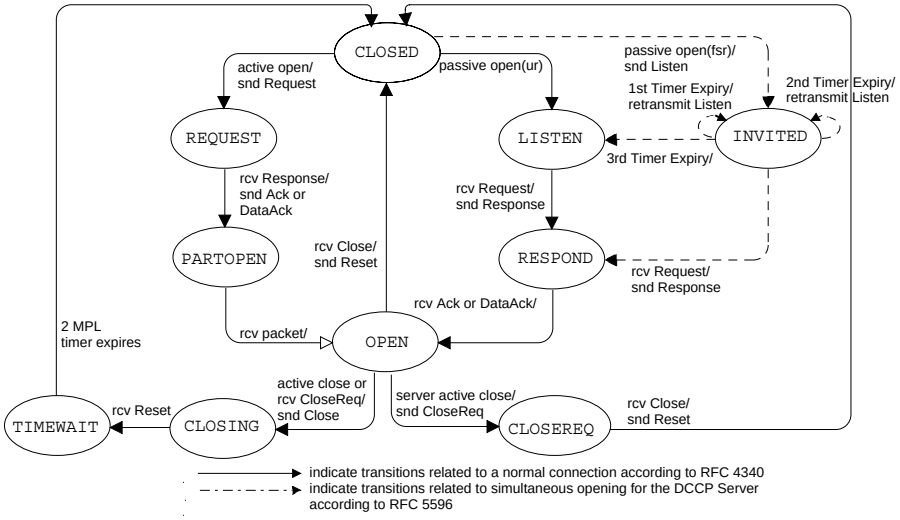


Figure 3 DCCP state diagram.

on the arc from REQUEST to PARTOPEN is "rcv Response snd Ack or DataAck". This means that when the Client receives a DCCP-Response while in the REQUEST state, it returns a DCCP-Ack or DataAck (if it has data to send) and moves to the PARTOPEN state. The Client is identified by an "active open" from its application and passing through the REQUEST state. On the other hand, the Server always receives a "passive open" and passes through the LISTEN state. Applications on both sides can issue an "active close" command but only the Server's application can issue the "server active close" command.

RFC 5596 defines a new packet type called DCCP-Listen and two new states called INVITED and LISTEN1. RFC 5596 differentiates between the cases when the Server connection end point is partially specified (the remote address and port number are unknown) and when it is fully specified. This corresponds to the commands "passive open(ur)" and "passive open(fsr)" respectively, where ur is for 'unspecified remote' and fsr stands for 'fully specified remote'. After receiving a passive open(fsr), a DCCP-Listen packet is sent, a timer is set and the Server transitions from CLOSED to INVITED. If a DCCP-request is not received in time, the DCCP-Listen packet can be retransmitted up to two times before moving to the LISTEN1 state. If the Server receives a DCCP-Request (in INVITED or LISTEN1), it sends a DCCP-Response and transitions to RESPOND. Because the behaviour of DCCPs in the LISTEN and LISTEN1 states are the same, to simplify the state diagram (Fig. 3), we suggest[2] to merge these two states. For more details of these procedures, see [9,18].

2.2 Hole Punching Procedures

The message sequence chart in Fig. 4 provides an example of the hole punching procedure. Prior to connection establishment, we assume that both the Client and Server know each other's public address via a well-known rendezvous server (Fig. 1) using another signalling protocol such as the Session Description Protocol (SDP) [14]. As shown in Fig. 4, when the Client sends the first DCCP-Request packet via NAT-A, NAT-A creates a binding table (a

[2] This was suggested by Professor Jonathan Billington.

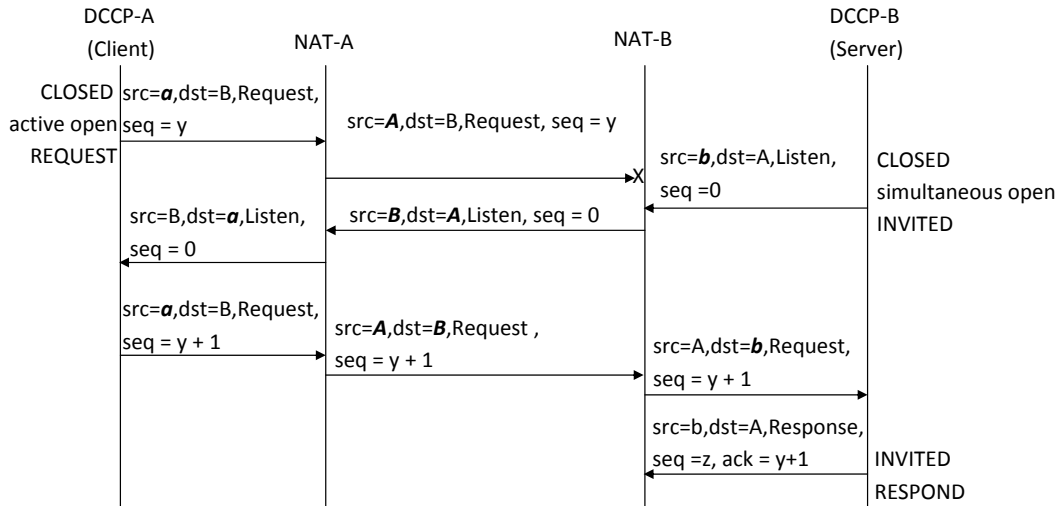


Figure 4 The hole punching procedure.

hole) and replaces the private source address "a" in the DCCP-Request with public source address "A". In this paper, a lower-case letter represents a private address and an upper-case letter represents a public address. The private source address,"a", in every outgoing packet from NAT-A is replaced by the public source address, "A". Similarly, the public destination address "A" in every incoming packet is replaced by the private address "a". However, since no binding for DCCP-B exists in NAT-B, the DCCP-Request packet is blocked by NAT-B, and discarded. In order to allow incoming packets to pass NAT-B and be delivered to DCCP-B, another hole (binding table) is required at NAT-B. As a consequence of prior signalling sessions via the Rendezvous server, DCCP-B sends a DCCP-Listen packet to indicate its willingness to set up a connection with public destination address "A". On receipt of the Listen packet, NAT-B creates a binding table so that the private source address,"b", in every outgoing packet from NAT-B will be replaced by the public source address, "B", for every packet destined for "A". Similarly, the public destination address "B" in every incoming packet from public source address, "A", will be replaced by the private address "b". Because the hole at NAT-A is already punched by the previous DCCP-Request packet, the DCCP-Listen packet can get through NAT-A and arrives at DCCP-A. When DCCP-A, in REQUEST, receives a DCCP-Listen packet, it retransmits the previous DCCP-Request with its sequence number incremented by one. The DCCP-Request is now accepted by NAT-B because "A" has the required entry in its binding table. NAT-B provides the address translation to the private address. The DCCP-request arrives at DCCP-B which sends a DCCP-Response packet and enters the RESPOND state. After that, the connection is established according to the normal connection set up procedure described in RFC 4340. Other scenarios are possible. For example, if the Listen packet is lost, DCCP-A will resend its Request packet after a timeout. Thus it is not essential for Listen packet to be received by DCCP-A, it just provides a speed-up if it gets through before the timeout occurs. It is possible for the DCCP-Listen packet to be sent before the DCCP-Request packet. In this case, the Listen packet will be blocked by NAT-A until it receives the Request packet from DCCP-A.

3 Modelling Approach

3.1 Layer Architecture

Protocols are often organized into a layered structure. Each layer represents a protocol which provides a standard interface to the lower and higher layers. From it own point of view, a specific layer may only observe the interaction at its interface so that the details of the underlying network infrastructure are hidden. Despite the fact that data flows vertically between layers at each end, we can consider that a specific layer horizontally conveys the data between the peer entities at the same layer. Thus each protocol specification at each layer needs to defines only its peer-to-peer behaviour. This peer-to-peer or end-to-end principle[3] abstracts away all lower layers and merges them into an underlying channel. We observe that almost all CPN models of the Internet protocol e.g. [1, 3–5, 12, 13, 19, 20, 22], implicitly use the end-to-end principle and hide all other underlying layers into two channel places. However, there are a few researchers who have investigated multi-layer protocols. For example, [10] modelled and validated connection establishment in the Generic Access Network which involves multiple layers of the protocol stacks. [10] suggested that studying multi-layer protocols provide us insights and understanding how protocol components interact to each other.

As the Internet technology has advanced considerably over recent years, we discover that the end-to-end principle is often violated. For example, the cross-layer design modifies interfaces to higher layers in order to provide performance optimization across layers. NAT is another example that violates the end-to-end principle. Thus NAT can not be abstracted away and its detailed model is required.

3.2 Embedding the NAT Functions in the CPN Models

Two approaches for embedding the underlying channel into a CPN protocol model have been discussed in [6]. The first approach integrates the channel model with the protocol entities. Applying this to our work, the channel model is the NAT functions that are implemented on the output arc inscriptions of the protocol entities. Although this approach helps to reduce the state space size, the model is subtle and tedious. The second approach embeds the channel model or the NAT functions as a module implemented by a substitution transition [15]. This is an elegant way of including NAT devices in the model. This modular approach requires two more substitution transition instances (NAT) and four more buffer places than the integrate approach does. As discussed in [6], from the analysis perspective, this modular approach significantly suffers from state explosion. However for sake of modelling clarity we have selected the modular approach.

4 DCCP Simultaneous Open CPN Model

DCCP simultaneous open CPN models have been developed using both CPN Tools and Design/CPN. Prioritized transitions play an important role in this paper so this section only examines the CPN Tools model. Since our model is extended from [5], this section emphasises on the extension part of the model. For more details of the declarations and the explanation of the previous work, see [5, 25]

[3] "End-to-end principle is an assumption of the Internet property that all nodes can send packets to other nodes of the network, without requiring intermediate network elements to further interpret them."

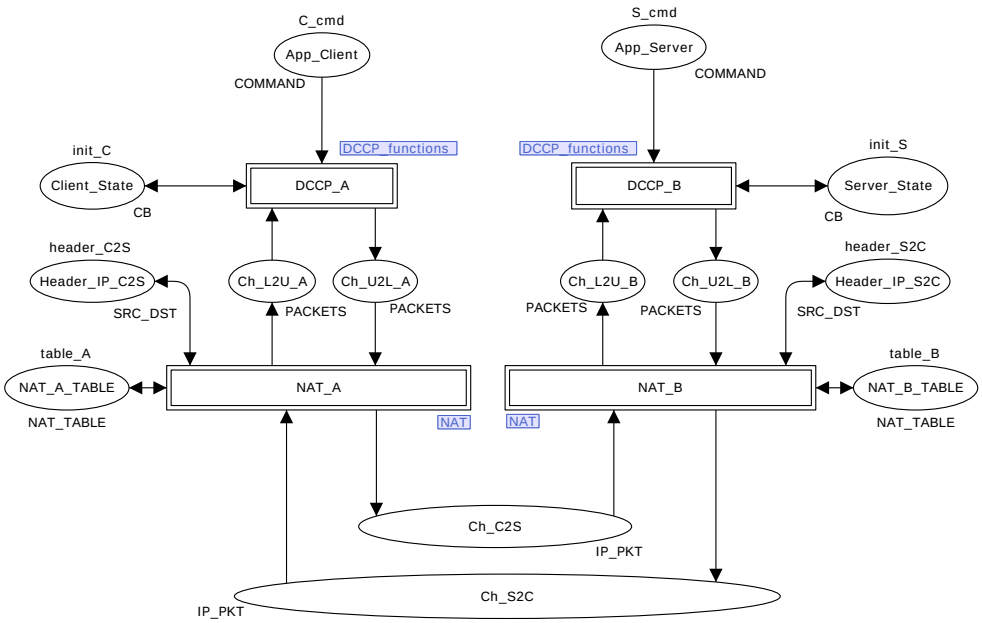


Figure 5 DCCP Top level.

4.1 Model Overview

Our procedure based DCCP-CPN model from Section 4 of [5] has been extended to incorporate the network layer comprising two Network Address Translators (NATs). In spite of the existence of many types of NAT, this paper investigates only "Address and Port-Dependent Mapping[4]". Because the NATs are embedded as a module (substitution transition), another type or combination of different types can be easily integrated in our model. Our procedure based CPN model comprises five hierarchical levels. The complete model comprises 14 places, 68 executable transitions and 25 ML functions.

Figure 5 shows the top level of our CPN model. Two places, App_Client and App_Server, typed by COMMAND (line 15 of Fig. 6), store tokens representing user commands. Substitution transitions, DCCP_A and DCCP_B, represent the DCCP procedures in the Client and the Server, respectively. Substitution transitions, NAT_A and NAT_B, which link to the second level CPN subpage, NAT, models the IP-Port address mapping procedure. Strictly speaking, we do not actually model the hole punching procedure because the hole punching behaviour automatically emerges from interactions among four component in the network: DCCP-A, NAT-A, NAT-B and DCCP-B.

4.2 Declaration of State Variables

DCCP states and variables are stored in Places Client_State and Server_State typed by CB (Control Block). Two new states: LISTEN1 and INVITED are specified by RFC 5596. Figure 6 defines CB (line 10) as the union of four colour sets: IDLE (for CLOSED, LISTEN, LISTEN1 and TIMEWAIT states), RCNT (for INVITED state), RCNTxGSSxISSxlisten_flag

[4] " The NAT reuses the port mapping for subsequent packets sent from the same internal IP address and port to the same external IP address and port" [2]

```
 1: (*         Retransmit Counter          *)
 2: colset RCNT         = int;
 3: colset ACTIVE_STATE = with  RESPOND | PARTOPEN | S_OPEN | C_OPEN
 4:                           | CLOSEREQ | C_CLOSING |S_CLOSING;
 5: colset IDLE         = with CLOSED_I | LISTEN | TIMEWAIT | CLOSED_F | LISTEN1;
 6: colset RCNTxGSSxISSxlisten_flag = product RCNT*SN48*SN48*BOOL;
 7: colset GS                       = record GSS:SN48*GSR:SN48*GAR:SN48;
 8: colset ISN                      = record ISS:SN48*ISR:SN48;
 9: colset ActiveStatexRCNTxGSxISN  = product ACTIVE_STATE*RCNT*GS*ISN;
10: colset CB = union IdleState:IDLE
11:                          + INVITED:RCNT
12:                          + ReqState:RCNTxGSSxISSxlisten_flag
13:                          + ActiveState:ActiveStatexRCNTxGSxISN;
14: (*        User Command        *)
15: colset COMMAND = with simu_Open | p_Open | a_Open | server_a_Close | a_Close;
```

Figure 6 The definition of CB (Control Block) and COMMAND.

(for REQUEST state), and ActiveStatexRCNTxGSxISN (for RESPOND, PARTOPEN, OPEN, CLOSEREQ and CLOSING states). INVITED in the union coloured set CB (line 10) is distinguished from others because this state stores only a retransmission counter. LISTEN1 is declared in the colour set IDLE (line 5). The Client's action, in the REQUEST state, depends whether it has ever received a DCCP-Listen or not. Thus a boolean flag is added in the state variables (line 6).

4.3 Declaration of DCCP and IP Packets

DCCP entities communicate with NATs via buffer places, Ch_L2U_A, Ch_U2L_A, Ch_L2U_B and Ch_U2L_B typed by PACKETS. Two substitution transitions, NAT_A and NAT_B, exchange IP packets via two buffer places, Ch_S2C and Ch_C2S, typed by IP_PKT. Figure 7 declares PACKETS (line 22) as the union of four colour sets: SN48 (for DCCP-Request), SN48 (for DCCP-Listen), SN (for DCCP-Data), Ack_DataAckPacket and OtherPackets. The new packet type defined by RFC 5596 is DCCP-Listen which always has the sequence number equal to zero. The Request, Listen and Data packets are distinguished from the others by ML selectors of the same name as defined in line 22. Figure 7 declares IP_PKT (line 28) as a record of three colour sets: IP (for source address), IP (for designation address) and PACKETS (for DCCP packets). IP are defined as a product of five integers instead of four integers because the port address is also included.

4.4 CPN Subpage NAT

Apart from input and output buffer places, subpage NAT comprises two places and two transitions. Place src_dst typed by SRC_DST stores a record of private source address and public designation address. Transition NAT_TX views the token {src=a, dst=B} together with the token packet forming an incoming IP packet from the private network. Transitions NAT_TX and NAT_RX the priority value, P_HIGH = 100, while P_NORMAL is equal to 1000. Place TABLE typed by NAT_TABLE stores binding tables used for address translations. NAT_TABLE is defined in Fig. 7 (line 26) as a record of three tuples: private source address, public source address and public designation address. When creating a binding table, function put(a) is used to set up the public source address.

4.5 Connection Establishment Pages

This section illustrates two CPN subpages which model connection establishment, the Server and Client pages. Initially, both entities are CLOSED with a simultaneous open command

```
(*       Sequence and Acknowledgement Numbers      *)
colset SN48      = int with 0..MaxSeqNo48;
colset SN24      = int with 0..max_seq_no24;
colset SN48_AN48 = record SEQ:SN48*ACK:SN48;
colset SN24_AN24 = record SEQ:SN24*ACK:SN24;
colset SN        = union longSN:SN48  + shortSN:SN24
colset SN_AN     = union longSA:SN48_AN48 + shortSA:SN24_AN24

(*      Sequence and Acknowledgement Variables     *)
var sn:SN;       var sn48:SN48;           var sn24:SN24;
var sn_an:SN_AN; var sn48_an48:SN48_AN48; var  sn24_an24:SN24_AN24;

(*  Define the DCCP Packet Structure *)
colset Ack_DataAckPktTypes = with Ack | DataAck;
var    ack_dataack:Ack_DataAckPktTypes;

colset OtherPktTypes     = with Sync | SyncAck | Response | CloseReq | Close | Rst;
var p_type:OtherPktTypes;

colset Ack_DataAckPacket = product Ack_DataAckPktTypes*SN_AN;
colset OtherPackets      = product OtherPktTypes*SN48_AN48;
colset PACKETS           = union Request:SN48 + Listen:SN48 +  Data:SN
               + Ack_DataAck:Ack_DataAckPacket + PKT:OtherPacket
(*  Define the IP Packet Structure *)
colset IP        = product INT*INT*INT*INT*INT;
colset NAT_TABLE = record local_src:IP*global_src:IP*global_dst:IP;
colset SRC_DST   = record src:IP*dst:IP;
colset IP_PKT    = record src_add:IP*dst_add:IP*dccp:PACKETS;
var packet:PACKETS;
var a, A, B, gb_src:IP;
```

Figure 7 The definition of DCCP PACKETS and IP_PKT.

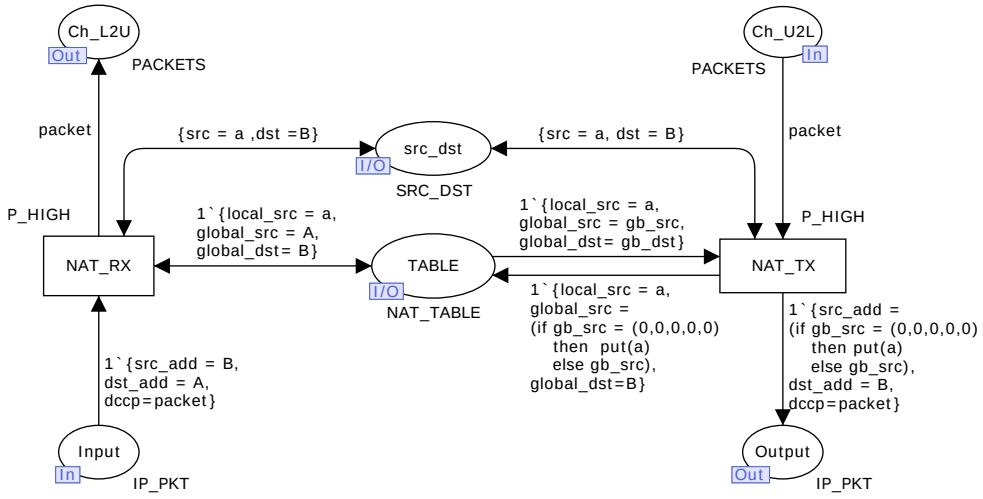


Figure 8 CPN Subpage NAT.

(1‘simu_Open) in Place App_Server and an active open command (1‘a_Open) in Place App_Client.

4.5.1 Server Page

The part of Fig. 9 below App_Server, is the normal connection establishment specified in RFC 4340. The upper part is the standard simultaneous open procedure specified in RFC 5596. With reference to Fig. 3, the occurrence of transition simuOpen (Fig. 9) transmits DCCP-Listen and puts the Server in the INVITED state, waiting for DCCP-Request from the Client.

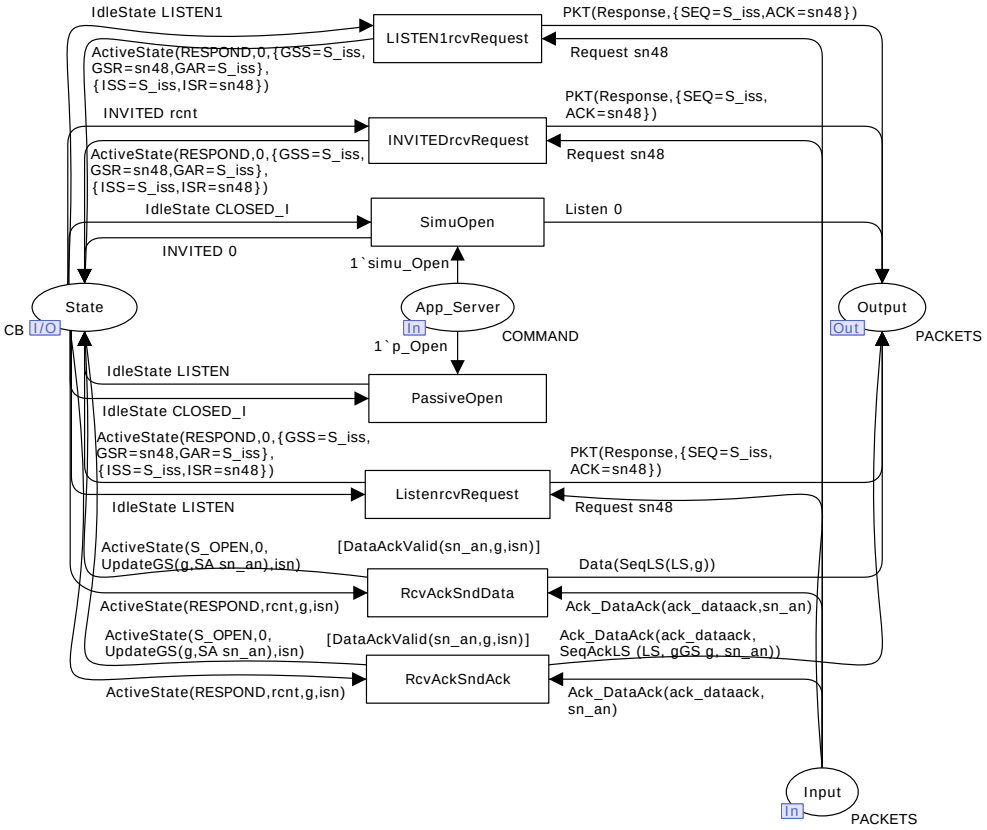


Figure 9 DCCP Server.

After retransmitting twice , the Server enters the LISTEN1 state. These actions are modelled in other CPN subpages: Retransmission and BackOffFails pages. When the Server, in either INVITED, LISTEN1 or LISTEN, receives a DCCP-Request (transition INVITEDrcvRequest, LISTEN1rcvRequest, LISTENrcvRequest) it replies with a DCCP-Response containing the Server's initial sequence number and an acknowledgement for the DCCP-Request. It enters the RESPOND state and appropriately initialises its state variables. These upper three transitions are directly related to the state diagram in Fig. 3.

4.5.2 Client Page

The transition RcvListen in Fig. 10 models actions specified by RFC 5596. On receipt of the DCCP-Listen(seq=0), if the Client has never received DCCP-Listen, it replies with DCCP-Request. If the Client has received DCCP-Listen before, it silently discards the DCCP-Listen.

5 Analysis Approach

A typical approach to alleviate the state explosion problem is to make the number of generated states more compact. We observe that *after writing* the address translation table, NAT in our *specification* model performs only two functions, reordering and forwarding the packets. Intuitively the CPN model of the underlying layer and NAT can be combined and

reduced into two channel places. Thus, the outgoing packet from a DCCP entity is immediately the incoming packet to the other. However the NAT cannot be abstracted away because its behaviour *before writing* the address translation table is different.

By separating the actions before and after writing the address translation table, we suggest that transitions NAT_RX and NAT_TX in the NAT page should get the highest priority. When NAT reorders packets, many sequences of these actions (reorder) lead to the same markings. Analysis using *prioritized transitions* will keep one sequence but discard the rest. Thus, the number of total states is significantly reduced and the safety properties (terminal markings) are preserved.

5.1 CPN Tools versus Design/CPN

Previously, our model [25] was created and maintained using Design/CPN. Because Design/CPN does not support prioritized transitions, we switch to CPN Tools instead. Although using prioritized transitions can reduce the state space significantly, the CPN Tools can generate full state spaces of our model for only a few scenarios. To gain more confidence in the specification RFC 5596, analysis of more scenarios is required. A technique that was successfully used to analyse the DCCP connection management CPN models in [25] is the sweep-line technique. We also wish to apply the sweep-line technique to analyse the DCCP simultaneous open properties. Unfortunately CPN Tools, which support prioritized transitions, do not support sweep-line library. On the other hand, Design/CPN has the sweep-line library but does not support prioritized transitions.

5.2 Prioritized Transitions versus Timed Models

To circumvent the problem in Design/CPN, prioritized transitions are imitated using a timed token enabling all transitions in the DCCP layer. Enabling transitions in NAT layer does not require a timed token. When any transition in the DCCP layer is fired, the time stamp

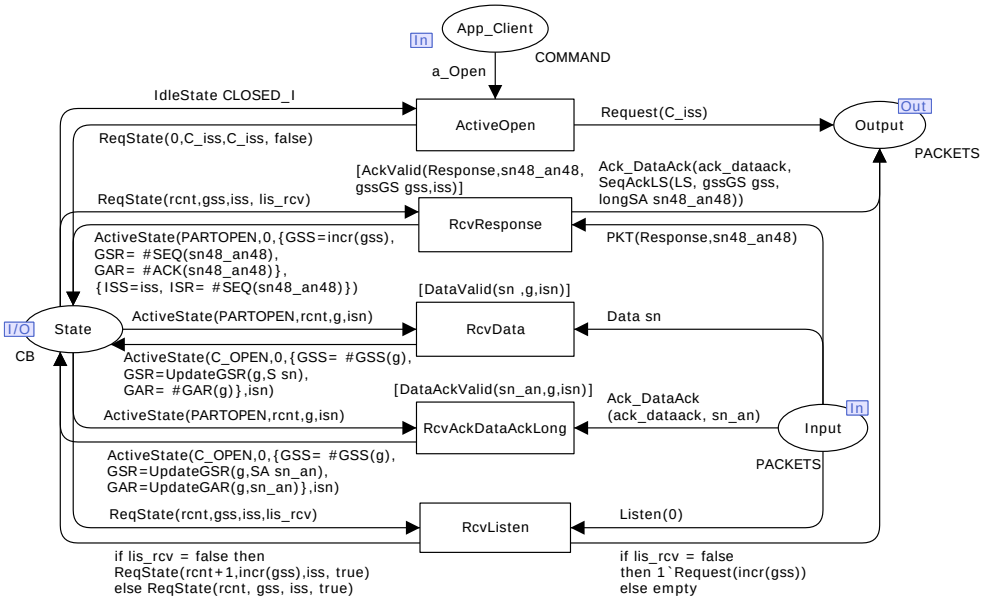


Figure 10 DCCP Client.

```
(*  The Initial State of NAT_A and NAT_B  *)
val header_C2S = 1`{src=(10,0,0,1,4321), dst=(138,76,29,7,31000)};
val header_S2C = 1`{src=(10,1,1,3,4321), dst=(155,99,25,11,62000)};
val table_A=
  1`{global_src=(0,0,0,0,0), global_dst=(0,0,0,0,0), local_src=(10,1,2,3,4322)}
++1`{global_src=(0,0,0,0,0), global_dst=(0,0,0,0,0), local_src=(10,0,9,1,4361)}
++1`{global_src=(0,0,0,0,0), global_dst=(0,0,0,0,0), local_src=(10,0,0,1,4321)};
val table_B=
  1`{global_src=(0,0,0,0,0), global_dst=(0,0,0,0,0), local_src=(10,1,1,3,4321)}
++1`{global_src=(0,0,0,0,0), global_dst=(0,0,0,0,0), local_src=(10,2,9,1,5321)}
++1`{global_src=(0,0,0,0,0), global_dst=(0,0,0,0,0), local_src=(10,0,6,1,4341)};
```

Figure 11 The Initial State of NAT_A and NAT_B.

in the token advances one step. Because the global clock is less than the time stamp by one step, all transitions in NAT layer (if any) have to finish firing before the global clock advances and the transitions in the DCCP layer can be enable. Thus the transitions in the NAT layer have higher firing priority than every transition in the DCCP layer. This imitated method has a drawback that the timed state space is always larger because the global clock and time stamps contribute to the presence of new states. Increasing state space sizes seems to be the wrong path because it encourages state explosion. However [25] demonstrated that if a new additional variable, such as time stamp, is used as progress measure for the sweep-line analysis, in spite of a larger state space size, the peak memory used and exploration time can be significantly reduced. Finally we analyse the augmented model similar to the Sweep-line analysis in [25]. The experimental results are discussed in section 6.2.

6 Experimental Results

This section contains analysis results for the DCCP simultaneous open procedures when operating over *reordering channels without loss*. In contrast to the previous work that considers various cases according the combination of user commands. This paper investigates only the simultaneous open scenario when the Client user issues an "active open" and the Server user issues a "simultaneous open" command. The initial markings of all buffer and channel places are empty. The initial state of both side are CLOSED and the initial send sequence number (ISS) on both sides is set to 10. The initial markings in Places Header_IP_C2S, Header_IP_S2C, NAT_A_TABLE and NAT_B_TABLE are specified in Fig. 11. Without loss of generality, only long sequence numbers are used. All experiments are conducted on a AMD 9650 2.31GHz PC with 4 GByte RAM. CPN Tools runs on Window XP while Design/CPN runs on Fedora Core version 6.

6.1 The Prioritized Transition Model

Table 1 illustrates the experimental results when we use prioritized transitions and analyse the model by CPN Tools. The first column (*Config.*) in this table defines the configuration being analysed, where the 3-tuple represents the maximum number of retransmissions allowed for Request, Listen and Ack packets respectively. Columns *total nodes* and *total arcs* record the total number of markings and arcs in the state space, respectively. The time (hours:minutes:seconds) to generate the full state space is given in Column *time*. The next two columns (*DMs*) records the number of dead markings. Dead markings are classified into type I and type II. Type I dead markings are desirable and correspond to successful connection establishment where both the Client and Server are in the OPEN state. In Type II dead markings both the Client and Server are in still CLOSED state. Both types are

Table 1 DCCP simultaneous open using Prioritized Transitions.

Config.	total nodes	total arcs	time	DMs I	DMs II	Bounds Ch L2U_B	Bounds Ch L2U_A
(0,0,0)	16,441	28,308	00:00:43	13	1	3	5
(0,0,1)	78,360	141,749	00:10:36	33	1	4	5
(0,1,0)	24,579	43,612	00:01:23	13	1	3	6
(0,1,1)	117,264	217,964	00:22:00	33	1	4	6
(0,2,0)	32,736	58,952	00:01:58	13	1	3	7
(0,2,1)	156,187	294,215	00:37:47	33	1	4	7

Table 2 DCCP simultaneous open using the sweep-line method with the augmented model.

Config.	Sweep-line with the augmented model: total nodes	total arcs	peak nodes	time	DMs I	DMs II	Bounds Ch L2U_B	Bounds Ch L2U_A	% space
(0,0,0)	40,984	65,463	288	00:00:29	26	14	3	5	1.75
(0,0,1)	279,581	469,298	1,080	00:02:33	66	19	4	5	1.38
(0,1,0)	81,531	135,246	496	00:00:50	39	16	3	6	2.02
(0,1,1)	557,615	967,911	2,059	00:07:34	99	21	4	6	1.76
(1,0,0)	2,896,471	4,921,848	3,142	00:38:27	148	24	4	6	-
(1,0,1)	34,412,454	60,468,592	17,908	09:29:13	360	30	5	6	-
(1,1,0)	5,770,971	10,105,648	5,810	01:22:53	222	26	4	7	-
(1,1,1)	68,581,787	123,703,372	34,892	20:57:25	540	32	5	7	-
(0,2,0)	135,454	229,717	794	00:01:14	52	18	3	7	2.43
(0,2,1)	927,819	1,642,398	3,347	00:09:04	132	23	4	7	2.14
(0,2,2)	6,719,017	12,943,167	16,034	00:01:51	236	29	5	8	-
(1,2,0)	9,596,365	17,103,716	9,486	01:42:07	296	28	4	8	-
(1,2,1)	114,060,085	208,918,444	57,427	36:58:00	720	34	5	8	-

expected dead markings. All dead markings have no packets left in all buffers and channels. The last two columns, *Bounds*, record the maximum number of packets that can occur in the channel places Ch_L2U_B and Ch_L2U_A.

6.2 Analyses the Timed Model using the Sweep-line Method

Using prioritized transitions reduces the state space sizes significantly but we can analyse only six scenarios. When we attempt to analyse the scenarios (1,0,0) , (0,2,2) and (0,1,2), the available memory is exhausted. As discuss in Section 5, we turn to the sweep-line technique (with the augmented model). Table 2 illustrates the experimental results when the sweep-line is applied to the timed CPN model. We use the progress vector suggested in Section 4.5 of [25] together with the time stamp. Conducting search experiments, we discover that the best position of the time stamp in the progress vector is at the end of the list.

Column *peak nodes* in Table 2 lists the peak number of nodes stored in main memory at any one time. Column *time* records the time used to explore the state space. The last column (*% space*) of Table 2 shows the ratio of the number of peak states compared to the total number of states in Table 1. The smaller the number, the more efficient the sweep-line algorithm is. The number of peak states is reduced to only 1–2% of the full untimed state space. This analysis method has potential to explore more scenarios.

7 Conclusions and Future Work

This paper has presented a Coloured Petri Nets model and analysis of DCCP simultaneous open procedure. Our CPN model is developed based on both RFC 4340 and RFC 5596.

Because NATs with the hole punching procedure affect DCCP behaviour, they cannot be simply abstracted away using the layered architecture. We suggest to separate NAT operations into before and after writing the address translation table and remove some transition occurrences using prioritized transitions. It is possible to use the timed model to imitate prioritized transitions. Analysing the timed models using Sweep-line method is more efficient than generating full state space of the prioritized transitions models

In future, we are interested in modelling different types of NATs, and increasing the number of protocol entities. Instead of studying functional behaviour, we wish to investigate performance behaviour of each protocol entity as well.

Acknowledgments. This work is supported by Research Grant from the Thai Network Information Center Foundation and the Thailand Research Fund. The author is thankful to Professor Jonathan Billington, Professor Lar M. Kristensen and the anonymous reviewers. Their constructive comments have helped to improve the quality of this paper.

References

1 *Application of Petri Nets to Communication Networks*, volume 1605 of *Lecture Notes in Computer Science*. Springer, Heidelberg, 1999.

2 F. Audet and C. Jennings. Network Address Translation (NAT) Behavioral Requirements for UNicast UDP RTP: A Transport Protocol for Real-Time Applications, RFC 4787. Available via http://www.rfc-editor.org/rfc/rfc4787.txt, January 2007.

3 J. Billington, G. E. Gallasch, and B. Han. A Coloured Petri Net Approach to Protocol Verification. In *Lectures on Concurrency and Petri Nets, Advances in Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 210–290. Springer, Heidelberg, 2004.

4 J. Billington and B. Han. Modelling and Analysing the Functional Behaviour of TCP's Connection Management Procedures. *International Journal on Software Tools for Technology Transfer*, 9(3-4):269–304, June 2007. Available via http://dx.doi.org/10.1007/s10009-007-0034-1.

5 J. Billington and S. Vanit-Anunchai. Coloured Petri Net Modelling of an Evolving Internet Standard: the Datagram Congestion Control Protocol. *Fundamenta Informaticae*, 88(3):357–385, 2008.

6 J. Billington, S. Vanit-Anunchai, and G. E. Gallasch. Parameterised Coloured Petri Nets Channel Models. In *Transactions on Petri Nets and Other Models of Concurrency*, volume 5800 of *Lecture Notes in Computer Science*, pages 71–97. Springer, Heidelberg, 2009.

7 S. Christensen, L.M. Kristensen, and T. Mailund. A Sweep-Line Method for State Space Exploration. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *Lecture Notes in Computer Science*, pages 450–464, Genova, Italy, 2-6 April 2001. Springer, Heidelberg.

8 Design/CPN Online. http://www.daimi.au.dk/designCPN/.

9 G. Fairhurst. Datagram Congestion Control Protocol (DCCP) Simultaneous-Open Technique to FacilitateNAT/Middlebox Traversal, RFC 5596. Available via http://www.rfc-editor.org/rfc/rfc5596.txt, September 2009.

10 P. Fleischer and L. M. Kristensen. Formal Specification and Validation of Secure Connnection Establishment in a Generic Access Network Scenario. In *Proceedings of ICATPN'08*, volume 5062 of *Lecture Notes in Computer Science*, pages 153–171. Springer, Heidelberg, 2008.

11 S. Floyd, M. Handley, and E. Kohler. Problem Statement for the Datagram Congestion Control Protocol (DCCP), RFC 4336. Available via http://www.rfc-editor.org/rfc/rfc4336.txt, March 2006.

12 S. Gordon. *Verification of the WAP Transaction Layer uisng Coloured Petri Nets*. PhD thesis, Institute for Telecommunications Research and Computer Systems Engineering Centre, School of Electrical and Information Engineering, University of South Australia, Adelaide, Australia, November 2001.

13 B. Han. *Formal Specification of the TCP Service and Verification of TCP Connection Management*. PhD thesis, Computer Systems Engineering Centre, School of Electrical and Information Engineering, University of South Australia, Adelaide, Australia, December 2004.

14 M. Handley, V. Jacobson, and C. Perkins. SDP: Session Description Protocol, RFC 4566. Available via http://www.rfc-editor.org/rfc/rfc4566.txt, July 2006.

15 K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Vol. 1, Basic Concepts*. Monographs in Theoretical Computer Science. Springer, Heidelberg, 2nd edition, 1997.

16 K. Jensen and L.M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer, Heidelberg, 2009.

17 E. Kohler, M. Handley, and S. Floyd. Designing DCCP: Congestion Control Without Reliability. In *Proceedings of the 2006 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'06)*, pages 27–38, Pisa, Italy, 11-15 September 2006.

18 E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol, RFC 4340. Available via http://www.rfc-editor.org/rfc/rfc4340.txt, March 2006.

19 L. M. Kristensen and K. Jensen. Specification and Validation of an Edge Router Discovery Protocol for Mobile Ad Hoc Networks. In *Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*, pages 248–269. Springer, Heidelberg, 2004.

20 L. Liu. *Towards Parametric Verification of the Capability Exchange Signalling Protocol*. PhD thesis, Computer Systems Engineering Centre, School of Electrical and Information Engineering, University of South Australia, Adelaide, Australia, May 2006.

21 T. Mailund. *Sweeping the State Space - A Sweep-Line State Space Exploration Method*. PhD thesis, Department of Computer Science, University of Aarhus, February 2003.

22 C. Ouyang. *Formal Specification and Verification of the Internet Open Trading Protocol using Coloured Petri Nets*. PhD thesis, Computer Systems Engineering Centre, School of Electrical and Information Engineering, University of South Australia, Adelaide, Australia, June 2004.

23 S. Vanit-Anunchai. *An Investigation of the Datagram Congestion Control Protocol's Connection Management and Synchronisation Procedures*. PhD thesis, Computer Systems Engineering Centre, School of Electrical and Information Engineering, University of South Australia, Adelaide, Australia, November 2007.

24 S. Vanit-Anunchai and J. Billington. Modelling the Datagram Congestion Control Protocol's Connection Management and Synchronisation Procedures. In *Proceedings of the 28th International Conference on Application and Theory of Petri Nets and other models of concurrency (ICATPN'07)*, volume 4546 of *Lecture Notes in Computer Science*, pages 423–444, Siedlce, Poland, 25-29 June 2007. Springer, Heidelberg.

25 S. Vanit-Anunchai, J. Billington, and G.E. Gallasch. Analysis of the Datagram Congestion Control Protocol's Connection Management Procedures using the Sweep-line Method. *International Journal on Software Tools for Technology Transfer*, 10(1):29–56, 2008. Available via http://dx.doi.org/10.1007/s10009-007-0050-1.

Dynamic Clock Elimination in Parametric Timed Automata

Étienne André

Université Paris 13, Sorbonne Paris Cité, LIPN, CNRS, UMR 7030
93430 Villetaneuse, France
Etienne.Andre@univ-paris13.fr

Abstract

The formalism of parametric timed automata provides designers with a formal way to specify and verify real-time concurrent systems where timing requirements are unknown (or parameters). Such models are usually subject to the state space explosion. A popular way to partially reduce the size of the state space is to reduce the number of clock variables. In this work, we present a technique for dynamically eliminating clocks. Experiments using IMITATOR show a diminution of the number of states and of the computation time, and in some cases allow termination of the analysis of models that could not terminate otherwise. More surprisingly, even when the number of clocks remains constant, there is little noticeable overhead in applying the proposed clock elimination.

1998 ACM Subject Classification D.4.7 Real-time systems and embedded systems

Keywords and phrases Verification, Real-time systems, Parameter synthesis, State space reduction, Inverse Method

Digital Object Identifier 10.4230/OASIcs.FSFMA.2013.18

1 Introduction

Ensuring the correctness of critical real-time systems, involving concurrent behaviors and timing requirements, is crucial. Formal verification methods may not always be able to verify full size systems, but they provide designers with an important help during the design phase, in order to detect otherwise costly errors. Timed automata (TA) are an extension of finite state automata with clocks, i.e., real-valued variables that are compared with constants in guards and invariants, and may be reset along transitions. TA have been extensively used in the past decades, and led to useful and efficient implementations.

Parameter synthesis for real-time systems is a set of techniques aiming at synthesizing dense sets of valuations for the timing requirements of a system. It consists in considering the delays as unknown constants, or *parameters*, and synthesizing constraints on these parameters guaranteeing the system correctness. Parameterizing TA gives parametric timed automata (PTA) [4].

A fundamental problem in the exploration of the reachability space in PTA is to compact as much as possible the generated space of symbolic states. We propose here a state space reduction based on clock elimination.

Related Work

It is well known that the fewer clocks, the more efficient real-time model checking is [11]. Furthermore, a smaller number of clocks may imply a more compact state space: when constraints are represented using arrays and matrices, the fewer clocks, the smaller the

1st French Singaporean Workshop on Formal Methods and Applications 2013 (FSFMA'13).
Editors: Christine Choppy and Jun Sun; pp. 18–31

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

constraints are, the more compact the state space is. Formalisms such as (parametric) timed Petri nets [24] or stateful timed CSP [22] have the advantage to dynamically create and discard clocks (or firing times in Petri nets). Hence, clocks only appear in symbolic states when they are actually useful. In contrast, in (parametric) timed automata, according to their standard semantics, clocks must be present in all states.

Still, several works have been proposed to reduce the state space based on the clocks. A well known approach in timed automata is to abstract the value of the clocks as soon as they become larger than the system's largest constant. This technique is implemented in most tools for TA such as UPPAAL [19]; unfortunately, this approach does not apply to PTA, where the constants are replaced with parameters. In [15], two methods are proposed to reduce the number of clocks: (1) the detection of active clocks (the other clocks can be safely eliminated) and (2) the detection of clocks equal to each others (in which case only one such clock can be kept). It is shown that the resulting automaton is bisimilar to the original one, and experiments show large state space reductions. Our work is close to the first method, but extended to the parametric case. Furthermore, the constraints are implemented in [15] in the form of difference bound matrices, where adding and removing clocks is straightforward. In contrast, we use polyhedra where such operations are much more costly; however, experiments show that the overhead in the worst case is still very limited in our setting. Finally, our original motivation was to ensure termination of some systems, which is not necessary in the non-parametric setting since most algorithms rely on symbolic state space partitions guaranteeing termination.

More recently, an approach has been proposed in [10] to avoid the use of global clocks in networks of timed automata, to be analyzed in a distributed setting. Although that approach does not reduce the number of clocks (in contrast to ours), it simplifies the model since less synchronization is needed between the different TA in parallel.

Finally, our work is partially inspired by the parametric extension of stateful timed CSP (PSTCSP) [7]. In PSTCSP, clocks are dynamically created, and discarded when no longer used. Whereas this clock elimination natively belongs to the semantics of PSTCSP, and hence does not require any additional computation, we have to propose algorithms to be able to dynamically eliminate clocks in PTA.

Contribution

We introduce here a technique to eliminate clocks on-the-fly, when it is guaranteed that they will not be read in guards and invariants until their next reset. Our approach is based on a static computation of the location where clocks can be safely eliminated, as well as on a dynamic elimination of these clocks during the analysis.

We implemented our approach in IMITATOR [5], a tool for the synthesis of timing parameters in which operations on constraints rely on the Parma Polyhedra Library [9]. Experiments show a diminution of the number of states and of the computation time, and in some cases allow termination of the analysis of models that could not terminate otherwise. Surprisingly, even when the number of clocks (and hence of states) remains constant, the computation time does not increase, i.e., there is little noticeable overhead in applying the proposed clock elimination.

Outline

We recall preliminaries in Section 2. We define and characterize our dynamic clock elimination technique in Section 3. We present experiments using IMITATOR in Section 4 and conclude in Section 5.

2 Preliminaries

We denote by $\mathbb{N}$, $\mathbb{Q}_+$ and $\mathbb{R}_+$ the sets of non-negative integers, non-negative rational and non-negative real numbers, respectively.

2.1 Clocks, Parameters and Constraints

Throughout this paper, we assume a fixed set $X = \{x_1, \ldots, x_H\}$ of *clocks*. A *clock* is a variable x_i with value in $\mathbb{R}_+$. All clocks evolve linearly at the same rate. A *clock valuation* is a function $w : X \to \mathbb{R}_+^H$ assigning a non-negative real value to each clock variable. We will often identify a valuation w with the point $(w(x_1), \ldots, w(x_H))$. Given a constant $d \in \mathbb{R}_+$, we use $X + d$ to denote the set $\{x_1 + d, \ldots, x_H + d\}$. Similarly, we write $w + d$ to denote the valuation such that $(w + d)(x) = w(x) + d$ for all $x \in X$.

Throughout this paper, we assume a fixed set $P = \{p_1, \ldots, p_M\}$ of *parameters*, i.e., unknown constants. A *parameter valuation* π is a function $\pi : P \to \mathbb{R}_+^M$ assigning a nonnegative real value to each parameter. There is a one-to-one correspondence between valuations and points in $(\mathbb{R}_+)^M$. We will often identify a valuation π with the point $(\pi(p_1), \ldots, \pi(p_M))$.

We define here constraints as a set of linear inequalities. An *inequality* over X and P is $e \prec e'$, where $\prec \in \{<, \leq\}$, and e, e' are two linear terms of the form

$$\sum_{1 \leq i \leq N} \alpha_i z_i + d$$

where $z_i \in X \cup P$, $\alpha_i \in \mathbb{Q}_+$, for $1 \leq i \leq N$, and $d \in \mathbb{Q}_+$. We define in a similar manner inequalities over X (resp. P). A *constraint* is a conjunction of inequalities.

We denote by $\mathcal{L}(X)$, $\mathcal{L}(P)$ and $\mathcal{L}(X \cup P)$ the set of all constraints over X, over P, and over X and P respectively. In the sequel, the letter $D \in \mathcal{L}(X)$ denotes a constraint over the clocks, the letter $K \in \mathcal{L}(P)$ denotes a constraint over the parameters, and the letter $C \in \mathcal{L}(X \cup P)$ denotes a constraint over the clocks and the parameters.

Given a clock valuation w, $D[w]$ denotes the expression obtained by replacing each clock x in D with $w(x)$. A clock valuation w *satisfies* constraint D (denoted by $w \models D$) if $D[w]$ evaluates to true.

Given a parameter valuation π, $C[\pi]$ denotes the constraint over the clocks obtained by replacing each parameter p in C with $\pi(p)$. Likewise, given a clock valuation w, $C[\pi][w]$ denotes the expression obtained by replacing each clock x in $C[\pi]$ with $w(x)$. We say that a parameter valuation π *satisfies* a constraint C, denoted by $\pi \models C$, if the set of clock valuations that satisfy $C[\pi]$ is nonempty. We use the notation $<w, \pi> \models C$ to indicate that $C[\pi][w]$ evaluates to true. Given a constraint C and a clock x, we write $x \in C$ to denote that x is not a free variable in C.

Given two constraints C_1 and C_2 over the clocks and the parameters, C_1 is said to be *included in* C_2, denoted by $C_1 \subseteq C_2$, if $\forall w, \pi : <w, \pi> \models C_1 \implies <w, \pi> \models C_2$.

We denote by $C\backslash_X$ the constraint over the parameters obtained by eliminating its clock variables (e.g., using Fourier-Motzkin [21]). Similarly, we denote by $C{\downarrow_P}$ the constraint over the parameters obtained by projecting C onto the set of parameters, that is after elimination of the clock variables. Formally, $C{\downarrow_P} = \{\pi \mid \exists w : <w, \pi>\}$. Note that $C\backslash_X = C{\downarrow_P}$.

Sometimes we will refer to a variable domain X', which is obtained by renaming the variables in X. Explicit renaming of variables is denoted by the substitution operation. Given a constraint C over the clocks and the parameters, we denote by $C_{[X \leftarrow X']}$ the constraint

obtained by replacing in C the variables of X with the variables of X'. We sometime write $C(X)$ or $C(X')$ to denote the set of clocks used within C.

We define the *time elapsing* of C, denoted by $C^{\uparrow}$, as the constraint over X and P obtained from C by delaying an arbitrary amount of time. Formally:

$$C^{\uparrow} = \Big((C \wedge X' = X + d)\backslash_{X \cup \{d\}}\Big)_{[X' \leftarrow X]}$$

where d is a new parameter with values in $\mathbb{R}_+$, and X' is a renamed set of clocks. The inner part of the expression adds the same delay d to all clocks; then the original set of clocks X and d are eliminated; the outer part of the expression renames clocks X' with X.

2.2 Labeled Transition Systems

We introduce below labeled transition systems, which will be used later in this section to represent the semantics of parametric timed automata.

▶ **Definition 1.** A *labeled transition system* is a quadruple $\mathcal{LTS} = (\Sigma, S, S_0, \Rightarrow)$, with Σ a set of symbols, S a set of *states*, $S_0 \subset S$ a set of *initial states*, and $\Rightarrow \in S \times \Sigma \times S$ a *transition relation*. We write $s \overset{a}{\Rightarrow} s'$ for $(s, a, s') \in \Rightarrow$. A *run* (of length m) of $\mathcal{LTS}$ is a finite alternating sequence of states $s_i \in S$ and symbols $a_i \in \Sigma$ of the form $s_0 \overset{a_0}{\Rightarrow} s_1 \overset{a_1}{\Rightarrow} \cdots \overset{a_{m-1}}{\Rightarrow} s_m$, where $s_0 \in S_0$. A state s_i is *reachable* if it belongs to some run r.

2.3 Parametric Timed Automata

Parametric timed automata are an extension of the class of timed automata [3] to the parametric case, where parameters can be used within guards and invariants in place of constants [4].

Syntax

▶ **Definition 2** (Parametric Timed Automaton)**.** A *parametric timed automaton (PTA)* $\mathcal{A}$ is a 8-tuple of the form $\mathcal{A} = (\Sigma, L, l_0, X, P, K, I, \rightarrow)$, where

- Σ is a finite set of actions,
- L is a finite set of locations, $l_0 \in L$ is the initial location,
- X is a set of clocks, P is a set of parameters, $K \in \mathcal{L}(P)$ is the initial constraint,
- I is the invariant, assigning to every $l \in L$ a constraint $I(l) \in \mathcal{L}(X \cup P)$, and
- $\rightarrow$ is a step relation consisting of elements of the form (l, g, a, ρ, l'), where $l, l' \in L$ are the source and destination locations, $a \in \Sigma$, $\rho \subseteq X$ is a set of clocks to be reset by the step, and $g \in \mathcal{L}(X \cup P)$ is the step guard.

The constraint K corresponds to the *initial* constraint over the parameters, i.e., a constraint that will be true in all the states of $\mathcal{A}$ (see semantics in Definition 4). For example, in a PTA with two parameters *min* and *max*, one may want to constrain *min* to be always smaller or equal to *max*, in which case K is defined to be $min \leq max$.

Semantics

The (symbolic) semantics of PTA relies on the following notion of state.

▶ **Definition 3** (State)**.** Let $\mathcal{A} = (\Sigma, L, l_0, X, P, K, I, \rightarrow)$ be a PTA. A *state* s of $\mathcal{A}$ is a pair (l, C) where $l \in L$ is a location, and $C \in \mathcal{L}(X \cup P)$ its associated constraint.

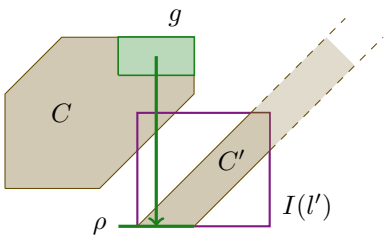


Figure 1 Forward reachability for timed automata.

For each valuation π of P, we may view a state s as the set of pairs (l, w) where w is a clock valuation such that $<w, \pi> \models C$.

The *initial state* of $\mathcal{A}$ is $s_0 = (l_0, C_0)$, where $C_0 = K \wedge I(l_0) \wedge \bigwedge_{i=1}^{H-1} x_i = x_{i+1}$. In this expression, K is the initial constraint over the parameters, $I(l_0)$ is the invariant of the initial location, and the rest of the expression lets clocks evolve from the same initial value.

The semantics of PTA is given in the following in the form of an LTS.

▶ **Definition 4** (Semantics of PTA). Let $\mathcal{A} = (\Sigma, L, l_0, X, P, K, I, \rightarrow)$ be a PTA. The *semantics of* $\mathcal{A}$ is $\mathcal{LTS}(\mathcal{A}) = (\Sigma, S, S_0, \Rightarrow)$ where

$$S = \{(l, C) \in L \times \mathcal{L}(X \cup P) \mid C \subseteq I(l)\},$$
$$S_0 = \{(l_0, K \wedge I(l_0) \wedge \textstyle\bigwedge_{i=1}^{H-1} x_i = x_{i+1})\}$$

and a transition $(l, C) \overset{a}{\Rightarrow} (l', C')$ belongs to $\Rightarrow$ if $\exists C'' : (l, C) \overset{a}{\rightarrow} (l', C'') \overset{d}{\rightarrow} (l', C')$, with

- discrete transitions $(l, C) \overset{a}{\rightarrow} (l', C')$ if there exists $(l, g, a, \rho, l') \in \rightarrow$ and

$$C' = \Big(\big(C(X) \wedge g(X) \wedge X' = \rho(X)\big)\backslash_X \wedge I(l')(X')\Big)_{[X' \leftarrow X]} \text{ and}$$

- delay transitions $(l, C) \overset{d}{\rightarrow} (l, C')$ with $C' = C^{\uparrow} \wedge I(l)(X)$.

In Figure 1, we present in a graphical way the computation of the successor constraint of a state (l, C). First, C is intersected with the guard g of the transition. Then, the clocks that must be reset by the transition (as in ρ) are projected onto zero. Then, the constraint is intersected with the invariant of the destination location $I(l')$. Time elapsing is then applied. The resulting constraint C' is finally obtained by intersecting again with the invariant of the destination location $I(l')$.

Let $\mathcal{LTS}(\mathcal{A}) = (\Sigma, S, S_0, \Rightarrow)$. When clear from the context, given $(s_1, a, s_2) \in \Rightarrow$, we write $(s_1 \overset{a}{\Rightarrow} s_2) \in \Rightarrow(\mathcal{A})$; and we write s_0 for the (only) state in S_0.

A *path* of $\mathcal{A}$ is a finite alternating sequence of states and actions.

▶ **Definition 5** (Path). Let $\mathcal{A}$ be a PTA. Let $s_0 \overset{a_0}{\Rightarrow} \dots \overset{a_{n-1}}{\Rightarrow} s_n$, such that $s_i \overset{a_i}{\Rightarrow} s_{i+1} \in \Rightarrow(\mathcal{A})$, for all $0 \leq i \leq n-1$.

Then $s_0 \overset{a_0}{\Rightarrow} \dots \overset{a_{n-1}}{\Rightarrow} s_n$ is said to be a *path* of $\mathcal{A}$. The set of all paths of $\mathcal{A}$ is denoted by $Paths(\mathcal{A})$.

We define *traces* as time-abstract paths.

▶ **Definition 6** (Trace). Given a path $(l_0, C_0) \overset{a_0}{\Rightarrow} (l_1, C_1) \overset{a_1}{\Rightarrow} \cdots \overset{a_{m-1}}{\Rightarrow} (l_m, C_m)$, the corresponding trace is $l_0 \overset{a_0}{\Rightarrow} l_1 \overset{a_1}{\Rightarrow} \cdots \overset{a_{m-1}}{\Rightarrow} l_m$.

Finally, we recall the parallel composition of PTA: N PTA can be composed into a single parametric timed automaton, by performing a product of the N PTA.

▶ **Definition 7.** Let $N \in \mathbb{N}$. For all $1 \leq i \leq N$, let $\mathcal{A}_i = (\Sigma_i, L_i, (l_0)_i, X_i, P_i, K_i, I_i, \rightarrow_i)$ be a PTA. The sets L_i are mutually disjoint. A *network of PTA* is $\mathcal{A} = \mathcal{A}_1 \| \dots \| \mathcal{A}_N$, where $\|$ is the operator for parallel composition defined in the following way. This network of PTA corresponds to the PTA $\mathcal{A} = (\Sigma, L, l_0, X, P, K, I, \rightarrow)$ where

- $\Sigma = \bigcup_{i=1}^{N} \Sigma_i$, $L = \Pi_{i=1}^{N} L_i$, $l_0 = \langle (l_0)_1, \dots, (l_0)_N \rangle$,
- $X = \bigcup_{i=1}^{N} X_i$, $P = \bigcup_{i=1}^{N} P_i$, $K = \bigwedge_{i=1}^{N} K_i$,
- $I(\langle l_1, \dots, l_N \rangle) = \bigwedge_{i=1}^{N} I_i(l_i)$ for all $\langle l_1, \dots, l_N \rangle \in L$,

and $\rightarrow$ is defined as follows. For all $a \in \Sigma$, let T_a be the subset of indices $i \in 1, \dots, N$ such that $a \in \Sigma_i$. For all $a \in \Sigma$, for all $\langle l_1, \dots, l_N \rangle \in L$, for all $\langle l'_1, \dots, l'_N \rangle \in L$, we have that $(\langle l_1, \dots, l_N \rangle, g, a, \rho, \langle l'_1, \dots, l'_N \rangle) \in \rightarrow$ if:

- for all $i \in T_a$, there exist g_i, ρ_i such that $(l_i, g_i, a, \rho_i, l'_i) \in \rightarrow_i$, $g = \bigwedge_{i \in T_a} g_i$, $\rho = \bigcup_{i \in T_a} \rho_i$, and,
- for all $i \notin T_a$, $l'_i = l_i$.

3 On-the-fly Clock Elimination

3.1 Motivation

Consider the PTA depicted in Figure 2. This PTA contains 2 locations, 2 clocks x_1 and x_2, as well as 2 parameters p_1 and p_2. Although the clock x_2 is not used in l_2, its existence will generate an infinite set of states. More precisely, an infinite number of states with a constraint of the form $x_2 = x_1 + i \times p_1$ (with i infinitely growing) will be generated.

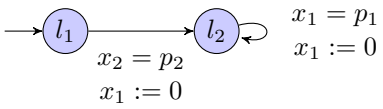


Figure 2 A looping automaton.

This situation is not met in the non-parametric setting. Indeed, it is well known that, once the value of a clock gets larger than the system's largest constant c, this clock value can be safely abstracted to an abstract value "greater than c". Unfortunately, this is not possible in the parametric setting, due to the fact that constants are unknown.

Here, we propose a simple technique based on dynamic clock elimination. We can note that x_2 is "useless" in l_2: indeed, it is not read in any guard, nor reset, and, since l_2 has no successor location except itself, x_2 will not be read in the future. As a consequence, x_2 can be safely discarded or *eliminated* in l_2, so as to ensure termination of the analysis.

Recall that this situation is not met in formalisms such as the parametric extension of stateful timed CSP [7]. Indeed, in this formalism, clocks are dynamically created, and discarded when no longer used.

3.2 General Approach

We propose here to eliminate useless clocks on-the-fly, i.e., during the analysis. By useless, we mean clocks that will not be useful in the future (i.e., not read in guards and invariants), until their next reset. Technically, detecting useless clocks would require to explore the

system, and check whether a given clock will be used (i.e., read in a guard or in an invariant) in the future. Unfortunately, this would not be interesting to do in practice since this would require to analyze the whole system, which we want to avoid. Hence, one must accept to possibly exhibit an under-approximation of the set of useless clocks, in order to find a trade-off between efficiency and accuracy.

In this work, we propose the following technique. First, we detect the useless clocks in a static manner; hence, we construct prior to the analysis a table associating each location with the list of the clocks useless in this location. During the analysis, it is sufficient to check this table in order to know which clocks are useless.

Second, we consider only *local* clocks, i.e., used in a single PTA. (Recall that the PTA analyzed can be made of a network of N PTA in parallel.) This requirement is motivated by obvious efficiency reasons: exploring each PTA in an independent manner is by far more efficient than exploring the composition of several PTA, required to detect the locations in which global clocks (used by several PTA) can be safely discarded. Note that, in all case studies we considered, all clocks were always local. Extending our work to the case of global clocks is discussed in Section 5.

3.3 Static Computation of the Useless Clocks per Location

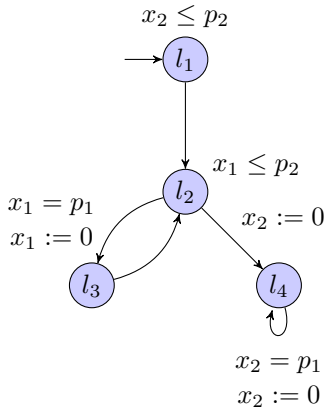
We introduce in Algorithm 1 an algorithm $useless(\mathcal{A}, x)$, that computes in a static manner the set of locations where a clock x is useless. This algorithm takes as input a PTA $\mathcal{A}$ and a clock x, and outputs the list of locations in $\mathcal{A}$ where x is useless.

Algorithm 1: $useless(\mathcal{A}, x)$

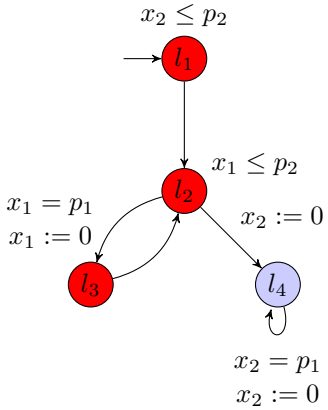
input : PTA $\mathcal{A}$, clock x
output : List of locations where x is unnecessary

```
1  Marked ← {l | ∃l', a, g, ρ : (l, a, g, ρ, l') ∈ → ∧ x ∈ g} ∪ {l | x ∈ I(l)}
2  Waiting ← Marked
3  while Waiting ≠ ∅ do
4      pick l' from Waiting
5      foreach (l, a, g, ρ, l') ∈ → do
6          if x ∉ ρ then
7              if l ∉ Marked then
8                  Marked ← Marked ∪ {l}
9                  Waiting ← Waiting ∪ {l}
10 return L \ Marked
```

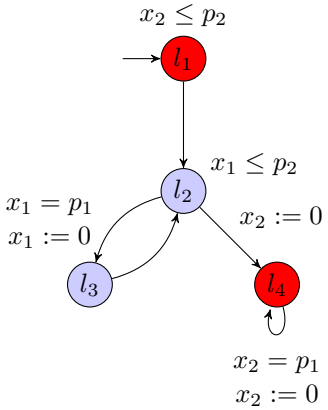
The algorithm makes use of a set of waiting locations ("*Waiting*") and a set of marked locations ("*Marked*"); this latter set corresponds to the locations where x is actually useful. Lines 1– 2 initialize the value of *Waiting* and *Marked* to the set of locations that are either predecessors of a guard involving x or have an invariant involving x. Then, it proceeds by coloring locations in a backward manner, starting from *Marked*. As long as the set of waiting locations is not empty, the algorithm picks a location l' from this set (line 4); then, for each transition whose destination location is l', the algorithm checks whether the clock x is reset along the transition (line 6). If not, and if the transition source l is not marked yet, then l is added both to the set of marked locations and to the waiting set (lines 8–9). The algorithm finally returns the set of locations in $\mathcal{A}$ that are not marked (line 10).



(a) A toy PTA $\mathcal{A}$



(b) Locations marked in $useless(\mathcal{A}, x_1)$



(c) Locations marked in $useless(\mathcal{A}, x_2)$

Figure 3 Static computation of the useless clocks: an example.

Let us apply Algorithm 1 to the simple PTA in Figure 3a and to clock x_1. Initially, $Marked = Waiting = \{l_1, l_2\}$. Let us pick l_1 from *Waiting*. Since l_1 has no predecessor, no action is performed. Let us pick l_2 from *Waiting*; l_2 has two predecessors l_1 and l_3. For l_1, $x_1 \notin \rho$, but $l_1 \in Marked$, hence again no action is performed. For l_3, $x_1 \notin \rho$, and $l_3 \notin Marked$, hence we add l_3 to both *Marked* and *Waiting*. We now pick l_3 from *Waiting*; l_3 has one predecessor l_1, already in *Marked*. The *Waiting* set is now empty, and the algorithm has marked l_1, l_2 and l_3, as showed in Figure 3b; the non-marked locations are returned, viz., $\{l_4\}$.

The result of the application of Algorithm 1 to $\mathcal{A}$ and x_2 is given in Figure 3c. The locations for which x_2 is useless are l_2 and l_3.

In the case of a network of PTA (see Definition 7), the list of useless clocks in a global location is the union, for each of the PTA in parallel, of the clocks useless in the local location for this PTA.

▶ Remark. An alternative and equivalent way to present Algorithm 1 is to use the following recursively defined function (given in a functional programming-like syntax), that decides whether a clock is useless in a given location.

```
let uselessInLoc (x, l) =
    x notin I(l)
    and
    foreach (l, a, g, rho, l' ) in steps then
        x notin g
        and ( x in rho  or  uselessInLoc(x, l') )
```

3.4 Dynamic Elimination of the Clocks in Practice

Following the static computation of the locations in which each clock is useless, we can now eliminate the clocks on-the-fly during a reachability analysis. More precisely, this is performed *after* computing the constraint associated with a new state; once this constraint has been computed, useless clocks are eliminated. This elimination is a variable elimination à la Fourier-Motzkin [21], so as not to modify the relationship between the other clocks and parameters.

Algorithm 2: Computation of a new state in IMITATOR.

input : PTA $\mathcal{A}$, state (l, C), transition (l, a, g, ρ, l')
output : New state (l', C')

1 $C' \leftarrow C \wedge g$
2 $C' \leftarrow \rho(C')$
3 $C' \leftarrow C' \wedge I(l')$
4 $C' \leftarrow C'^{\uparrow}$
5 $C' \leftarrow \mathit{Eliminate}(C')$
6 **return** (l', C')

We give in Algorithm 2 a simplified[1] version of the computation of the successor state (l', C'), generated from a source state (l, C) via transition (l, a, g, ρ, l'), as implemented in IMITATOR [5]. The addition of the clock elimination is highlighted (line 5); in this expression, $\mathit{Eliminate}(C')$ denotes the elimination of the clocks useless in the destination location l', as computed by Algorithm 1 for each clock. In IMITATOR, the variable elimination is performed using the dedicated function of the Parma Polyhedra Library [9].

3.5 Characterization

In this section, we show that applying the dynamic clock elimination during a reachability analysis preserves parametric analyses, as well as the satisfiability of linear-time properties.

Let us denote by $U(l)$ the list of clocks useless in a given location l; the result of this function can be computed by applying Algorithm 1 for each clock.

We define below the semantics of PTA under dynamic clock elimination.

[1] IMITATOR also features discrete variables, as well as stopwatches; these features are beyond the scope of this paper, and are discarded here. Furthermore, after each modification of C', a satisfiability test is performed to check whether (l', C') is valid new state; if not, it is discarded (using an exception mechanism).

▶ **Definition 8.** Let $\mathcal{A} = (\Sigma, L, l_0, X, P, K, I, \rightarrow)$ be a PTA. The *semantics of $\mathcal{A}$ under dynamic clock elimination* is $\mathcal{LTS}_{dyn}(\mathcal{A}) = (\Sigma, S, S_0, \Rightarrow_{dyn})$ where

$$S = \{(l, C) \in L \times \mathcal{L}(X \cup P) \mid C \subseteq I(l)\},$$
$$S_0 = \left\{(l_0, \left(K \wedge I(l_0) \wedge \bigwedge_{i=1}^{H-1} x_i = x_{i+1}\right)\backslash_{U(l_0)}\right\}$$

and a transition $(l, C) \Rightarrow^{a}_{dyn} (l', C')$ belongs to $\Rightarrow_{dyn}$ if $\exists C'' : (l, C) \stackrel{a}{\Rightarrow} (l', C'')$, and $C' = C''\backslash_{U(l')}$.

Hence, a transition in the semantics under dynamic clock elimination corresponds to a transition conform to the standard semantics of PTA (i.e., $(l, C) \stackrel{a}{\Rightarrow} (l', C'')$), followed by the elimination of the clocks useless in l' (i.e., $C' = C''\backslash_{U(l')}$).

We denote by $Paths_{dyn}(\mathcal{A})$ the set of paths of $\mathcal{A}$ computed using the semantics of $\mathcal{A}$ under dynamic clock elimination.

We characterize below the effect of dynamically eliminating clocks while performing a reachability analysis.

▶ **Theorem 9.** *Let $\mathcal{A}$ be a PTA. Then:*

$\Rightarrow$ *Let $(l_0, C_0) \stackrel{a_0}{\Rightarrow} \cdots \stackrel{a_{m-1}}{\Rightarrow} (l_m, C_m)$ be a path in $Paths(\mathcal{A})$. Then there exist C'_i, $0 \leq i \leq m$ such that $(l_0, C'_0) \stackrel{a_0}{\Rightarrow} \cdots \stackrel{a_{m-1}}{\Rightarrow} (l_m, C'_m)$ is a path in $Paths_{dyn}(\mathcal{A})$, with $C'_i = C_i\backslash_{U(l_i)}$ for $0 \leq i \leq m$.*

$\Leftarrow$ *Conversely, let $(l_0, C'_0) \stackrel{a_0}{\Rightarrow} \cdots \stackrel{a_{m-1}}{\Rightarrow} (l_m, C'_m)$ be a path in $Paths_{dyn}(\mathcal{A})$. Then there exist C_i, $0 \leq i \leq m$ such that $(l_0, C_0) \stackrel{a_0}{\Rightarrow} \cdots \stackrel{a_{m-1}}{\Rightarrow} (l_m, C_m)$ is a path in $Paths(\mathcal{A})$, with $C'_i = C_i\backslash_{U(l_i)}$ for $0 \leq i \leq m$.*

Proof (sketch). The first part ($\Rightarrow$) is obtained by induction on the length of the paths. Suppose the result holds for i, and let us prove it for $i+1$. Consider $(l_i, C_i) \stackrel{a_i}{\Rightarrow} (l_{i+1}, C_{i+1})$. From the induction hypothesis, there exists (l_i, C'_i) with $C'_i = C_i\backslash_{U(l_i)}$. Since $C_i \subseteq C'_i$, then there exists C''_{i+1} such that $(l_i, C'_i) \stackrel{a_i}{\Rightarrow} (l_{i+1}, C''_{i+1})$. The fact that $C'_{i+1} = C_{i+1}\backslash_{U(l_{i+1})}$ can be proved by showing that the operations in the two items of Definition 4 preserve this equality. Note that this holds only because the clocks in $U(l_i)$ and $U(l_{i+1})$ are not used in the invariants, guards and resets in the definition.

The second part ($\Leftarrow$) is obtained using a similar reasoning.

◀

Basically, Theorem 9 states that each path in $Paths(\mathcal{A})$ has an equivalent in $Paths_{dyn}(\mathcal{A})$, and conversely. Furthermore, in each state, the relationship between all parameters and all clocks (except the clocks useless in this state) is the same in both semantics; this comes from the fact that $C'_i = C_i\backslash_{U(l_i)}$.

We exhibit below two corollaries of Theorem 9. The first corollary states that the projection of the constraints associated to the states of a path in both the standard semantics and the semantics under dynamic clock elimination are the same. Hence, the clock elimination is suitable to perform parametric model checking based on paths.

▶ **Corollary 10.** *Let $\mathcal{A}$ be a PTA. Let $(l_0, C'_0) \stackrel{a_0}{\Rightarrow} \cdots \stackrel{a_{m-1}}{\Rightarrow} (l_m, C'_m)$ be a path in $Paths_{dyn}(\mathcal{A})$, and let $(l_0, C_0) \stackrel{a_0}{\Rightarrow} \cdots \stackrel{a_{m-1}}{\Rightarrow} (l_m, C_m)$ be its equivalent path in $Paths(\mathcal{A})$.*

Then $C_i{\downarrow_P} = C'_i{\downarrow_P}$, for all $0 \leq i \leq m$.

Proof. Since $C'_i = C_i\backslash_{U(l_i)}$ then $C'_i\backslash_X = C_i\backslash_X$, hence $C'_i{\downarrow_P} = C_i{\downarrow_P}$. ◀

The second corollary states that the dynamic clock elimination preserves linear time properties. Given a linear-time property, we denote by $\varphi \models Paths(\mathcal{A})$ the fact that all paths of $\mathcal{A}$ satisfy φ (and similarly for $Paths_{dyn}$).

▶ **Corollary 11.** *Let $\mathcal{A}$ be a PTA. Let φ be a linear-time property. Then $\varphi \models Paths(\mathcal{A})$ if and only if $\varphi \models Paths_{dyn}(\mathcal{A})$.*

Proof. Since each path in $Paths(\mathcal{A})$ has an equivalent path in $Paths_{dyn}(\mathcal{A})$ and vice-versa, the sets of traces are equal. Hence the linear-time properties satisfied are equal. ◀

4 Experimental Validation

This clock elimination technique has been implemented in IMITATOR [5] (since version 2.6.1) as an optional feature (option `-dynamic-elimination`). We compare the efficiency of our dynamic clock elimination technique on the inverse method *IM* [8]. This algorithm takes advantage of a known reference parameter valuation, and synthesizes a constraint around the reference valuation guaranteeing the same traces as for the reference valuation, i.e., guaranteeing that the same linear-time properties are satisfied. The two algorithms compared are (1) *IM* and (2) IM_{dyn}, i.e., *IM* where useless clocks are eliminated on-the-fly using the algorithms of Section 3. Note that, since *IM* relies on the exploration of the parametric state space (after eliminating all clocks), from Corollary 10, the result of both algorithms will be the same.

Table 1 compares the performances and results of *IM* and IM_{dyn}. Columns $|X|$ and $|P|$ denote the number of clocks and parameters of the PTA, respectively. For each algorithm, columns $|S|$, $|T|$ and t denote the number of states, of transitions and the computation time in seconds, respectively. In the last 2 columns, we compare the results: first, we divide the number of states in *IM* by the number of states in IM_{dyn} and multiply by 100 (hence, a number smaller than 100 denotes an improvement of the clock elimination); second, we perform the same comparison for the computation time. Experiments were performed on a KUbuntu 13.04 64 bits system running on an Intel Core i7 CPU 2.67GHz with 4 GiB of RAM.

Table 1 Experiments.

			IM			IM_{dyn}			Comparison	
Example	$\|X\|$	$\|P\|$	$\|S\|$	$\|T\|$	t	$\|S\|$	$\|T\|$	t	$\|S\|$	t
Figure 2	2	2	-	-	loop	2	2	0.007	0	0
Figure 3	2	2	-	-	loop	6	8	0.006	0	0
AndOr	4	12	11	11	0.047	11	11	0.050	100	106
SPSMALL	10	26	31	30	0.580	31	30	0.584	100	101
Train	3	6	78	94	0.100	61	76	0.072	78	72
BRP	7	6	429	474	3.50	429	474	3.21	100	92
CSMA/CD_6	3	3	13,365	14,271	19.6	13,365	14,271	19.5	100	99
RCP	5	6	327	518	0.68	181	282	0.41	55	60
AAM06	3	8	1,497	1,844	8.28	768	997	2.92	51	35
AM02	3	4	182	215	0.392	182	215	0.386	100	98
BB04	6	7	806	827	25.4	806	827	27.2	100	107
CTC	15	21	1,364	1,363	83.4	201	291	2.52	15	3.0
LA02	3	5	6,290	8,023	710	4,932	7,154	473	78	67
LPPRC10	4	7	78	102	0.375	78	102	0.395	100	105

Description of the Models

The first 2 models are the looping PTA in Figure 2 and Figure 3a. The next 2 models are asynchronous circuits [13, 8]. The next case study is a classical train–gate–controller from [4]. The next 3 models are common protocols [14, 18, 17]. The other models are scheduling problems [1, 2, 12, 23, 20]. All models are described and available (with sources and binaries of IMITATOR) on IMITATOR's Web page[2].

Interpretation of the Experiments

Let us comment the experiments in Table 1. Although only the 2 toy models are such that only IM_{dyn} can analyze them whereas IM loops, the optimization of IM_{dyn} also leads to state space reductions in many other models. These state space reductions come from the fact that useless clocks may in general lead to the creation of many similar states, only different with respect to the (generally increasing) value of these clocks; when the useless clocks are eliminated, all these similar states are replaced with only one state.

The use of the optimized version IM_{dyn} has the following advantages. First, the state space is often reduced compared to the classical IM (without clock elimination). Although the dynamic elimination of clocks does not seem to bring anything in the case of hardware verification, it seems much more interesting for protocols and scheduling problems. This is particularly interesting for the scheduling problems, with a division of the number of states by a factor of up to 6 (CTC). Second, the computation time is always reduced when the dynamic clock elimination indeed reduces the state space, by a factor of up to 33 (CTC). Third, and more surprisingly, the overhead brought by the dynamic elimination does not yield a significant augmentation of the computation time, even when the clock elimination does not reduce the state space at all; the worst case is +7 % (BB04), which remains very reasonable. These experiments encourage us to consider to set this optimization as default in IMITATOR.

Finally, in some cases (BRP, CSMA/CD, AM02), the computation time is smaller in the case of dynamic clock elimination, despite the absence of state space reduction – which is surprising. This may be due to little variations of the processor. This might also be explained by the fact that, even when states are not merged, the computation of the successor states may be more efficient when the constraints are smaller (i.e., have fewer clocks).

5 Conclusion

We introduced here a state space reduction technique based on an on-the-fly elimination of unnecessary clocks in parametric timed automata. This technique has the following advantages: (1) some models that include loops preventing termination may terminate; (2) the relationship between the remaining clocks and parameters is preserved, which makes it suitable for many (parametric) model checking algorithms; (3) the application of this technique to the inverse method (implemented in IMITATOR) shows interesting state space reductions without adding any significant overhead in terms of computation time.

[2] `http://www.lsv.ens-cachan.fr/Software/imitator/dynamic/`

Future Work

So far, we considered only local clocks, i.e., clocks used in only one of the different PTA in parallel. Considering global clocks (i.e., used in most of the PTA describing the model) would be interesting. In order to avoid the static composition of all PTA prior to the analysis, this would require more complex algorithms than our current detection of the locations where a clock can be discarded. An alternative is to combine our technique with the technique of global clock elimination introduced in [10], if this latter technique can be extended to the parametric setting.

A future extension consists in extending the second algorithm of [15], i.e., to dynamically eliminate clocks that are *equal* to another clock. Although simple in theory, this optimization would require some operations on the constraints that may turn more complex and time-consuming in the parametric setting (using polyhedra) than in the non-parametric setting (using difference bound matrices).

We aim at extending this work to the case of hybrid systems, where clocks are generalized to variables with (in general) arbitrary rates. This could then be applied to the inverse method generalized to hybrid systems [16].

We are also interested in studying the optimization presented here with the (more restrictive) state space reduction based on convex merging recently proposed in [6].

Acknowledgement

I am grateful to an anonymous reviewer for his/her useful comments.

References

1 Yasmina Adbeddaïm, Eugene Asarin, and Oded Maler. Scheduling with timed automata. *Theoretical Computer Science*, 354(2):272–300, 2006.

2 Yasmina Adbeddaïm and Oded Maler. Preemptive job-shop scheduling using stopwatch automata. In *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 113–126. Springer, 2002.

3 Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

4 Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. Parametric real-time reasoning. In *STOC*, pages 592–601. ACM, 1993.

5 Étienne André, Laurent Fribourg, Ulrich Kühne, and Romain Soulat. IMITATOR 2.5: A tool for analyzing robustness in scheduling problems. In *FM*, volume 7436 of *Lecture Notes in Computer Science*, pages 33–36. Springer, 2012.

6 Étienne André, Laurent Fribourg, and Romain Soulat. Merge and conquer: State merging in parametric timed automata. In *ATVA*, Lecture Notes in Computer Science. Springer, 2013. To appear.

7 Étienne André, Yang Liu, Jun Sun, and Jin Song Dong. Parameter synthesis for hierarchical concurrent real-time systems. In *ICECCS*, pages 253–262. IEEE Computer Society, 2012.

8 Étienne André and Romain Soulat. *The Inverse Method*. FOCUS Series in Computer Engineering and Information Technology. ISTE Ltd and John Wiley & Sons Inc., 2013.

9 Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

10 Sandie Balaguer and Thomas Chatain. Avoiding shared clocks in networks of timed automata. In *CONCUR*, volume 7454 of *Lecture Notes in Computer Science*, pages 100–114. Springer, 2012.

11 Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003.

12 Enrico Bini and Giorgio C. Buttazzo. Schedulability analysis of periodic fixed priority systems. *IEEE Transactions on Computers*, 53(11):1462–1473, 2004.

13 Robert Clarisó and Jordi Cortadella. The octahedron abstract domain. *Science of Computer Programming*, 64(1):115–139, 2007.

14 Pedro R. D'Argenio, Joost-Pieter Katoen, Theo C. Ruys, and Jan Tretmans. The bounded retransmission protocol must be on time! In *TACAS*, volume 1217 of *Lecture Notes in Computer Science*, pages 416–431. Springer, 1997.

15 Conrado Daws and Sergio Yovine. Reducing the number of clock variables of timed automata. In *RTSS*, pages 73–81. IEEE Computer Society, 1996.

16 Laurent Fribourg and Ulrich Kühne. Parametric verification and test coverage for hybrid automata using the inverse method. *International Journal of Foundations of Computer Science*, 24(2):233–249, 2013.

17 Thomas Hune, Judi Romijn, Mariëlle Stoelinga, and Frits W. Vaandrager. Linear parametric model checking of timed automata. *Journal of Logic and Algebraic Programming*, 52-53:183–220, 2002.

18 Marta Z. Kwiatkowska, Gethin Norman, Jeremy Sproston, and Fuzhi Wang. Symbolic model checking for probabilistic timed automata. *Information and Computation*, 205(7):1027–1077, 2007.

19 Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

20 Thi Thieu Hoa Le, Luigi Palopoli, Roberto Passerone, Yusi Ramadian, and Alessandro Cimatti. Parametric analysis of distributed firm real-time systems: A case study. In *ETFA*, pages 1–8. IEEE, 2010.

21 Alexander Schrijver. *Theory of linear and integer programming.* John Wiley & Sons, Inc., 1986.

22 Jun Sun, Yang Liu, Jin Song Dong, Yan Liu, Ling Shi, and Étienne André. Modeling and verifying hierarchical real-time systems using Stateful Timed CSP. *ACM Transactions on Software Engineering and Methodology*, 22(1):3.1–3.29, 2013.

23 Naoyuki Tamura. CSP2SAT: JSS benchmark results. `http://bach.istc.kobe-u.ac.jp/csp2sat/jss/`, 2007.

24 Louis-Marie Traonouez, Didier Lime, and Olivier H. Roux. Parametric model-checking of stopwatch Petri nets. *Journal of Universal Computer Science*, 15(17):3273–3304, 2009.

On the Determinism of Multi-core Processors*

Vladimir-Alexandru Paun[1], Bruno Monsuez[1], and Philippe Baufreton[2]

1 **UIIS**
ENSTA ParisTech
828, Boulevard des Maréchaux, 91762 Palaiseau Cedex, France
surname@ensta-paristech.fr
2 **Sagem – SAFRAN Electronics**
Etablissement F. Hussenot – R&T
100 avenue de Paris – 91344 MASSY Cedex France

Abstract

Hard real time systems are evolving in order to respond to the increasing demand in complex functionalities while taking advantage of newer hardware. Software development for safety critical systems has to comply with strict requirements that will facilitate the certification process. During this process, each part of the system is evaluated, requiring a certain level of assurance in order to provide confidence in the product. In particular there must be a level of confidence that the system behaves deterministically that may be based on functionality, resources and time. The success of system verification depends greatly on the capacity to determine its exact behavior. Nonetheless, hardware evolved in order to maximize the average computation power throughput with little to no regard to the deterministic aspect. Therefore modern architectural features of processors, like pipelines, cache memories and co-processors, make it hard to verify that all the needed properties are respected. The multi-core is furthermore difficult to analyze as the architecture employs mechanisms that compromise strong spatial and temporal partitioning when using shared resources without rigorous access control like shared caches or shared input/outputs. In this paper we identify and analyze the main sources of nondeterminism of the multi-cores with regard to the timing estimation. Precise determination of the worst case execution time is a challenging task even in single-core architectures. The problems are accentuated in the multi-core context mainly due to the resource sharing that can lead to highly complex interactions or to nondeterminism. Most of the units that generate behaviors that are hard to take into account can be deactivated, but it is not always easy to predict the impact on the performance. Nevertheless some of the features cannot be disabled (such as the out of order execution or some nondeterministic crossbar access policies) which leads to the invalidation of the respective platform for applications with high criticality level. We will address the problematic units, propose configuration or architecture guidelines and estimate their impact on the performance and determinism of the system.

1998 ACM Subject Classification C.3 Special-Purpose and Application-Based Systems

Keywords and phrases multi-core, determinism, hard-real time systems

Digital Object Identifier 10.4230/OASIcs.FSFMA.2013.32

1 Introduction

The use of complex computers in safety-critical systems creates the need to ensure that the embedded systems act in the way they are supposed to and that consequences of a

* This work was supported by SAFRAN Sagem.

1st French Singaporean Workshop on Formal Methods and Applications 2013 (FSFMA'13).
Editors: Christine Choppy and Jun Sun; pp. 32–46

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

malfunction are completely handled in a safe manner. Different standards apply according to the danger level of the system failure. These standards are presented in a collection of guidelines to follow in order to empower the system with a necessary confidence level. The respect of these recommendations determine the success of the certification process necessary for the software approval.

In this paper we focus on the determinism issues related to the worst-case execution time (WCET) with regards to the avionics standards of software certification that will guide the study of potential difficulties of embedding multi-cores. For the assurance of commercial avionics systems a document called "Software Considerations in Airborne Systems and Equipment Certification" is used. Bearing the name DO-178B [21] in the US and ED-12B in Europe, it describes the objectives of software life-cycle processes, process activities and the evidence of compliance required at different software levels. Safety standards like DO-178B and IEC-61508 [13] explicitly call for the identification of functional and non-functional hazards and for software compliance with the relevant safety goals. In these standards three important non-functional software characteristics related to safety are mentioned: absence of run-time errors, execution time and memory consumption. The IEC-61508 has a great impact on the hardware selection as it requires the absence of unpredictable timing-related interferences which might affect real-time functions. This impacts directly the multi-cores as it must be ensured that no inherent timing interferences between cores take place. Nevertheless, this type of interferences are quite common and must be dealt with. Existing multi-core architectures employ mechanisms that compromise strong spatial and temporal partitioning when using shared resources without rigorous access control. Therefore we do not always dispose of precise information regarding the timing of some instructions in all circumstances due to their complex interactions with the memory and other dependencies.

The WCET estimation consists of two main steps, namely the control-flow analysis that determines the feasible paths in a program, and the processor-behavior analysis based on low-level analysis, hence the need to throughly determine the hardware behavior. The choice of a hardware platform is therefore greatly influenced by the visibility on the device internal structure as precise architectural implementation details are proprietary data often undisclosed.

The WCET estimation in multi-cores has different levels of difficulties. The first one is inherited from the single core world. Certain modern features in processors cannot be safely analyzed, nor disabled, making the certification of the processor impossible. Other features generate imprecision that will increase the safe margins needed to take in order to comply with the safety constraints, which impacts the feasibility. The second one is introduced by the multi-core architecture and can lead to the impossibility to determine the WCET. Problems from a core are not only translated, when integrated into the multi-cores, but amplified by the context of resource sharing.

State of the art works deal with this issue either by constraining some platforms, or by handling only a part of the issues and giving some new architectural workarounds that are custom tailored for some applications [30, 6, 5, 15, 31, 17, 11, 10, 9, 29]. Another approach is to gather best practices for future multi-core architectures [4] after acknowledging that analysing current multicore architectures is impossible in general. Nevertheless a unified and detailed approach is yet to be available.

Identifying which parts are impossible to analyze, or at what cost in precision, is key to even considering the choice of a certain multi-core processor. Our work is intended as a guideline in such choices, exposing the inherent problems of multi-cores and straitening the path towards a solution in the matter. The article is structured as follows: First we

explain inherent problems to the use of microprocessors in hard real-time systems. In Sec. 3 we describe the major units of the processor, identify problematic execution scenarios and estimate the impact on the predictability and we conclude in Sec. 4.

2 Inherent problems to the use of microprocessors in hard real-time systems

The hardware platform is a central point when analyzing a system. Therefore it is essential to dispose of a precise model of the processor in order to determine its effective behavior. The available information comes mainly in reference manuals and application notes that present the processor's architecture, how to interface it with the environment and how to configure its different function modes. Nevertheless, information present in the user manual is not intended for testing or verification purposes. Furthermore information relevant to the design method and the verification methodology are only briefly discussed, if not at all in these documents, mainly from the integrator point of view. Another issue is the questionable validity of the information presented in the reference document altogether as contradictory information is sometimes provided in different document.

The behavior of a microprocessor is challenging or even impossible to characterize. This is either due to the uncertainty of the effectively calculated result or the uncertainty on the actual time of the effective calculation. De facto, these two aspects are directly related to the notion of data availability. The main consequence of the difficulty in architectural optimization is an interdependence between the data and the instructions of a same task. In the context of multitask applications, an interdependence between various tasks coming from the commutation of the environments during the passing from one task to another, is introduced. In the case of multi-cores additional difficulties appear in the estimation of the WCET, like the issue of two competing processes if they share common architecture elements, notably, memory access controllers. Futhermore challenges are given by the exchange of data between the two applications being executed in both processors. Let us consider two tasks, one critical task being executed in one core, and a second critical task monitoring the calculations of the first task on the second core, it is necessary to ensure the data handled by the two tasks is coherent. Without adding at the application level advanced synchronization operation, it is impossible for the majority of current multi-core microprocessors to guarantee that both applications handle the same data. In fact, due to induced latencies, nothing prevents one of the two applications from handling data d_{t-1} present at $t-1$ instant while the other application handles the data d modified at t instant. In fact, one of the rare means to remove these uncertainties would be to strongly pair the monitoring application with the command application, by implanting communications between the two applications via semaphores.

3 Hardware considerations

Most of the available processors were not especially designed for the hard real-time systems. The multi-cores are no exception as their goal is to maximize the shared resource average utilization. Data communication and synchronization between the different units are optimized for maximum throughput of executed instructions. Therefore multiple execution paths can be taken depending on the execution history, the current state of units or even local choices based on random decisions. A natural way of analyzing the hardware components that influence the determinism would be to first look at those who give a local impact and

proceed to components that have a global impact. One can also start by looking into units already present in single-cores, and proceed with units unique to the multi-cores. However, as shaped by this section, there is a thin frontier between the two as even classical, predictable units have a different impact when integrated in the multi-cores. Therefore the analysis of this components can not be solely based on the analysis of the same component in the single-core context.

3.1 Pipeline

Present in all modern processors, the pipeline was introduced in order to increase the average performance by ensuring that, whenever possible, an available hardware resource will be occupied. Nevertheless, different events can introduce pipeline stalls such as structural hazards, data hazards and control hazards.

Out of order execution (OoOE), a feature introduced in order to avoid pipeline stalls by decoupling the issue/dispatch and the execution/completion stages, allows execution not following the instructions program order. A fetched instruction will be executed when the input operands and needed resources are available with no regard to whether it is the next in order instruction. The interaction between the cache memory and instruction scheduling influences the precision of the timing estimation.

Pipeline impact on the predictability

The impact of the pipeline varies from local influence with local monotonic optimizations to global influences with timing anomalies that cancel the monotonicity and compositionality. The size of the pipeline has an influence on the predictability of the WCET. A wrong branch prediction causes n cycles penalty, where n is the pipeline depth. The pipeline depth can further influence the predictability potentially generating more hazards as more instructions are being treated at the same time. Besides intrinsic impacts on the predictability, the pipeline, in conjunction with other units, can lead to precision loss or nondeterminism. For example, in case of a L2 cache miss, the number of pipeline stages influences the memory access time [8]. Furthermore, in the shared resources context of multi-core sharing a common bus, the pipeline can lead to nondeterminism.

3.2 Branch Prediction Unit (BPU)

Through the BPU, processor attempts an early resolve of a branching instruction, before its time, by applying a strategy in order to anticipate the result. The BPU strategy can be either static or based on complex algorithms, un-deterministic in some cases. Based on this estimation, a speculative execution is initiated that will lead eventually to a significant time gain in the case the result is correct. The influence on the cache memory content is non negligible as a miss-prediction is not generally followed by a cache reorganization, therefore the cache configuration is polluted with information from the untaken path.

BPU impact on the predictability

The BPU can make incorrect branch predictions or incorrect branch target address lookup. It is systematically active and directly impacts the temporality of the instruction change. Furthermore, this unit relies on a set of data protection tables stored in tables. The impact is high because of the general unpredictable success rate of the early branching target resolution. It can be largely avoided in the case of statically resolved loops or branches that are not data

dependent. Tailoring the condition of the jump taking into account the branching strategy in order to help it succeed in the majority of cases is also a solution as long as the WCET analyzer can take it into account. In this case, most techniques of adding watermarks in the code with information that help or enable the prediction can be useful.

3.3 Floating Point Unit (FPU)

Floating point computation timing can also be hard to accurately estimate because of their implementation. A micro-pipelined unit takes advantage of consecutive instructions that can be pipelined. Units can have either a part of the FPU instructions pipelined or all of them. Therefore consecutive pipelined and non-pipelined instructions can cause stalls, making the timing difficult to compute especially in the case of the out of order execution.

Floating-point data formats and instruction set generally conform to the IEEE Standard for Binary Floating-point Arithmetic, ANSI/IEEE Standard 754-1985. However, the SPARC V8 architecture, for example, does not require that all aspects of the standard, such as gradual underflow, be implemented in hardware [25]. This can be a problem if precise information about the implementation is not given. One of its implementations, the GR712RC/LEON3 does not provide sufficient information on this matter and precise timings in case of this exception could prove difficult to estimate. Similarly the ARM Cortex A9 architecture manual, does not provide precise timing of all instructions. This is mainly due to the unpredictable timing behavior at the instruction level generated by the unit's structure itself and memory system interactions [32].

FPU impact on the predictability

The impact of the FPU depends on its implementation. Instructions can take either a single cycle to execute or several cycles but they can also be pipelined. A combination of either way in parallel is also possible. In conjunction with the instruction rescheduling and thus with the change of data, cascade effects can occur and lead to pathological effects like it can be seen in some PowerPC architectures.

3.4 Level 1 Cache

Memories for instructions and data are implemented in order to make the most common case fast, benefiting from a program's spatial locality and temporal locality. Not taking this fact into account in the WCET estimation gives highly pessimistic timing estimations. Cache memory is usually organized in different levels, some local to the core and others situated outside the core. Different cache replacement strategies must be implemented in order to optimize the performance because a strategy that can fit all the possible cases is impossible to find. Therefore the average case performance is optimized. Commonly used strategies are LRU, pseudo LRU, FIFO or round robin and MRU each having a different impact on the predictability of the system. The analysis of the Level 1 cache must be made in conjunction with the other units and is discussed within the timing anomalies in the following.

Impact on the predictability

Worst-case analysis on cache memories is a challenging problem, mainly because they are conceived in order to maximize the average performance. Achieving good results for data cache analysis, for example, is still an open problem, as they are difficult to statically analyze. An approach that enables time-predictable caching, is to lock cache blocks. Combining

cache locking with cache partitioning for multiple tasks in the case of task preemption can improve the predictability in some cases [26]. Unknown abstract cache states during the analysis generate loose WCET bounds. For example, unified cache for instruction and data can break down all the information on abstract cache states. After accessing n unknown addresses in an n-way set-associative cache all the cache lines will be unclassified in the analysis. Therefore, separation between the instruction and data cache memories should be chosen whenever possible (the problem still holds for shared caches and is discussed in the Level 2 cache section). For this reason Harvard architectures, with physically separate signal pathways for instructions and data, should be privileged in despite of the von Neumann architecture. Context switch or cache misses it can lead to a relatively high global impact due to timing anomalies or Translation Lookaside Buffer (TLB) strategies. In order to improve the performances of the cache memory, instruction and data locality could be increased using compiler techniques for example (code reposition, loop permutation, tiling [28] etc.). When performing the WCET analysis, the most problematic features to analyze are the replacement policies for set-associative caches [12]. Pseudo-round-robin and the 4-way associative cache is also a difficult combination in the Motorola ColdFire 5307 [23]. In order to ensure the time-predictability of processors, locally deterministic update strategies for caches should be used. According to [22] the LRU strategy performs best in terms of predictability, far ahead pseudo-LRU and FIFO.

3.5 Scratchpad

Scratchpad memories (SPMs) are used to guarantee a unit can work without main memory contention in a system employing multiple processors. As the memory access latencies are predictable, scratchpad memories have become popular for real-time embedded systems. However, the difficulty of allocating code/data to scratchpad memory lies now with the compiler. Scratchpad memory works like a local store and act like "software caches" therefore the strategy is implemented in software and the interactions in the global hardware must be analysed. Timing anomalies with regard to the replacement strategy should be integrated into the hardware model. The most convenient approach to manage the SPM is using static allocation [18] but dynamic SPM allocation is more efficient (it can use profile-based optimization but multiple strategies exist). Analyzing dynamic strategies is challenging, especially the software implemented ones that give optimal allocation for the average execution time. Some WCET-centric techniques exist but they do not handle all architectures.

3.6 Memory Management Unit (MMU) and Translation Lookaside Buffer

The TLB is a cache that MMU use to improve virtual addresses translation speeds. The time needed to determine the physical address depends on the number of performed operations. TLB time access is variable. In order to enforce the predictability, the MMU can be deactivated (however the performance loss is significant) or by reducing the size and thus complexity of the TLB (the TLB main entries can be blocked in order to ensure their persistence). A solution is to increase the TLB size so that we only have hits but we still have the problem of an error in the translation that is detected late and takes an undefined (even if still reasonable) time to be corrected. Typical user manuals [1] give upper bounds in the TLB miss case. Timing anomalies invalidate the monotonicity assumption in the general case [27], which means that we cannot directly use the upper-bound information as the worst-case scenario. Therefore, without precise information on the exact behavior all

possible cases must analyzed, leading to a potential state space explosion. In order to reduce the potential temporal variability, the MMU should be disabled. Nevertheless due to the consequences on the global performances it is not recommendable.

In general, virtual memory raises predictability issues at two levels. First at the level of address translation that provides mapping between virtual to physical pages requires a TLB lookup. If the mapping is absent from the TLB a page table lookup is performed. The duration of address translation is hard-to-predict, because not all mappings can be stored in the limited capacity of the TLB or because the TLB might be shared between concurrent processes. Second at the level of paging activity as knowing whether or not a reference to a virtual page will result in a page fault. This is hard to predict because physical memory is shared between concurrent processes.

Impact on the predictability

The virtual addressing and tasks using the shared cache the MMU has a global impact. In the case of multi-cores it is problematic to ensure the micro-TLB coherency. Choosing to handle the TLBs separately introduces new problems of guaranty.

3.7 BUS

As competition for resources grows, the natural solution was to use techniques that enable the access from master to slave, and utilization of shared resources in general. Through the use of switching mechanism, permission is granted to one master or the other, which introduces the need of a bus arbiter. Therefore a controller is usually implemented following different strategies that are more or less straightforward. The first difficulty comes from the implementation of the aforementioned strategies. In the case of multi-cores, the resolve of access conflicts is not always deterministic. Therefore at a given processor execution step, a strongly dataflow dependent transition can be made with no way of determine which of the competing units will have gain access to the shared resources. This behavior, otherwise seen as random, at possibly every program point makes it impossible for the analysis to converge to a useful result.

In the following, we will refer to the AMBA AHB bus protocol, an open standard widely used that give a good case study enabling us to pin-point more general advantages and disadvantages of interconnection protocols. Some of the features it provides, are the following: split transactions, several bus masters, burst transfers, pipelined operations and single-cycle bus master handover. The bus arbiter ensures that only one bus master at a time is allowed to initiate data transfers. The arbiter also receives requests from the slaves that wish to complete SPLIT transfers [3].

Preventing starvation

The arbitration algorithm between the channels can ensure that if the current owner requests the interface again it will always acquire it. The starvation problems are avoided in the LEON 3FT implementation of the AMBA AHB since the DMA engines always deassert their requests between accesses [2].

Preventing deadlocks

The SPLIT and RETRY transfer responses can both produce deadlocks. The deadlock can occur when different masters try to access a slave that issues SPLIT and RETRY responses

in a way that the slave is unable to deal with. If a slave issues a RETRY response only one master must access it at a time. More importantly, this constraint is not enforced by the AMBA AHB protocol and should be ensured by the system architecture. According to the GR712RC manual, *cache snooping should always be enabled in SMP systems to maintain data cache coherency between the processors* [2].

On master data concurrency

The bus arbiter of the AMBA AHA can manage up to 16 bus masters. It grants bus access according to the master's priority. The signals used are: HBUSREQx, HLOCKx, HGRANTx, HMASTER[3:0], HMASTLOCK and HSPLIT[15:0] as described in [3]. When a master is granted access, the HGRANTx signal is generated by the arbiter that indicates the appropriate master is currently the highest priority master requesting the bus. After the current transfer completes, the HREADY signal is HIGH and the arbiter will change the HMASTER[0:3] signal to indicate the bus master number. The ownership of the data bus is delayed from the ownership of the address bus. When the HREADY signal is HIGH, the master that owns the address bus will continue to own the address bus until its transfer will be completed. Several problems can occur from this behavior.

(a) When the master is in burst mode, performing bursts of undefined length, it should continue to assert the request until it has started the last transfer. A problem occurs if the arbiter cannot predict when to change the arbitration at the end of an undefined length burst, leading to the impossibility to accurately determine the timing of the transfer. This is what happens in our case study also.

(b) A different behavior can lead to a data inconsistency. Using a central multiplexer, each potential master present on the bus can drive out the address of the transfer immediately without having to wait to be granted the bus.
Let $\text{HADDR}_{M_1} = addr_1$ at clk_1 and $\text{HADDR}_{M_1} = addr_2$ at clk_2 and the first master, M_1 be granted master at clk_1. If $\text{HADDR}_{M_2} = addr_1$ at clk_2, the data at $addr_1$ is still unmodified but it will be as M_1 still owns the data bus which leads to data access consistency conflict at $addr_1$. The time needed to resolving such a conflict is hard to estimate. A typical example is the case of sharing un-partitioned memory. In order to avoid this case, a strong coupling is recommended between the monitoring application and the command application, by implementing the communication though semaphores.

(c) A case where the timing is difficult to compute is when both M_1 and M_2 drive out the same $addr_1$ on the same slave, M_1 is granted ownership of the bus and then it is stalled by the arbiter that grants the bus ownership to M_2. If M_2 needs to access $addr_1$ we cannot precisely determine the time when M_2 will finish its action.
For example, this can occur in shared un-partitioned or partitioned memory.

Impact on the predictability

The bus and its arbitration strategy are at the core of the predictability and determinism issues. Because the main role is to grant access to different participants to the shared resources, certain properties must be ensured (fairness, deadlocks prevention) while still being able to ensure good average performances and timing predictability. Therefore when choosing a particular architecture for the hard real-time systems, certain bus architectures and arbitration algorithms should be privileged. Most of the multi-core architectures implement round-robin-like arbiters which allows considering an upper bound on the latency of the access to the shared resources. In [19] a round-robin-like bus arbiter to the shared memory

hierarchy in a multi-core architecture is proposed that facilitates the systems predictability. The round robin arbiter can avoid the bus starvation. Nevertheless, the maximum length of a burst for each peripheral connected to the bus influences the maximum delay induced by bus contention. This may lead to high maximum delay bounds and may not be enough to provide firm real time guarantees for heavily loaded systems. Furthermore, having variable burst lengths, combined with the ability to pause them (split transfers), influences the predictability of the WCET. Bus contention can be avoided by using the TDMA bus arbitration with the cost of wasting bus band when the bus load is low. By manipulating the TDMA time slots, the maximum delay bound on transactions is controllable by the designer.

3.8 Direct Memory Access (DMA)

DMA allows access to the system memory independently from the CPU. Therefore the processor can proceed with its computations while waiting for relatively slow input/output data transfer. In the multi-core case, DMA is also used for intra-chip data transfer.

Error handling

The DMA controller does not generally detect deadlocks in its communication channels, so it is up to the system to manually abort the DMA transfer. The DMA unit can be disabled not without a strong impact on the performances of certain class of applications.

3.9 Level 2 cache

Level 2 cache memory can be either private to each core or shared among cores. Data hazard is one of the factors that become even more prominent in the case of multi-cores because of memory sharing, even though already present in the non-shared L1 memory. Moreover, timing anomalies render the result hard to predict like in the case when a cache miss from a core can reconfigure the memory in a state that is, timing wise, beneficial to the other. This also applies in other cache related scenarios. The problems that can occur in the analysis of the interactions with the pipeline are detailed in the timing anomalies part.

Level 2 cache impact on the predictability

The impact of the shared cache is high and global and gets amplified in the context switch case. Modeling the behavior of shared caches between cores is practically impossible because of the possible interactions between concurrent threads running on different cores [23]. When using a shared cache with parallel programs running on the multi-core processor, a cache-coherency mechanism must be implemented. The WCET analysis of such systems must calculate the worst-case delay caused by maintaining the cache coherence between different cores. Furthermore, resource contention and inter-thread conflicts among the program threads should be considered. Under the assumption that the bus strategy can be statically analyzed, the second level of cache can be made predictable by partitioning the L2 cache for each core [24].

3.10 Timing Anomalies

Timing anomalies inside a given core influence the WCET estimation of integrating multi-cores in the case of shard memory because of the tight coupling of its internal units and the

shared resources. Therefore we cannot ignore the WCET estimation problems of single cores as they are translated into the multi-core case also.

Example of timing anomalies in multi-cores A timing anomaly occurs when a local worst case contributes to the global favorable case. In the case of multi-core, such an example is a cache miss on one core that generates a series of cache hits on the other and vice versa. An example of each of these timing anomalies is given. The architectural setup is configured of two cores with private L1 cache memory but shared L2 cache memory (instruction and data).

a A cache miss on one core can generate an overall improvement of the global WCET. Let l be the cache line that will replace the obsolete line in the cache according to the implemented strategy. If l contains information that will benefit the second core then it can generate several cache hits that were not predicted. Furthermore, this line will be promoted in the priority hierarchy and could furthermore avoid future misses of the first core.

b A cache hit on the first core generates the persistence of a cache line in the disadvantage of another that will be replaced after a future cache miss. If the replaced cache line would have generated several cache hits on the second core, the overall timing performance is affected. A first core's cache hit followed by a cache miss is worse than a first core's cache miss followed by a cache miss.

c Timing amplification example as a generalization of b) A series of cache hits on the first core with a higher frequency, then the cache accesses of the second core make that every cache miss of the first core lead to the elimination of the second core's cached lines and a great amount of cache misses.

Timing anomalies remarks

As previously stated, several types of timing anomalies exist. Some are inherent to instruction execution order and are generally caused by greedy scheduler that will change the instruction execution order causing inversion or amplification of the execution time difference. Others are caused by parallel decomposition and divide et impera approaches to WCET estimations. As the first ones cannot be avoided, the others may prove essential for the possibility to construct an efficient processor behaviour analysis that does not need to search the whole state space for the whole program at once. The timing anomalies determine three infeasibility criterions in the following.

Criterion a) Let p be the processor architecture model, we say that the estimation of the WCET or more generally the processor behaviour analysis cannot be completed on behalf of the parallel decomposition ($PD(p)$) if there is no other scalable method that can do the analysis of p without the $PD(p)$. In other words, not applying parallel decomposition can affect the scalability and applicability of the estimation method. We proceed by questioning the safeness and efficiency on dealing with parallel timing anomalies. [16] formalizes the different types of timing anomalies and presents cases when the parallel timing anomalies can lead to the underestimation of the WCET with parallel composition.

Criterion b) The use of parallel decomposition in the processor behaviour analysis leads to the underestimation of the WCET in the case of coupled parallel timing anomalies. This point can be referred to in order to decide the use of parallel decomposition. In [14] a solution to take into account timing anomalies in general is described. The method uses compilation techniques and modifies the binary by instruction injection in order to avoid timing anomalies. The main idea is to interfere with the prefetch stage and ensure that we start with an empty or flushed prefetch window hence there is never an excess

instruction waiting to be executed. It can be seen as a compile time method to disable the instruction prefetch in the case where all the slots of the instruction prefetch queue are filled with NOPs. The reference also provides estimation of the overhead when using this method as ranging between 33% and 300%. This leads us to another infeasibility criterion that is related to the maximum overhead allowed for the target platform.

Criterion c) Let $Ot(p)$ be the upper-bound of the overhead on a target platform and $Op(p)$ be the overhead of filling prefetch slots with NOPs. If $Op(p) > Ot(p)$ then the analysis does not pass the feasibility test.

Table 1 Architectural impact on the determinism.

Unit	Problems / Failure mode	Problem frequency	Solutions	Impact Level
TLB	TLB misses times are hard to predict	M/ L	Increase the TLB size	M
TLB	TLB error	L	**None => disable**	M/H
MMU	The time to access the tables can take several cycles	M	Depends on the availability of the behavioural model and corresponding timings. Disabling the MMU will only affect performances if we use its features. This means, flat address mapping, no memory protection in the case a process reads/writes the address space of another process and not least, when performing a context switch there is no longer possible to identify the cached lines of a certain process.	**H**
Scratch-pad	Application controlled -> hard to estimate the timings		Analysing dynamic strategies is not an easy task, especially when being software implemented that give optimal allocation for the average execution time. Some WCET-centric techniques exist but they do not handle all architectures.	M
L1 cache	Timing anomalies	H	Construct complex, accurate processor model	**H**
L2 cache	Timing anomalies	H	**Deactivate**	**H**

Table 1 – continued from previous page

Unit	Problems / Failure mode	Problem frequency	Solutions	Impact Level
L2 cache	Data conflicts	M	Partition (core access separation) Partial solution by considering inter-thread instruction conflicts [30] Only solutions for instruction caches in some configurations are presented in [11]. [6] Addresses only instruction caches with no code sharing, LRU strategy, no data –instruction memory interference, without timing anomalies, so a very restricted environment. [7] Deals with inter-thread interferences but in a restricted architecture. No details are given concerning the shared resources granting policy or about the BUS context.	H
L2 cache	Unknown behaviour induced by the arbiter	L	**Deactivate**	**H**
L3 cache	Timing anomalies	H	Partition	M/H
L3 cache	Data conflicts	M/L	**Deactivate**	M/H
BUS	Arbitration, timing anomalies, memory interference	H	None for the general case In [6] a very restricted "BUS" is analysed with fully separated code and data accesses and no inter-process communication through shared memory and TDMA based static scheduling where a fixed length bus slot is allocated to each core in a round-robin fashion.	**H**
Arbiter	Nondeterministic		None	**H**
Arbiter	Starvation		Software supervision, but the risk of using it is even higher. DMA engines should always deassert their requests between accesses in order to prevent starvation.	**H**
Arbiter	Deadlocks		Prevent by careful hardware integration of the bus arbiter protocol. Can rarely be disregarded by construction.	**H**
Pipeline	Timing anomalies	H	Construct complex, accurate processor model. A solution to take into account all the timing anomalies (that might prove efficient in the multi-core case) is presented in [20].	**H**
FPU	High complexity		Construct complex, accurate processor model	M/L

Table 1 – continued from previous page

Unit	Problems / Failure mode	Problem frequency	Solutions	Impact Level
FPU	Mixes multi-cycle instructions (for which timing estimation is not always possible) and single cycle instructions		None	M/H
FPU	Can generate computational errors. Timing of exception catching is hard to precisely determine		Must analyse the fault tolerant mechanism. If the behaviour is taken into account at each step, might prove very costly.	M/H
ALU	Can generate computational errors.	L	Construct complex, accurate processor model taking into account the fault tolerance.	M/L
BPU	In the strategy is fixed and the prediction is wrong, the time penalty is very important.	M	Construct complex, accurate processor model	**H**
BPU	If the prediction strategy is not fixed, it can be very difficult to model or even impossible if randomness is used.	L	None	**H**

4 Conclusion

Software verification and quality assurance process of hard real-time systems in general are of great importance. Non-functional properties, such as timing, are highly dependent on the underlying hardware platform. Nevertheless, there is a rising demand to integrate more complex processors, such as the multi-cores, even though many problems are yet to be solved in single-cores. Powerful industrial WCET estimation tools available today can do nothing against the lack of information regarding the exact behavior of the platform or the nondeterministic behavior of certain units. Therefore the choice of the processor is crucial in ensuring the success of the system verification.

We have presented the behavior of several units that pose problems concerning the WCET estimation, found either in multi-cores or single-cores. Each unit description is followed by the problematic behavior and the remarks regarding its impact on the predictability. The results can be used to invalidate certain units or architectures and also as a guideline for further analysis.

References

1 Aeroflex. *UT699 LEON 3FT/SPARCTM V8 MicroProcessor, Functional Manual*, 2012.

2 Aeroflex Gaisler AB. *GR712RC - Dual-Core LEON3FT SPARC V8 Processor, User's Manual*, 2011.

3 ARM. *AMBA Specification (Rev 2)*, 1999.

4 Christoph C., Christian F., Gernot G., Grund D., Maiza C., Reineke J., Triquet B., and Wilhelm R. Predictability considerations in the design of multi-core embedded systems. In *Proceedings of Embedded Real Time Software and Systems*, pages 36–42, May 2010.

5 S. Chattopadhyay and A. Roychoudhury. Scalable and precise refinement of cache timing analysis via model checking. In *Proceedings of the 2011 IEEE 32nd RTSS*, RTSS'11, 2011.

6 S. Chattopadhyay, A. Roychoudhury, and T. Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In *Proceedings of the 13th International Workshop on Software 38; Compilers for Embedded Systems*, SCOPES'10, pages 6:1–6:10, New York, NY, USA, 2010. ACM.

7 F. Chen, D. Zhang, and Z. Wang. Characterizing the inter-thread interference of multi-core architectures for accurate wcet estimations of real-time applications. In *Przeglad Elektrotechniczny*, 2012.

8 N. Drach, A. Seznec, and D. Windheiser. Direct-mapped versus set-associative pipelined caches. In *Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques*, PACT 95, 1995.

9 D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, RTSS'09, pages 68–77, Washington, DC, USA, 2009. IEEE Computer Society.

10 D. Hardy and I. Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *Proceedings of the 2008 Real-Time Systems Symposium*, RTSS'08, pages 456–466, Washington, DC, USA, 2008. IEEE Computer Society.

11 D. Hardy and I. Puaut. Estimation of cache related migration delays for multi-core processors with shared instruction caches. In Laurent George and Maryline Chetto andMikael Sjodin, editors, *17th International Conference on RTNS*, pages 45–54, Paris, France, 2009.

12 R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038 – 1054, july 2003.

13 International Electrotechnical Commission. *IEC 61508 Functional safety of electrical/electronic/programmable electronic safety-related systems*, 2010.

14 A. Kadlec, R. Kirner, and P. Puschner. Avoiding timing anomalies using code transformations. In *Proc. 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 123–132, May. 2010.

15 T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury. Bus-aware multicore WCET analysis through TDMA offset bounds. In *Proceedings of the 2011 23rd Euromicro Conference on Real-Time Systems*, ECRTS'11, pages 3–12, Washington, DC, USA, 2011. IEEE Computer Society.

16 R. Kirner, A. Kadlec, and P. Puschner. Precise worst-case execution time analysis for processors with timing anomalies. In *Real-Time Systems, 2009. ECRTS'09. 21st Euromicro Conference on*, pages 119–128, July.

17 Y. Liang, H. Ding, T. Mitra, A. Roychoudhury, Y. Li, and V. Suhendra. Timing analysis of concurrent programs running on shared cache multi-cores. *Real-Time Syst.*, 48(6):638–680, November 2012.

18 P. Panda, N. Dutt, and A. Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *Proceedings of the 1997 European conference on Design and Test*, EDTC'97, pages 7–, Washington, DC, USA, 1997. IEEE Computer Society.

19 M. Paolieri, E. Quiñones, F. Cazorla, G. Bernat, and M. Valero. Hardware support for WCET analysis of hard real-time multicore systems. *SIGARCH Comput. Archit. News*, 37(3):57–68, June 2009.

20 V. A. Paun and B. Monsuez. Adaptable and precise worst case execution time estimation tool. In *LCTES 2012 Work-in-Progress Proceedings*, LCTES'12, 2012.

21 Radio Technical Commission for Aeronautics. *DO-178B Software Considerations in Airborne Systems and Equipment Certification.*

22 J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Syst.*, 37(2):99–122, November 2007.

23 M. Schoeberl. Time-predictable cache organization. In *Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD 2009)*, pages 11–16. IEEE Computer Society, 2009.

24 M. Schoeberl, B. Huber, and W. Puffitsch. Data cache organization for accurate timing analysis. *Real-Time Systems*, DOI: 10.1007/s11241-012-9159-8:1–28, 2012.

25 SPARC International Inc. *SPARC V8 architecture manual, Revision SAV080SI9308*, 1992.

26 Xavier Vera, Björn Lisper, and Jingling Xue. Data caches in multitasking hard real-time systems. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, RTSS'03, pages 154–, Washington, DC, USA, 2003. IEEE Computer Society.

27 I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *Quality Software, 2005. (QSIC 2005). Fifth International Conference on*, pages 295 – 303, sept. 2005.

28 M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI'91, pages 30–44, New York, NY, USA, 1991. ACM.

29 J. Yan and W. Zhang. Hybrid multi-core architecture for boosting single-threaded performance. *SIGARCH Comput. Archit. News*, 35(1):141–148, March 2007.

30 J. Yan and W. Zhang. WCET analysis for multi-core processors with shared L2 instruction caches. In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS'08. IEEE*, pages 80 –89, april 2008.

31 J. Yan and W. Zhang. Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches. In *15th IEEE International Conference RTCSA'09*, 2009.

An Improved Construction of Petri Net Unfoldings

César Rodríguez and Stefan Schwoon

LSV, ENS Cachan & CNRS, INRIA Saclay
61, Av. du Président Wilson
94235 Cachan Cedex, France
cesar.rodriguez@lsv.ens-cachan.fr, stefan.schwoon@lsv.ens-cachan.fr

Abstract

Petri nets are a well-known model language for concurrent systems. The unfolding of a Petri net is an acyclic net bisimilar to the original one. Because it is acyclic, it admits simpler decision problems though it is in general larger than the net. In this paper, we revisit the problem of efficiently constructing an unfolding. We propose a new method that avoids computing the concurrency relation and therefore uses less memory than some other methods but still represents a good time-space tradeoff. We implemented the approach and report on experiments.

1998 ACM Subject Classification D.2.2 Design Tools and Techniques, F.1.1 Models of Computation, F.3.1 Specifying and Verifying and Reasoning about Program

Keywords and phrases Concurrency, Petri nets, partial orders, unfoldings

Digital Object Identifier 10.4230/OASIcs.FSFMA.2013.47

1 Introduction

Model checking is a practical way of ensuring the correctness of concurrent systems, but suffers from the problem of *state-space explosion* (SSE). One source of SSE is the explicit representation of concurrent actions by their interleavings. Petri nets are a model of concurrent systems, and their *unfoldings* are an established approach for coping with this source of SSE.

An unfolding can be thought as a partial order that compactly represents the state space of a Petri net. Roughly speaking, the unfolding of a net N is another *acyclic* net $\mathcal{U}_N$ that behaves like N. Actually, one is usually interested in a prefix $\mathcal{P}_N$ of $\mathcal{U}_N$ that represents all reachable markings of a bounded net N. An unfolding can be seen as a time/space tradeoff: problems such as coverability or deadlock checking are PSPACE-complete in N, but only NP-complete in $\mathcal{P}_N$. On the other hand, $\mathcal{P}_N$ is usually rather larger than N but often exponentially smaller than its reachability graph, and the aforementioned problems can easily be encoded into SAT. Also, the same prefix can answer multiple queries once constructed. See [2] for a survey on unfoldings. Tools like Mole [11] or Punf [6] efficiently construct unfoldings of safe nets.

Unfoldings are built iteratively. The central challenge of their construction is to identify the events of $\mathcal{U}_N$, which requires to find sets of concurrent conditions of $\mathcal{U}_N$. This is a computationally difficult problem (NP-complete), and several approaches to it have been proposed in the literature. In [3], the authors propose using a *concurrency relation*, i.e., determine for all pairs of conditions of $\mathcal{U}_N$ whether they are part of some reachable marking. This tends to be fast but memory-intensive. An orthogonal technique are *prefix trees* [7], which try to reduce the combinatorial overhead associated to the search. These techniques can be combined, for instance Punf implements them both.

In this paper, we propose an alternative to using concurrency relations for the case of safe nets. Our contribution is an efficient traversal of the unfolding that detects concurrent pairs

1st French Singaporean Workshop on Formal Methods and Applications 2013 (FSFMA'13).
Editors: Christine Choppy and Jun Sun; pp. 47–52

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

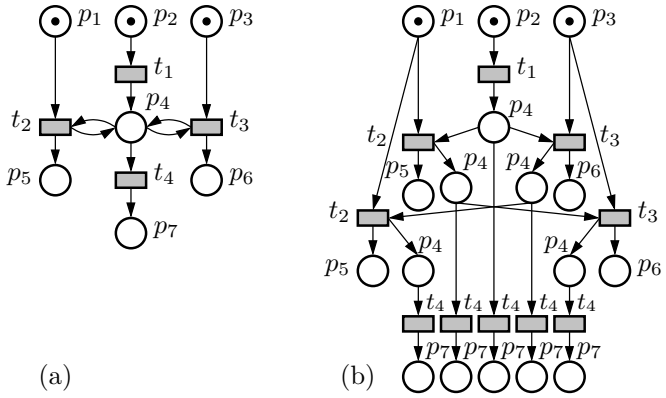


Figure 1 A Petri net N (a) and its unfolding $\mathcal{U}_N$ (b) and associated labelling.

of conditions 'on demand', without computing or storing the entire concurrency relation.

In Sec. 2, we formally introduce Petri nets and unfoldings. The algorithm that constitutes the main contribution of the paper is presented in Sec. 3. We implemented and tested the approach and report on benchmarks in Sec. 4 and conclude in Sec. 5.

2 Unfoldings of Petri Nets

A *Petri net*, or just *net*, is a tuple $N := \langle P, T, F, m_0 \rangle$, where P and T are the *places* and *transitions*, F is the *flow relation*, and $m_0 \colon P \to \mathbb{N}$ is the *initial marking*. Places and transitions together are called *nodes*. Fig. 1 (a) shows the usual graphical representation of a net with seven places and four transitions. The arrows depict the flow relation.

For any node x, let ${}^\bullet x := \{\, y \in P \cup T \colon (y, x) \in F \,\}$ be the *preset*, and $x^\bullet := \{\, y \in P \cup T \colon (x, y) \in F \,\}$ the *postset* of x. We lift these notions to sets of nodes in the expected way. A *marking* is a function $m \colon P \to \mathbb{N}$ that assigns *tokens* to every place. A transition t is *enabled* at m if $m(p) \geq 1$ for all $p \in {}^\bullet t$. Such t can *fire*, producing marking m', where $m'(p) = m(p) - |\{p\} \cap {}^\bullet t| + |\{p\} \cap t^\bullet|$. A sequence $\sigma = t_1 \dots t_n \in T^*$ is a *run* leading to marking m if t_1 is enabled at m_0, all t_i, $i \geq 2$, are enabled at the marking produced by t_{i-1}, and m is produced by t_n. A marking m is *reachable* if some run σ leads to it. N is *safe* if $m(p) \leq 1$ for all reachable m and $p \in P$. In this paper we only consider safe nets, and identify their markings with subsets of P. A set $X \subseteq P$ of places is *coverable* if $X \subseteq m$ for some reachable marking m.

The *unfolding* of N is a net $\mathcal{U}_N := \langle B, E, G, \tilde{m}_0 \rangle$ equipped with a *labelling* $h \colon (B \cup E) \to (P \cup T)$ that maps places and transitions of $\mathcal{U}_N$, called *conditions* and *events*, to places and transitions of N, respectively. When $h(x) = y$, we say that x is a "copy" of y or that x is y-labelled, and naturally extend h to sets and sequences. $\mathcal{U}_N$ and h are defined inductively:

$$\frac{p \in m_0}{c := \langle \bot, p \rangle \in B \quad h(c) := p \quad c \in \tilde{m}_0} \textsc{Ini} \qquad \frac{t \in T \quad X \subseteq B \quad h(X) = {}^\bullet t \quad X \text{ is coverable}}{e := \langle X, t \rangle \in E \quad {}^\bullet e := X \quad h(e) := t} \textsc{Ev}$$

$$\frac{e \in E \quad h(e) = t \quad t^\bullet = \{\, p_1, \dots, p_n \,\}}{c_i := \langle e, p_i \rangle \in B \quad e^\bullet := \{\, c_1, \dots, c_n \,\} \quad h(c_i) := p_i} \textsc{Cond}$$

Intuitively, $\mathcal{U}_N$ is an acyclic version of N: One starts with a "copy" of marking m_0, i.e. one condition $\langle \bot, p \rangle \in \tilde{m}_0$ for each $p \in m_0$ (see Ini). Then, whenever $\mathcal{U}_N$ can reach a marking $\tilde{m}$ such that $h(\tilde{m})$ enables t, we attach a copy of t to $\mathcal{U}_N$ (Ev). This copy, called $e = \langle X, t \rangle$ satisfies $X = {}^\bullet e$, $h(X) = {}^\bullet t$, and has 'fresh' copies of $t^\bullet$ in its postset (Cond). Thus, $\mathcal{U}_N$

is 'acyclic', and all conditions have at most one event in their preset. Fig. 1 (b) shows the unfolding, and associated labelling, of the net shown in Fig. 1 (a).

$\mathcal{U}_N$ has the same reachable markings and firing sequences as N, modulo h. In general, $\mathcal{U}_N$ is infinite, and applications usually compute a finite prefix $\mathcal{P}_N$ of it that is *complete* w.r.t. some application-dependent criterion, e.g., $\mathcal{P}_N$ is called *marking-complete* when for all reachable marking m in N there exists a reachable marking $\tilde{m}$ in $\mathcal{P}_N$ with $h(\tilde{m}) = m$. The details of such completeness criteria are beyond the scope of this exposition, see e.g. [4, 8].

For every pair of nodes x, y in $\mathcal{U}_N$ exactly one of three cases holds [4]:

- x and y are *causally related*, denoted $x < y$ (or $y < x$), if there is a path of flow arcs from x to y (resp. from y to x) in G. By construction, $<$ is an irreflexive partial order; if $x < y$, then x needs to occur before y in a finite firing sequence. We denote $\leq$ as reflexive closure of $<$. The *cone* of node x contains the causal predecessors of x, i.e., $[x] := \{\, y \colon y \leq x \,\}$.
- x and y are *in conflict*, written $x \# y$, if they compete for a token, i.e., $\#$ is the least symmetric relation on nodes satisfying (1) $e \# e'$ if $e, e' \in E$ with $e \neq e'$ and ${}^\bullet e \cap {}^\bullet e' \neq \emptyset$; and (2) $x \# z$ if there is $y \in B \cup E$ such that $x \# y$ and $y < z$. Intuitively, if $x \# y$, then no run fires or marks *both* x and y.
- x and y are called *concurrent*, written $x \parallel y$, if they are neither causally related nor in conflict. Thus, if x and y are conditions, then $\{\, x, y \,\}$ is coverable.

The principal algorithmic challenge to construct a prefix of $\mathcal{U}_N$ is to identify coverable sets of conditions X in applying the rule Ev. Given a prefix $\mathcal{P}_N$ of $\mathcal{U}_N$, it is NP-complete to decide whether $\mathcal{P}_N$ can be extended with another event [9]. The following approaches were proposed and implemented in tools:

- Since X is coverable iff $c \parallel c'$ for all $c, c' \in X$, it is promising to construct the *concurrency relation* $\parallel$ restricted to conditions. In [3], it is shown how $\parallel$ can be computed "on the fly" while constructing $\mathcal{P}_N$. This approach is implemented in the tool Mole.
- Eschewing the computation and storage of $\parallel$, [7] proposes several techniques to optimize the computations of relevant coverable sets using only memory linear in the size of $\mathcal{P}_N$; these techniques are implemented in Punf.

Experimentation over realistic benchmarks suggest that the first approach is usually faster but also more memory-intensive, in the worst-case quadratic in $|B|$; the second approach therefore succeeds to solve some big instances where the first runs out of memory. Punf actually allows to switch from the first to the second after the unfolding exceeds a given size.

3 The Algorithm

In this section, we describe the contribution of this paper: a new way of computing the events of $\mathcal{U}_N$. Like [7], this method does not employ the concurrency relation between conditions and uses only a constant amount of memory per condition and event, yet it is orthogonal to the tricks proposed in [7].

Before presenting the new contribution, we first describe a generic abstract algorithm for building $\mathcal{U}_N$, used for instance in [3, 7]. The algorithm maintains a set PE of so-called *possible extension*s, i.e., events that may be added by applying rule Ev to the prefix $\mathcal{P}_N$ generated so far. Its steps are:

1. Start with $\tilde{m}_0$, generated by the rule Ini, and fill PE with events $\langle X, t \rangle$ where $X \subseteq \tilde{m}_0$.
2. As long as PE is non-empty, remove an event e from PE. Let $\mathcal{P}'_N$ be the prefix obtained by adding e and its postset to $\mathcal{P}_N$, by means of rules Ev and Cond.
3. Identify and add to PE the set of possible extensions of $\mathcal{P}'_N$ that were not possible in $\mathcal{P}_N$. For any such extension $\langle X, t \rangle$, X necessarily intersects $e^\bullet$.
4. Set $\mathcal{P}_N := \mathcal{P}'_N$ and continue at step 2.

As mentioned in Sec. 2, this procedure may not terminate, and practical applications usually truncate $\mathcal{P}_N$ at certain *cutoff points*. This aspect is irrelevant to our contribution and we do not discuss it, focusing instead on step 3, the only difficult one. For the rest of this section, let $e := \langle X, t\rangle$ be the event in step 3. We proceed in two substeps:

3a. For each place $p \in {}^\bullet(t^{\bullet\bullet}) \setminus t^\bullet$, determine the set $C(p, e)$ of p-labelled conditions $\langle x, p\rangle$ that are concurrent with e. For $p \in t^\bullet$, we set $C(p, e) := \{\langle e, p\rangle\}$

3b. For all $t' \in t^{\bullet\bullet}$, use the sets $C(p, e)$ to discover new possible extensions, i.e., find coverable subsets X' with $h(X') = t'$ and add these to PE.

While step 3a can be implemented in time linear in $|\mathcal{P}_N|$, it is known that step 3b is NP-complete, even when the concurrency relation is given [5]. On the other hand, profiling on many benchmarks (for instance, using MOLE) suggests that step 3a is more expensive in practice than step 3b.

On the one hand, [3] proposes to compute sets $C(p, e)$ using the concurrency relation and discusses the on-the-fly computation and maintenance of the latter, giving little detail on step 3b. On the other hand, [7] presents heuristics for speeding up step 3b without discussing step 3a in detail. We shall discuss a method for implementing step 3a efficiently but without storing the concurrency relation and using only $\mathcal{O}(|\mathcal{P}_N|)$ memory. This method can be combined with the optimizations of step 3b from [7].

We start with a series of simple observations, which are valid for unfoldings of safe Petri nets. Let p be a place of N and $h^{-1}(p)$ the set of conditions labelled by p in $\mathcal{U}_N$. Since N is safe, no two elements of $h^{-1}(p)$ can be concurrent. Thus, the causality relation $<$, restricted to $h^{-1}(p)$, forms a forest where any pair of conditions c, c' that are not causally related are in conflict. Let us call this the *p-forest*. Now, let $c, c' \in h^{-1}(p)$ and e an event. We observe that (i) if $c \# e$ and $c < c'$ then $c' \# e$; (ii) if $c < e$ and $c' < c$ then $c' < e$; and in particular in both cases $c' \parallel e$ does not hold. Moreover, let $C' = h^{-1}(p) \cap [e]$ for some event e. Then no two elements of C' can be in conflict, and therefore (iii) C' must be totally ordered w.r.t. $<$.

Based on these observations, our algorithm for step 3a consists of the following steps:

I. We traverse the causal predecessors of e, i.e. the cone $[e]$. This serves two purposes:
- Mark all elements of $[e]$ with a special bit that allows to determine, in constant time, whether any given node of $\mathcal{P}_N$ belongs to $[e]$;
- Update the p-forest for all $p \in t^\bullet$: if $C' := h^{-1}(p) \cap [e]$ is empty, then the condition $\langle e, p\rangle \in e^\bullet$ is a root of the p-forest, otherwise it is a child of the maximal element of C', due to (iii).

The traversal takes linear time in $|[e]|$.[1]

II. Now, let $p \in {}^\bullet(t^{\bullet\bullet}) \setminus t^\bullet$. We determine $C(p, e) \subseteq h^{-1}(p)$ by traversing the p-forest in an order that respects $<$, starting at the roots of the forest. Let $c \in h^{-1}(p)$.
- if $c < e$ (constant-time check due to I.), then $c \notin C(p, e)$; however, some of its children in the p-forest may be, so we continue to explore those;
- if $c \# e$, then $c \notin C(p, e)$, and neither are any of its children in the p-forest, cf. (i).

To determine $c \# e$, we traverse the cone $[c]$ in reverse $<$-order. If we encounter an event $e' < e$, then no conflict can be detected by exploring e' or its causal precedessors, so we skip $[e']$. But if we encounter a condition $c' < e$ in the traversal, then we can conclude that $c \# e$ holds (because if $e' \in ([c] \cap c'^\bullet) \setminus [e]$ is the event that led us to c' in the traversal, then $c' \in {}^\bullet e' \cap {}^\bullet e''$ for some $e'' \leq e$). If we find no such c' in $[c]$, then $c \parallel e$ holds.

[1] Such a traversal is anyway necessary in most unfolding-based implementations to collect information relevant for determining which events are *cutoff points* [3, 8], so this step comes at almost no extra cost.

Table 1 Experimental results. Time and memory for PUNF and MOLE are *ratios*, see text.

Net	Unfolding		New Alg.		PUNF	MOLE	
Name	Events	Cond.	Time	Mem	Time (r)	Time (r)	Mem (r)
DPD(7)	10457	30248	0.34	9	6.59	1.76	2.18
FTP(1)	89046	178085	16.06	36	6.40	0.07	1.18
BYZ	14724	42276	0.73	21	11.48	2.66	3.15
Q(1)	7469	20969	0.21	9	6.81	2.14	2.04
ELEV(4)	16935	32354	0.50	9	5.06	0.24	1.08
BDS(1)	6330	12310	0.04	4	5.75	1.00	1.19
DME(6)	1830	6451	0.04	6	4.50	3.50	2.66
DME(7)	2737	9542	0.08	8	4.88	3.88	3.11
DME(8)	3896	13465	0.13	12	5.92	4.54	3.37
DME(9)	5337	18316	0.22	17	6.64	4.95	3.62
DME(10)	7090	24191	0.34	24	7.50	5.47	3.93
DME(11)	9185	31186	0.53	33	8.13	5.92	4.05
RW(1,2)	49179	147607	1.58	24	0.52	0.39	1.11
RW(3,1)	15401	28138	1.04	9	3.85	0.16	1.18
KEY(3)	6968	13941	0.23	5	2.52	0.30	1.03
KEY(4)	67954	135914	15.94	33	2.34	0.06	1.08
FURN(3)	25394	58897	0.69	13	3.48	1.01	1.61
FURN(4)	146606	342140	25.75	75	3.02	0.67	1.76
MMGT(3)	5841	11575	0.15	4	1.93	0.20	1.08
MMGT(4)	46902	92940	9.95	18	1.68	0.06	1.18

Notice that step II is repeated for different places p and conditions c. We make some further optimizations to avoid unnecessary repeated work during the computation for the same e:

- If we conclude that $c \parallel e$ holds for some c, we remember this information in the elements of $[c]$ (as a single bit), and any further conflict checks can safely skip $[c]$.
- If we conclude that $c \# e$ holds due to some condition $c' < e$ like above, then we propagate this information along the trail of nodes that led us from c to c'. Any further conflict checks that encounter one of those nodes can immediately stop and deduce a conflict, too.

4 Experiments

We experimentally compared our approach with other unfolding algorithms. MOLE computes a concurrency relation [3] and therefore uses quadratic memory in the worst-case. PUNF, when used with option `-n=0`, employs a linear-time exploration of $\mathcal{P}_N$ for step 3a [7]. Our implementation is based on MOLE, but we replaced MOLE's concurrency relation by our approach.

Tab. 1 summarizes our comparison on 20 classical benchmarks from the unfolding literature. For every net, we present the unfolding size together with the time (in seconds) and memory (in megabytes) of our approach, listed under 'New Alg.'. For PUNF and MOLE, the data is a *ratio relative to our approach*. Memory usage for PUNF could not readily be determined, but should be asymptotically the same as for our approach. The computed unfolding is obviously the same for the three approaches.

Quite positively, our approach runs faster than PUNF on all examples except one, with an overall running time 3.6 times smaller than that of PUNF. We interpret this as an encouraging result for our approach. Remark that PUNF implements an optimization called *prefix trees* in step 3b, which is still missing in our implementation. This technique is actually orthogonal to our contributions and seems to be particularly effective for RW(1,2). We therefore expect that our running times could be further improved in some cases by implementing prefix trees.

Also positively, our implementation consumes in average 48% of the memory that MOLE uses and still runs faster than it on roughly one half of the cases. These cases notably include all the instances of the DME series, where we obtain improved running times of up to 6 times. Here, the cost of computing the concurrency relation is actually larger than its benefit.

However, our approach is still overall slower than MOLE. Our accumulated running time is 2.3 times larger. The worst case seems to be MMGT(4), where MOLE runs 17 times faster using roughly the same memory. The concurrency relation proves to be very effective for this net, in average a condition is concurrent to only 0.2‰ of the other conditions. Compare this ratio with that of DME(11), where our approach performs 6 times faster. There, the aforementioned average is 38‰, making MOLE's approach inefficient. The same analysis holds for KEY(4), where MOLE is 16 times faster than our approach, and where the concurrency relation is even comparatively smaller than in MMGT(4).

Overall, the new approach seems to present a practical tradeoff in terms of time. For better comparison, we show only examples in which all tools terminated. However, it was already pointed out in [7] that there exist cases where the concurrency relation becomes too big to fit into memory, and approaches like PUNF and ours succeed where MOLE does not.

5 Conclusions

We presented an algorithmic improvement for the construction of Petri net unfoldings. While our implementation is still preliminary, the experimental results are promising; its running time beats the one of [7] (which also uses a linear amount of memory), and it represents an acceptable time/memory trade-off compared with [3], which uses more memory; in some instances it even performs faster.

For future work, it would be interesting to improve the implementation to incorporate the tricks from [7], which should further improve the running time. Moreover, we are interested in generalizing the approach to nets with read arcs, where it could be used as an alternative to [1] within the tool CUNF [10].

References

1 Paolo Baldan, Alessandro Bruni, Andrea Corradini, Barbara König, César Rodríguez, and Stefan Schwoon. Efficient unfolding of contextual Petri nets. *TCS*, 449:2–22, 2012.

2 Javier Esparza and Keijo Heljanko. *Unfoldings - A Partial-Order Approach to Model Checking*. EATCS Monographs in Theoretical Computer Science. Springer, 2008.

3 Javier Esparza and Stefan Römer. An unfolding algorithm for synchronous products of transition systems. In *Proc. CONCUR*, LNCS 1664, pages 2–20, 1999.

4 Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of McMillan's unfolding algorithm. *Formal Methods in System Design*, 20:285–310, 2002.

5 Keijo Heljanko. *Deadlock and reachability checking with finite complete prefixes*. Licentiate's thesis, Helsinki University of Technology, 1999.

6 Victor Khomenko. PUNF. `homepages.cs.ncl.ac.uk/victor.khomenko/tools/punf/`.

7 Victor Khomenko and Maciej Koutny. Towards an efficient algorithm for unfolding Petri nets. In *Proc. CONCUR*, LNCS 2154, pages 366–380, 2001.

8 Victor Khomenko, Maciej Koutny, and Walter Vogler. Canonical prefixes of Petri net unfoldings. *Acta Informatica*, 40(2):95–118, 2003.

9 Kenneth L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Proc. CAV*, LNCS 663, pages 164–177, 1992.

10 César Rodríguez. CUNF. `http://www.lsv.ens-cachan.fr/~rodriguez/tools/cunf/`.

11 Stefan Schwoon. MOLE. `http://www.lsv.ens-cachan.fr/~schwoon/tools/mole/`.

Constructing Attractors of Nonlinear Dynamical Systems by State Space Decomposition

Laurent Fribourg[1], Ulrich Kühne[2], and Romain Soulat[1]

1 **LSV, ENS Cachan & CNRS**
Cachan, France
2 **University of Bremen**
Bremen, Germany

Abstract

In a previous work, we have shown how to generate attractor sets of affine hybrid systems using a method of state space decomposition. We show here how to adapt the method to polynomial dynamics systems by approximating them as switched affine systems. We show the practical interest of the method on standard examples of the literature.

1998 ACM Subject Classification I.2.8, Control theory

Keywords and phrases Control theory, Hybrid Systems, Nonlinear dynamical systems

Digital Object Identifier 10.4230/OASIcs.FSFMA.2013.53

1 Introduction

The symbolic analysis of nonlinear dynamical systems has recently attracted considerable attention: the problem of computing the set of reachable states (reachability analysis) has thus been studied in [1, 3, 4, 5, 2], and the problem of computing polytopic invariants (invariant synthesis) has been studied in [10, 11, 12]. Here, we study a problem close to the problem of invariant synthesis: we want not only to generate a polytopic invariant P included in a given rectangle R, but we also want that all the trajectories starting from R converge to P. In other words, we want to construct an *attractor set* P of R, ideally as small as possible. We show that the state decomposition method given in [7] for computing attractors of linear systems can be extended to the case of polynomial dynamics, using the idea of local linearization developed in [1].

The plan of the paper is as follows. In Section 2, we recall the principles of the state space decomposition method for linear dynamical systems. In Section 3, we explain how to extend the method to polynomial dynamical systems. In Section 4, we apply the method to two standard examples of the literature. We conclude in Section 5.

2 Attractors for Linear Dynamics

We suppose that we are given a finite set $U = \{1, \ldots, N\}$ of elements called *modes*. We are also given a family of functions $\{f_u\}_{u \in U}$ with $f_u : \mathbb{R}^n \to \mathbb{R}^n$. Given a time step τ, a *sampled switched system* Σ is a dynamical system governed by an equation of the form $x(t + \tau) = f_\sigma(x(t))$, where σ is a *control signal*, which selects a mode $u \in U$ at each time step $\tau, 2\tau, \ldots$.

A *k-pattern* is a sequence of at most k modes of U. Given a set $X \subset \mathbb{R}^n$ and a mode $u \in U$, we define the *set of successors of* X *via* u, and denote by $Post_{f_u}(X)$, the set $\{x' \in \mathbb{R}^n \mid f_u(x) = x' \text{ for some } x \in X\}$. Given a pattern π of the form $(u_1 \cdot u_2 \cdot \cdots \cdot$

1st French Singaporean Workshop on Formal Methods and Applications 2013 (FSFMA'13).
Editors: Christine Choppy and Jun Sun; pp. 53–60

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

u_m), the *set of successors of* X *via* π, denoted by $Post_{f_\pi}(X)$, is given by: $Post_{f_\pi}(X) = Post_{f_{u_m}}(\cdots(Post_{f_{u_2}}(Post_{f_{u_1}}(X)))\cdots)$.

Suppose that we are given a *box* $R \subseteq \mathbb{R}^n$ (i.e., a cartesian product of closed intervals). We have given in [7] a general method in order to show the *controlled invariance* of Σ in R. By controlled invariance in R, we mean that if the system state is in R at some time, it will stay forever in R under the control of an appropriate signal σ. The method constructs a *k-decomposition* of R, that is, a set Δ of the form $\{(V_i, \pi_i)\}_{i\in I}$, where I is a finite set of indices, the V_is are sub-boxes of R, and the π_is are k-patterns. Furthermore, this decomposition Δ is *k-invariant* in the sense:

1. $\bigcup_{i\in I} V_i = R$
2. $Post_{f_{\pi_i}}(V_i) \subset R$, for all $i \in I$.

An algorithm of decomposition is given in [7], and is recalled in Appendix A: given a dynamical system $\{f_u\}_{u\in U}$ and a box R, it returns a k-invariant decomposition Δ of R.

▶ **Lemma 1.** *If* $\Delta = \{(V_i, \pi_i)\}_{i\in I}$ *is a k-invariant decomposition of* R*, then:*

$$Post_\Delta(R) \subset R,$$

where the operator $Post_\Delta$ *is defined, for all* $X \subset \mathbb{R}^n$*, by:*

$$Post_\Delta(X) = \bigcup_{i\in I} Post_{f_{\pi_i}}(X \cap V_i)$$

▶ **Lemma 2.** *Consider a k-invariant decomposition* $\Delta = \{(V_i, \pi_i)\}_{i\in I}$ *of* R*. The sequence* $\{R^j_\Delta\}_{j\geq 0}$ *defined by:*

- $R^0_\Delta = R$,
- $R^{j+1}_\Delta = Post_\Delta(R^j_\Delta)$

is a decreasing nested sequence and the set $R^*_\Delta = \bigcap_{j\geq 0} R^j_\Delta$ *is well-defined. Furthermore,* R^*_Δ *is an* attractor set *of* R*, i.e.:*

1. $Post_\Delta(R^*_\Delta) = R^*_\Delta$ *(*invariance*)*
2. $\forall x \in R,\ d(Post^j_\Delta(x), R^*_\Delta) \to 0$ *as* j *tends to* ∞[1] *(*attractivity*).*

Attractors and limit cycles have been studied in the context of affine dynamics in [6].

3 Nonlinear Dynamics

The decomposition procedure, explained in Section 2, is quite general, and does not suppose that the functions f_u are linear or affine. However, in the case where f_u is an affine function, the computation of the successor sets (via *Post* operator) can be done in an *exact* manner.

We now explain how to apply the state space decomposition procedure in the case of non-affine dynamics. This is done at the price of an *over-approximation* of the successor sets. Following [1], we compute (an overapproximation of) the successor sets using *local linearizations* of the system, and enlargement of the linear images by addition of *error* intervals. We will consider a system governed by a *unique* equation of the form $x(t+\tau) = f(x(t))$ where f is a polynomial. The set U is thus reduced to a single element ($U = \{1\}$). A pattern π_i associated to a subregion V_i, is now just an *integer* indicating the number of times the (local linearization of) f should be applied when the state is in V_i.

[1] d is the distance between a point and a subset of $\mathbb{R}^n$

3.1 Affine systems with uncertainty

As in [1], reachable sets are represented here by zonotopes. They are chosen because linear transformations and Minkowski sums[2] can be computed efficiently, allowing to compute reachable sets for large scale linear systems in continuous space. A zonotope is defined by a center c to which linear segments $l_i = \beta^{(i)} \cdot g^{(i)}$, $-1 \leq \beta^{(i)} \leq 1$ are added via Minkowski sum.

▶ **Definition 3.** A *zonotope* is a set

$$Z = \{x \in \mathbb{R}^n : x = c + \Sigma_{i=1}^{p} \beta^{(i)} \cdot g^{(i)},\ -1 \leq \beta^{(i)} \leq 1\}$$

with $c, g^{(1)}, \ldots, g^{(p)} \in \mathbb{R}^n$. The vectors $g^{(1)}, \ldots, g^{(p)}$ are referred to as the *generators* and c as the *center* of the zonotope. It is convenient to represent the set of generator as a matrix G. The notation is $< c, G >$, where the first element refers to the center of the zonotope and the second to the generators.

Zonotopes allow to extend easily the decomposition procedure in order to take into account small perturbations of the system dynamics (see [8]). Suppose that we the system is described by an equation of the form

$$x(t + \tau) = f_{lin}(x(t)) + \varepsilon$$

where:

- f_{lin} is an affine function defined by $f_{lin}(x) = Ax + b$ with $A \in \mathbb{R}^{n \times b}$, $b \in \mathbb{R}^n$
- ε is a disturbance vector belonging to a rectangle region $\Lambda = [-\varepsilon_1, +\varepsilon_1] \times \cdots [-\varepsilon_n, +\varepsilon_n]$ of $\mathbb{R}^n$, with $\varepsilon_i \geq 0$ for all i.

Since Λ is a product of intervals centered in 0, it can be written as a zonotope

$$Z_\Lambda =< 0, G_\Lambda > \text{ with } G_\Lambda = \begin{pmatrix} \varepsilon_1 & 0 & \ldots & 0 \\ 0 & \varepsilon_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & \varepsilon_n \end{pmatrix}$$

▶ **Lemma 4.** *Consider a zonotope $Z =< c, G >$ with G a square matrix, a box $\Lambda = [-\varepsilon_1, +\varepsilon_1] \times \cdots [-\varepsilon_n, +\varepsilon_n]$ of $\mathbb{R}^n$, and a function f defined by:*
$f(x) = Ax + b + \varepsilon$, *with $\varepsilon \in \Lambda$.*
We have: $Post_f(Z) \subset < Ac + b, AG + G_\Lambda >$.

3.2 Linearization of nonlinear dynamics

Consider now a system governed by equation $x(t + \tau) = f(x(t))$ where f is a polynomial. We can write:

$$f(x) = f_{lin}(x) + P(x),$$

where $f_{lin}(x)$ corresponds to the polynomial subpart of order 1, and P to the polynomial of order greater than or equal to 2. We can then apply the method explained in Section 3.1, by computing a local over-approximation Λ of $P(x)$.

[2] The Minkowski of two sets A, B is defined by $A + B = \{a + b \mid a \in A, b \in B\}$

▶ **Lemma 5.** *Consider a function f defined by: $f(x) = f_{lin}(x) + P(x)$, where $f_{lin}(x)$ is a 1st-order polynomial of the form $b + Ax$, and $P(x)$ a 2nd-order polynomial. Given a zonotope $Z :< c, G >$, we have:*

$$Post_f(Z) \subset Post_{f_{lin}}(Z) + Z_\Lambda$$

with:

- $Post_{f_{lin}}(Z) =< f(c), AG >$
- $Z_\Lambda =< 0, \begin{pmatrix} \varepsilon_1(Z) & 0 & \dots & 0 \\ 0 & \varepsilon_2(Z) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \varepsilon_n(Z) \end{pmatrix} >$

with $(1 \leq i \leq n)$: $\varepsilon_i(Z) = max_{x \in Z}(|P_i(x) - P_i(c)|)$.

Now, in order to apply the decomposition procedure (extended with error), we just have to find an upper bound for $|P(x) - P(c)|$ componentwise. In the following, we explain on two standard examples how to compute such upper bounds. Then we apply the decomposition procedure in order to find a decomposition Δ, and construct an attractor related to R^*_Δ.

4 Case studies

These examples are taken from [2]. Given a zonotope $Z =< c, G >$, we explain how to compute $Post_{f_{lin}}(Z)$ and Z_Λ appearing in Lemma 5. Experiments have been performed with the tool MINIMATOR [9] on a machine equipped with an Intel Core2 at 2.93GHz and 2 GB of RAM memory.

4.1 Van der Pol oscillator

4.1.1 Dynamics

The dynamics of the Van der Pol oscillator are the following:

$$x(\tau) = \begin{pmatrix} 1 & \tau \\ -\tau & 1+\tau \end{pmatrix} x(0) + \begin{pmatrix} 0 \\ -x_1(0)^2 x_2(0)\tau \end{pmatrix}.$$

When linearized to a point $c \in \mathbb{R}^2$, this gives:

$$x(\tau) = \begin{pmatrix} 1 & \tau \\ -\tau & 1+\tau \end{pmatrix} x(0) + \begin{pmatrix} 0 \\ -c_1^2 c_2 \tau \end{pmatrix}.$$

Thus, we have $Post_{f_{lin}}(Z) = \begin{pmatrix} 1 & \tau \\ -\tau & 1+\tau \end{pmatrix} Z + \begin{pmatrix} 0 \\ -c_1^2 c_2 \tau \end{pmatrix} = \begin{pmatrix} 1 & \tau \\ -\tau & 1+\tau(1-c_1^2) \end{pmatrix} x(0)$. It is easy to see that for a box $V \subset \mathbb{R}^2$ we are making an error of at most 0 on the x axis and $|(c_1^2 - (c_1 + G_{1,2} + G_{2,2})^2|\tau$ on the y axis, when $Z =< c, G >$ with $c = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}$ and $G = \begin{pmatrix} G_{1,1} & G_{1,2} \\ G_{2,1} & G_{2,2} \end{pmatrix}$ Thus we need to enlarge any image of a zonotope $Z =< c, G >$ by 0 on the x-axis and $\tau|(C_1^2 - (C_1 + G_{1,2} + G_{2,2})^2|$ on the y-axis (i.e., $Z_\Lambda =< 0, \begin{pmatrix} 0 & 0 \\ 0 & |(C_1^2 - (C_1 + G_{1,2} + G_{2,2})^2|\tau \end{pmatrix} >$).

4.1.2 Attractor Construction

The Decomposition procedure is applied to $R = [-3, 3] \times [-3, 3]$ and $\tau = 0.01$ (with parameters $k = 30$, $d = 7$). At boxes located around the center of R, the length of patterns is 1 while in

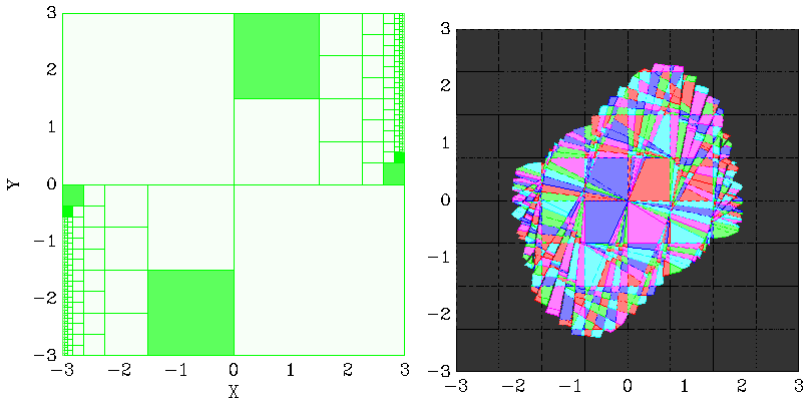


Figure 1 Decomposition for the Van der Pol oscillator (left) ; R^j_Δ for $j = 30$ (right).

the lower left and upper right edges, the length is up to 30. The result of the Decomposition is depicted in the left part of Figure 1 and the attractor set R^*_Δ in the right part. Experiments took 8 minutes to complete.

4.2 FitzHugh-Nagumo Neuron

4.2.1 Dynamics

The dynamics of the FitzHugh-Nagumo neuron are the following:

$$x(\tau) = \begin{pmatrix} 1+\tau & -\tau \\ 0.08\tau & -0.0064\tau + 1 \end{pmatrix} x(0) + \begin{pmatrix} -x_1(0)^3\tau/3 + 0.875\tau \\ 0.056\tau \end{pmatrix}$$

When linearized to a point $c \in \mathbb{R}^2$, this gives:

$$x(\tau) = \begin{pmatrix} 1+\tau & -\tau \\ 0.08\tau & -0.0064\tau + 1 \end{pmatrix} x(0) + \begin{pmatrix} -c_1^3\tau/3 + 0.875\tau \\ 0.056\tau \end{pmatrix}$$

It is easy to see that for a box $V \subset \mathbb{R}^2$ we are making an error of at most $max_{x\in V}(\frac{|x_1^3 - c_1^3|}{3})\tau$ on the x axis and 0 on the y axis. Thus we need to enlarge any image of a zonotope Z by $max_{x\in Z}(\frac{|x_1^3 - c_1^3|}{3})\tau$ on the x-axis and 0 on the y-axis ($Z_\Lambda =< 0, \begin{pmatrix} max_{x\in Z}(\frac{|x_1^3 - c_1^3|}{3})\tau & 0 \\ 0 & 0 \end{pmatrix} >$).

4.2.2 Attractor construction

The Decomposition procedure is applied to $R = [-2.5, 2.5] \times [-0.5, 2.5]$ and $\tau = 0.1$ (with parameters $k = 30$, $d = 7$). For boxes located around the center of R, the length of patterns is 1 while in the lower left and upper right corners, the length is up to 22. The result of the Decomposition is depicted in the left part of Figure 2 and the attractor set R^*_Δ in the right part. Experiments took 5 minutes to complete.

5 Future work

We have explained how to construct attractors of polynomial dynamical systems by extending a method designed for linear dynamical systems. The method consists in considering the subpolynomial subpart of order greater than 1 as a perturbation that is over-approximated.

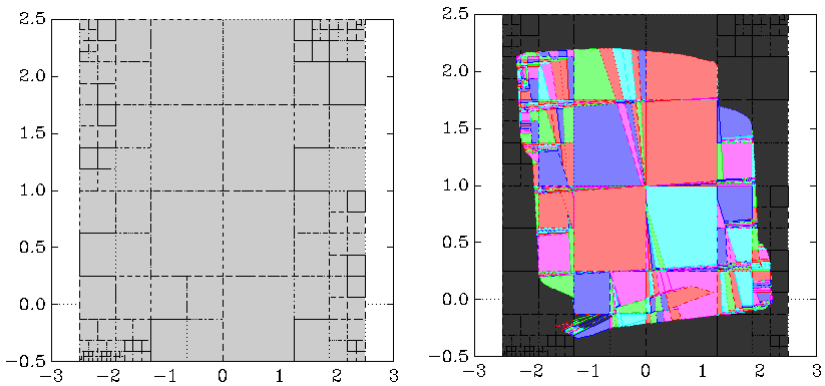


Figure 2 Decomposition for the FitzHugh-Nagumo Neuron (left) ; R^j_Δ for $j = 30$ (right).

So far, the over-approximation is done in an *ad hoc* fashion for each specific example. For future work, we plan to consolidate the method by using the formal technique of linearization of [1], based on the notion of Lagrange remainder.

A Appendix: Decomposition Algorithm

The Decomposition procedure generates a k-invariant decomposition of R, as follows:

It first calls sub-procedure Find_Pattern in order to get a k-pattern such that R is R-invariant. If it succeeds, then it is done. Otherwise, it divides R into 2^n sub-boxes $V_1, \ldots, V_{2^n}$ of equal size. If for each V_i, Find_Pattern gets a k-pattern making it R-invariant, it is done. If, for some V_j, no such pattern exists, the procedure is recursively applied to V_j. It ends with success when a k-invariant decomposition of R is found, or failure when the maximal degree d of decomposition is reached.

The algorithmic form of the procedure is given in Algorithms 1 and 2. (For the sake of simplicity, we consider the case of dimension $n = 2$, but the extension to $n > 2$ is straightforward.) The main procedure Decomposition(W,R,D,K) is called with R as input value for W, d for input value for D, and k as input value for K; it returns either $\langle \{(V_i, \pi_i)\}_i, True \rangle$ with $\bigcup_i V_i = W$ and $\bigcup_i Post_{\pi_i}(V_i) \subseteq R$, or $\langle \_, False \rangle$. Procedure Find_Pattern(W,R,K) looks for a K-pattern for which W is R-invariant: it selects all the K-patterns (which are in finite number) by non-decreasing length order until either it finds such a pattern π (output: $\langle \pi, True \rangle$), or no one exists (output: $\langle \_, False \rangle$).

The correctness of the procedure is stated as follows.

▶ **Theorem 6.** *If Decomposition(R,R,d,k) returns $\langle \Delta, True \rangle$, then Δ is a k-invariant decomposition of R.*

Algorithm 1: Decomposition(W,R,D,K)

Input: A box W, a box R, a degree D of decomposition, a length K of pattern
Output: $\langle\{(V_i, \pi_i)\}_i, True\rangle$ with $\bigcup_i V_i = W$ and $\bigcup_i Post_{\pi_i}(V_i) \subseteq R$, or $\langle \_, False\rangle$

$(\pi, b) := Find\_Pattern(W, R, K)$
if $b = True$ **then**
  return $\langle\{(W, \pi)\}, True\rangle$
else
  if $D = 0$ **then**
    return $\langle \_, False\rangle$
  else
    Divide equally W into (W_1, W_2, W_3, W_4) `/* (case` $n = 2$`)` `*/`
    (Δ_1, b_1) := Decomposition(W_1,R,$D-1$,K)
    (Δ_2, b_2) := Decomposition(W_2,R,$D-1$,K)
    (Δ_3, b_3) := Decomposition(W_3,R,$D-1$,K)
    (Δ_4, b_4) := Decomposition(W_4,R,$D-1$,K)
    return $(\Delta_1 \cap \Delta_2 \cap \Delta_3 \cap \Delta_4, b_1 \wedge b_2 \wedge b_3 \wedge b_4)$

Algorithm 2: Find_Pattern(W,R,K)

Input: A box W, a box R, a length K of pattern
Output: $\langle \pi, True\rangle$ with $Post_\pi(W) \subseteq R$, or $\langle \_, False\rangle$ when no pattern maps W into R

for $i = 1 \ldots K$ **do**
  Π := set of patterns of length i
  while Π *is non empty* **do**
    Select π in Π
    $\Pi := \Pi \setminus \{\pi\}$
    if $Post_\pi(W) \subseteq R$ **then**
      return $\langle \pi, True\rangle$
return $\langle \_, False\rangle$

References

1 M. Althoff, O. Stursberg, and M. Buss. Reachability analysis of nonlinear systems with uncertain parameters using conservative linearization. In *CDC*, pages 4042–4048. IEEE, 2008.

2 M. Amin Ben Sassi, R. Testylier, T. Dang, and A. Girard. Reachability analysis of polynomial systems using linear programming relaxations. In *ATVA*, pages 137–151, 2012.

3 Eugene Asarin, Thao Dang, and Antoine Girard. Reachability analysis of nonlinear systems using conservative approximation. In Oded Maler and Amir Pnueli, editors, *HSCC*, volume 2623 of *Lecture Notes in Computer Science*, pages 20–35. Springer, 2003.

4 Thao Dang, Colas Le Guernic, and Oded Maler. Computing reachable states for nonlinear biological models. In *CMSB*, pages 126–141, 2009.

5 G. Frehse. PHAVer: algorithmic verification of hybrid systems past HyTech. *STTT*, 10(3):263–279, 2008.

6 L. Fribourg and R. Soulat. Limit cycles of controlled switched systems: Existence, stability, sensitivity. In *Proc. 3rd NCMIP 2013 IOP Publishing "Journal of Physics: Conference Series"*, May 2013. To appear.

7 Laurent Fribourg and Romain Soulat. Finite controlled invariants for sampled switched systems. Research Report LSV-13-09, Laboratoire Spécification et Vérification, ENS Cachan, France, April 2013. 27 pages.

8 W. Kühn. Zonotope dynamics in numerical quality control. *Mathematical Visualization*, pages 125–134, 1998.

9 Minimator Team. Minimator Web page. http://www.lsv.ens-cachan.fr/ soulat/minimator/, 2013.

10 Mohamed Amin Ben Sassi and Antoine Girard. Controller synthesis for robust invariance of polynomial dynamical systems using linear programming. *CoRR*, abs/1107.1580, 2011.

11 Mohamed Amin Ben Sassi and Antoine Girard. Computation of polytopic invariants for polynomial dynamical systems using linear programming. *Automatica*, 48(12):3114–3121, 2012.

12 Mohamed Amin Ben Sassi and Antoine Girard. Control of polynomial dynamical systems on rectangles. In *European Control Conference*, 2013.

Formal Modelling and Verification of Pervasive Computing Systems

Yan Liu

National University of Singapore
yanliu@comp.nus.edu.sg

Abstract

Pervasive computing (PvC) systems are emerging as promising solutions to many practical problems, e.g., elderly care in home. However, such systems have long been developed without sufficient verification. Formal methods, eps. model checking are sound techniques for complex system analysis regarding correctness and reliability requirements. In this work, a formal modelling framework is proposed to model the general the system design (e.g., concurrent communications) and the critical environment inputs (e.g., human behaviours). Correctness requirements are specified in formal logics which are automatically verifiable against a system model. Furthermore, Markov Decision Processes (MDPs) are adopted for modelling probabilistic behaviours of PvC systems. Three problems are analysed which are overall reliability prediction based on component reliabilities, reliability distribution w.r.t., how reliable should the component be to reach an overall reliability requirement and sensitivity analysis w.r.t., how does a component reliability affect the overall reliability. Finally, the usefulness of our approaches are demonstrated on a smart healthcare system with unexpected bugs and system flaws exposed.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases System Analysis, Formal Modelling and Verification, Reliability Analysis

Digital Object Identifier 10.4230/OASIcs.FSFMA.2013.61

1 Introduction

Many problems arise with the proliferation of ageing population in all industrialised societies, e.g., creating enormous costs for elders' intensive care. The context-aware and self-adaptive PvC [11] system enables their independent living with little supervision [7]. Therefore PvC systems are safety critical and should be verified before deployment. However, traditional techniques such as simulation and testing are expensive and not complete. Formal methods instead, especially model checking [4] techniques are promising solutions for their expressive system modelling and exhaustive verification. Thus, we propose to apply these techniques to formally analyse PvC systems.

Motivation. PvC systems are inherently complex making it a challenging task to perform system analysis. They are usually composed of a physical layer with sensors to monitor the environment changes; a middleware layer to manage and reason about the sensed contexts with predefined rules; an application layer to make adaptations, as shown in Fig. 1a. Failures happen with various reasons like a wrong reminder could be caused by a sensor failure or an incorrect rule [5]. In practice, such faults could only be exposed during deployment as it is impossible to capture all scenarios at development phase. Thus, there is a need of a systematic and complete analysis approach. Furthermore, a PvC system is probabilistic due to limited reliability of its components [8]. It is essential to manage the system reliability at

1st French Singaporean Workshop on Formal Methods and Applications 2013 (FSFMA'13).
Editors: Christine Choppy and Jun Sun; pp. 61–67

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

an acceptable rate. Besides, the system behaviours are nondeterministic for unpredictable user activities. Thus, we choose MDPs-based verification technique for reliability analysis.

Our **contributions** are two-fold. Firstly, a formal modelling framework is proposed with modelling patterns for common features of PvC systems, e.g., compositional architecture and concurrent interactions. Critical requirements of stakeholders are specified as desirable properties (safety and liveness) and testing purposes (conflict cases). Case study on a smart healthcare system revealed multiple bugs such as conflict reminders. Secondly, reliability models of PvC systems are constructed using MDPs upon which three general problems are investigated: 1) "What is the overall system reliability given the component reliabilities?"' refers to *reliability prediction*. 2) "What is the reliability required on components for an expected reliability on overall system?" refers to *reliability distribution*. 3) "What is the most critical part contributes to the system reliability?" refers to *sensitivity analysis*. Experiments on AMUPADH system shows that the overall system reliability is below 50%.

2 A running example – AMUPADH

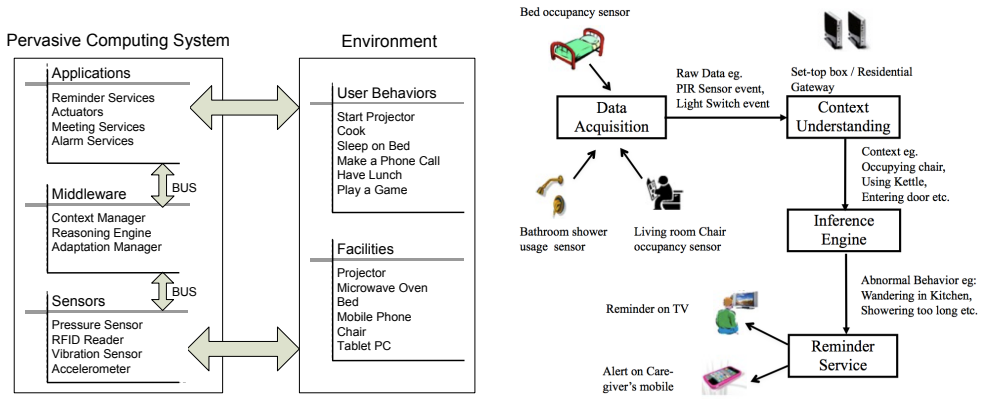
AMUPADH [2] is a smart healthcare system providing assistance to elderly people who have difficulties in remembering activities of daily living (ADLs). It is deployed in a nursing home, PeaceHeaven[1] for a six-month real life trial. The workflow in Fig. 1b consists of:

- **Step 1: Data Acquisition.** Multiple sensors are deployed to monitor the environment such as when someone turns on the shower tap, the shake sensor is triggered and a *Unstationary* signal is sent to the system.
- **Step 2: Context Processing and Reasoning.** Sensor signals are interpreted to low-level contexts like "Tap turned on" in the inference engine, Drools[2]. By evaluating predefined reasoning rules (written in first order logic), high-level contexts such as "Showering too long" are generated.
- **Step 3: Reminder Service Rendering.** If an abnormal activity is recognised, the system will render a reminder service to help the elder. For example, a bluetooth speaker will play a voice reminder upon a error message. A number of devices like a TV or iPad are used to prompt reminders.

3 Correctness Analysis of PvC Systems

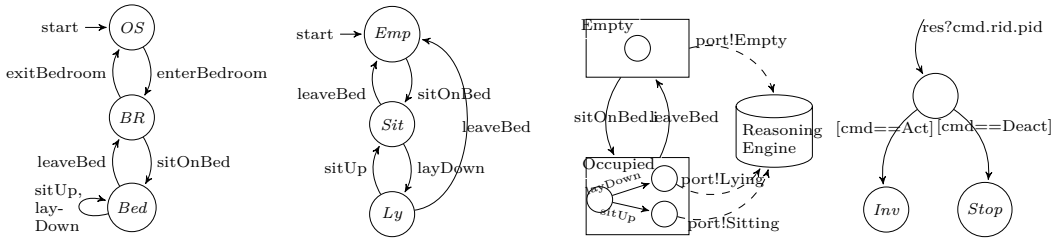
A Formal Modelling Framework. According to the general structure shown in Fig. 1a, we propose to model the system design and environment input separately. *Modeling Environments:* PvC systems are user centric, thus modelling the user behaviours is essential. However it is difficult because user behaviours are unpredictable. As suggested by domain experts, such systems usually target at a determined group of activities whose sequences remain unpredictable. In concurrent modelling languages, nondeterministic choices can be used to enumerate the sequences, an example is illustrated in Fig. 2a. At a location, each possible move of the user is modelled as a choice. An unrealistic scenario may arise that the model allows an activity to be performed repeatedly e.g., the user sits on the bed again and again without standing up. To eliminate such cases, we need a constraint model e.g., a bed model (Fig. 2b, no more sitting is allowed once the bed is occupied). *Modelling System*

[1] Located at 9 Upper Changi Road North, Singapore, 507706. Tel: +65-65465678.
[2] Drools Expert: http://www.jboss.org/drools/drools-expert.html



(a) Common Architectures **(b)** An example PvC System, AMUPADH

Figure 1 Introduction of Pervasive Computing (PvC) Systems.



(a) User Behaviours **(b)** Bed Behavior **(c)** Bed Pressure Sensor **(d)** Service Rendering

Figure 2 Modelling a PvC System.

Design: In PvC systems, communication and cooperation of components are most critical. The sensing behaviour of sensors is a concurrent happening with the detectable event in the environment. Thus, sensor models need to be paralleled with environment model such that they are synchronised on common events. As shown in Fig. 2a, 2b and 2c, a sitOnBed event will trigger three models to progress simultaneously. Additionally, the synchronised channel models the message sending from sensors to reasoning engine in negligible time [6]. In the middleware, context managing and reasoning are performed. They are modelled as data operations on global context variables and conditional statements respectively. At the application layer, services are rendered upon decoding of the messages. As shown in Fig. 2d, the reminding system is modelled using channel communications, shared variables and guarded processes. *Compose A Complete Model:* Finally, component models should be composed using sequential, interleave or parallel patterns according to their relations.

Formal Specification of Critical Requirements. Critical requirements from the stakeholders are identified as desirable properties and testing purposes. *Deadlock freeness* (DF) and *Guaranteed reminder services* (GR) properties are desirable which respectively requires the system has no dead state where no more actions can be performed and services should be provided at the right moment for the right user. For instance, a reminder should be sent to a patient whenever he is wandering somewhere is formalised as "$\Box$(Wandering $\rightarrow$ $\Diamond$RemindLeave)". It is also helpful to test the common problems i.e.,

Table 1 Results of Correctness Analysis Experiment.

Property	Result	# States	# Transitions	Time(s)	Bug?
$DF_{Complete}$	–	–	–	OOM	No
$DF_{Bedroom}$	True	1.43M	2.04M	815	No
$DF_{Washroom}$	True	10.8M	15.8M	7045	No
$GR_{UsingWrongBed(UWB)}$	True	1.60M	2.43M	1945	No
$GR_{TapNotOff(TNO)}$	False	0.07M	0.131M	39	Yes
$GR_{WanderInWashroom(WiW)}$	False	2.19M	4.53M	12414	Yes
$GR_{ShowerNoSoap(SNS)}$	False	0.832M	1.66M	729	Yes
$GR_{ShowerTooLong(STL)}$	False	4314	5150	1.6	Yes
$GR_{SitBedTooLong(SBTL)}$	True	1.58M	2.38M	1913	Yes
Inconsistency	True	572	745	0.3	Yes
Conflict Reminder	True	2446	3036	1.11	Yes
False Alarm	True	0.01M	0.02M	6.1	Yes

system inconsistency and *conflicting/ false services*. Both of them can be specified as reachability properties that are verified by checking if there is a state where system knowledge contradicts with actual environment and a state where two conflict services are invoked/ where a service is rendered for a wrong user respectively.

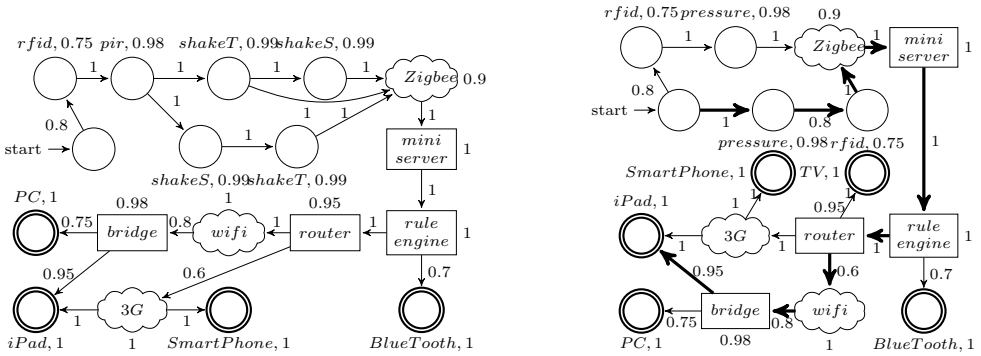
Case Study on AMUPADH System. In the experiment, we implement the modelling framework in CSP# language [9] and run verification by PAT model checker [10]. In Table 1[3], violation of guaranteed reminder(GR) properties reveals a design flaw i.e., inefficient update of the patient's location. An inconsistent state is found i.e., the patient's location context variable remains to be inside washroom even after he has left. The case study shows the usefulness of our approach in analysing PvC system with the counterexamples help in efficient system debugging. It is also observed that the state space reaches the limits of the model checker. Thus, a future direction is to explore state space reduction techniques.

4 Reliability Analysis using MDPs

System Modelling in MDPs. MDPs allow us to model both probabilistic and nondeterministic behaviours. In general, nondeterministic choice is adopted when no definitive information is given for resolving the choice. In Fig. 3a, it is used to model transitions between sensors because of the randomness of user behaviours. *States* are abstract nodes of sensors, software components and network devices associated with a target scenario while double circled nodes are goals. In PvC systems, there are two types of *transitions* which are the happen-before relation among sensors and message passing directions among the others. *Labels* denote the reliability values of a node or a transition which are usually provided by system engineers estimated from exemplar system runs.

Reliability Analysis Approaches. It consists of three parts, Fig. 4 (a) shows *reliability prediction* which calculates the reachable probability, $Pr(M, s)$ from an initial state to a goal state s on an MDP model M. A reliability range i.e., max. and min. reachability is produced since multiple reachable paths (aka. schedulers) are created due to nondeterminism. *Reliability distribution* calculation (Fig. 4 (b)) takes two inputs: (1) a reliability requirement R on the overall system; (2) a parameterised MDP model M with weights $w_i x$ (denotes the

[3] The test bed is a PC with Intel Xeon CPU at 2.13GHz and 32GB RAM. OOM stands for out of memory.



(a) Tap-Not-Off (TNO) Model **(b)** Using-Wrong-Bed (UWB) Model

Figure 3 MDP models for Scenario TNO and UWB.

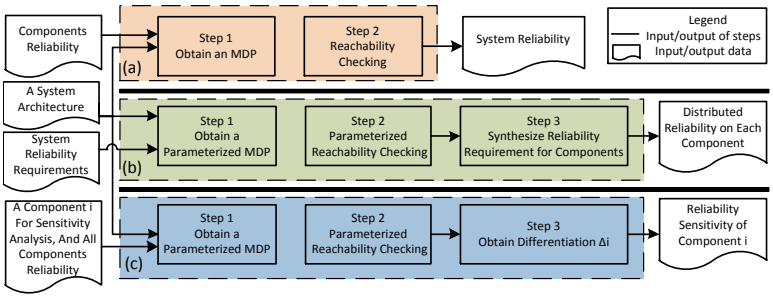


Figure 4 Workflow: (a) reliability prediction; (b) reliability distribution; (c) sensitivity analysis.

reliability of component x has a weight w_i). Given a scheduler δ, we can obtain the system reliability (i.e., $Pr(M, s)$) as a polynomial function of x only. Then the Newton's method is used to calculate the lower bounds on x for finitely many schedulers [1] among which the maximum value gives us the minimum requirement on component reliability. *Sensitivity analysis* is shown in Fig. 4 (c). The sensitivity s_i of the i^{th} component's reliability R_i is defined as a partial derivation (denoted by f w.r.t. R_i) of system reliability R, denoted as $\Delta_i = \frac{\delta f(R_1, R_2, ...R_i, ...R_n)}{\delta R_i}$. In this work, we investigate one component each time that the formula is then reduced to $\Delta_i = \frac{\delta V(init)}{\delta R_i}$ ($V(init)$ is obtained via reliability distribution).

Case Study on AMUPADH system. Six scenarios that need reminders are modelled similarly with Fig. 3. These MDPs models are then analysed using the model checker RaPid [3]. Reliability of these reminders ranges from 0.25 to 0.4 (Table 2a) which is quite low. It is because the RFID readers depend on the wearable tags that the patients throw away from time to time. Furthermore, Table 2b shows the network nodes need a reliability 0.913 for all the scenarios to achieve a system reliability of 0.4. For the requirement of 0.5, it is impossible to distribute. It's because the system cannot differentiate who is sitting on the bed that the SBTL reminder is sent to the wrong person half of the time. As for *sensitivity analysis*, we demonstrate one scheduler in UWB scenario (highlighted path in Fig. 3b). Fig. 5a suggests improvement on RFID and Wi-Fi nodes gains higher system reliability than Zigbee node. Furthermore, when their reliability reaches 0.7, improving Wi-Fi nodes is more efficient than others (Fig. 5b). These experiments give a good estimation and useful guidance on improving the system reliability, especially in budget concerned systems.

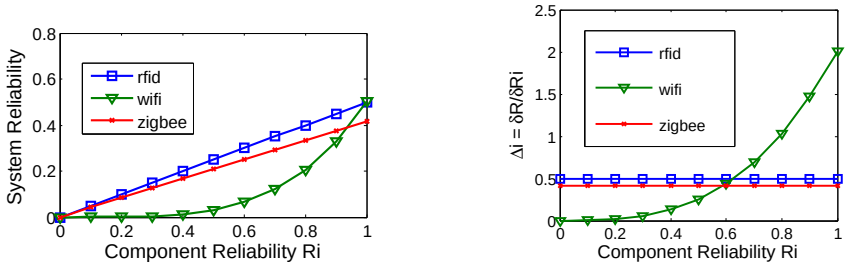
Table 2 Experiments of Reliability Prediction and Distribution Analysis.

(a) Reliability Prediction

Rel.	UWB	SBTL	SNS	STL	TNO	WiW
Scedulers	32	24	32	16	64	16
Max	0.374	0.419	0.367	0.371	0.371	0.371
Min	0.296	0.246	0.290	0.292	0.290	0.292
Time	<1 ms					

(b) Reliability Distribution

Req.	Nodes	UWB	SBTL	SNS	STL	TNO	WiW
0.4	Network	0.854	0.904	0.913	0.911	0.911	0.911
	Sensor	0.886	0.938	0.941	0.923	0.923	0.923
0.5	Network	0.914	-	0.965	0.963	0.963	0.963
	Sensor	0.996	-	0.995	0.994	0.994	0.994
Time(s)		3.45	2.68	3.86	1.87	11.00	2.35



(a) Distribution **(b)** Sensitivity

Figure 5 Using Wrong Bed (UWB)- Sensitivity Analysis on Nodes.

5 Conclusion

In this paper, we demonstrate the approaches of formally analysing PvC systems using model checking techniques, i.e., a formal modelling framework is proposed for correctness analysis; an MDPs-based approach for reliability analysis w.r.t., reliability prediction, distribution and sensitivity analysis. In future, we intend to develop algorithms to alleviate the state space explosion problem.

Acknowledgements. Supervisor: Dr. Jin Song Dong.

References

1 C. Baier and J. Katoen. *Principles of Model Checking*. The MIT Press, 2008.
2 J. Biswas, M. Mokhtari, J. S. Dong, and P. Yap. Mild dementia care at home - integrating activity monitoring, user interface plasticity and scenario verification. In *ICOST*, pages 160–170, 2010.
3 L. Gui, J. Sun, Y. Liu, Y. Si, J. S. Dong, and X. Wang. Combining model checking and testing with an application to reliability prediction and distribution. In *ISSTA*, Accepted 2013.
4 E. M. Clarke Jr., O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.
5 V. Lee, Y. Liu, X. Zhang, C. Phua, K. Sim, J. Zhu, J. Biswas, J. S. Dong, and M. Mokhtari. Acarp: Auto correct activity recognition rules using process analysis toolkit (pat). In *ICOST*, pages 182–189. 2012.

6 Y. Liu, X. Zhang, J. S. Dong, Y. Liu, J. Sun, J. Biswas, and M. Mokhtari. Formal analysis of pervasive computing systems. In *ICECCS*, pages 169 –178, 2012.

7 J. Nehmer, M. Becker, A. Karshmer, and R. Lamm. Living assistance systems: an ambient intelligence approach. In *ICSE*, pages 43–50, 2006.

8 M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Pers. Commun.*, 8:10–17, 2001.

9 J. Sun, Y. Liu, J. S. Dong, and C. Q. Chen. Integrating specification and programs for system modeling and verification. *TASE*, pages 127–135, 2009.

10 J. Sun, Y. Liu, J. S. Dong, and J. Pang. Pat: Towards flexible verification under fairness. In *CAV*, pages 709–714, 2009.

11 M. Weiser. The computer for the 21st century. *Scientific American*, 265(3):66–75, 1991.

Illustrating the *Mezzo* Programming Language

Jonathan Protzenko

INRIA
Rocquencourt, France
jonathan.protzenko@ens-lyon.org

Abstract

When programmers want to prove strong program invariants, they are usually faced with a choice between using theorem provers and using traditional programming languages. The former requires them to provide program proofs, which, for many applications, is considered a heavy burden. The latter provides less guarantees and the programmer usually has to write run-time assertions to compensate for the lack of suitable invariants expressible in the type system.

We introduce *Mezzo*, a programming language in the tradition of ML, in which the usual concept of a type is replaced by a more precise notion of a permission. Programs written in *Mezzo* usually enjoy stronger guarantees than programs written in pure ML. However, because *Mezzo* is based on a type system, the reasoning requires no user input. In this paper, we illustrate the key concepts of *Mezzo*, highlighting the static guarantees our language provides.

1998 ACM Subject Classification D.3.2 Applicative (functional) languages

Keywords and phrases Type system, Language design, ML, Permissions

Digital Object Identifier 10.4230/OASIcs.FSFMA.2013.68

1 Introduction

Type systems help programmers reason about the types of the manipulated objects, which embed information about their memory structure. Programs which obey a strong static discipline, such as that of ML, therefore have the powerful property that that they cannot go wrong. In other words, a well-typed program will not lead to a segmentation fault.

In practice, programmers want to reason beyond the memory layout of objects. Real-world objects often follow a protocol, going through different *states*, that only permit certain operations. A file descriptor starts *uninitialized*, then it may move to *ready*, before being *closed*. The "open" operation may only be performed on an uninitialized file descriptor, while the "close" operation only works when the file descriptor is ready. Thus, the file descriptor changes *states*, while preserving its *type*. However, traditional type systems fail to help programmers statically check *state* invariants.

Mezzo [6] is a programming language that reads and feels like ML, but that is equipped with a more powerful type system, which attempts to answer the above concerns. Since *Mezzo* has a more rigid typing discipline than ML, some programs that previously type-checked in ML will be deemed too unsafe. Conversely, as *Mezzo* allows more precise reasoning, some programs that previously could not be type-checked in ML will be understood.

In *Mezzo*, the notion of state and that of a type are conflated. An object which moves from a state to another is an object whose type changes. For instance, the "open" operation will change the type of its argument from *uninitialized* to *ready*. This design choice requires careful reasoning about *ownership*. Indeed, it is crucial that no other part of the system sees the object with its previous type, as this would naturally lead to an inconsistency, and protocol violations. Therefore, the type system should track ownership and avoid undesired aliases, as having two distinct names for the same object makes it difficult to ensure consistency.

1st French Singaporean Workshop on Formal Methods and Applications 2013 (FSFMA'13).
Editors: Christine Choppy and Jun Sun; pp. 68–73

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Literature offers a wealth of related work, and *Mezzo* draws inspiration from several areas. The biggest source of inspiration is *Separation Logic* [8, 4], a program logic that describes the state of the heap. In separation logic, asserting that an object has a given type amounts to owning that object. We reuse that principle, but turn it into a type system through our notion of permission, while also extending the reasoning to non-mutable portions of the heap. The *Plaid Project* [2] annotates references to objects with permission. This asserts both what one is allowed to do with the object, as well as what others may do with it. Our permission mechanism supports similar reasoning, but unifies both the state and the mutation invariants of an object, using permissions. In *Mezzo*, pointers can be copied, while the original permission on the object remains. We keep track of aliasing through the use of singleton types, inspired by *Alias Types* [9]. Our notion of affinity, expressing that items may be used at most once, while others may be used freely, is reminiscent of *Linear Types* [1].

We begin with an introduction to permissions, a core concept in *Mezzo*. Next, we discuss a case study, emphasizing several possible mistakes ruled out by our typing discipline. Finally, we give an overview of other *Mezzo* features and conclude with pointers to reference material.

2 An introduction to permissions

The central concept in *Mezzo* is that of a *permission*. While in the λ-calculus we say that "x has type t", in *Mezzo* we say we have permission x @ t, which we read "x at type t". We can think of a permission as a token that grants access to variable x with type t.

Unlike traditional typing judgements, permissions are transient. The user may possess x @ t at some point of the program, and have instead x @ u later on, which accounts for the fact that the type of x changed from t to u, i.e. that the *state* of x changed.

A permission may be obtained by creating a new value. Writing "`let x = (1, "hello")`" yields x @ (int, string), granting its owner the right to use x as a tuple of an int and a string.

2.1 Permissions control effects

Permissions appear in the signature of functions. Let us consider the type of the `length` function, which, as the name implies, computes the length of a list. The square brackets stand for universal quantification.

```
val length: [a] (x: list a) -> int
```

The introduction of the name x, along with its type list a, is syntactic sugar: the function expects an argument named x, along with a permission x @ list a. Conceptually, the function demands a token of ownership from its caller, so as to iterate over the list and compute its length. Hence, whenever one wishes to *call* the function on argument x, a permission x @ list a will be removed from the caller's current set of permissions.

Unless otherwise specified, and as a syntactic convention, we understand the permission x @ list a to be returned to the caller. Therefore, after the function call, the caller will regain x @ list a, along with a permission r @ int, where r is the name of the return value.

Another, more sophisticated function type, is that of the `annotate` function. It takes a *mutable* binary search tree of strings and modifies each node to store a pair, consisting of its original value and the size of the corresponding subtree, which the function returns.

```
val annotate: (consumes t: mtree string)->(int|t@ mtree (string,int))
```

Thanks to the `consumes` keyword, this function now takes a permission t @ mtree string from the caller and returns a *different* permission for t, namely t @ mtree (string, int). Therefore,

the type of t changes through a call to `annotate`. This is a *type-changing update*, which the permissions mechanism accurately describes.

2.2 Permission denote ownership

In order for the above function to be sound, no one else must own a copy of t @ mtree string, since that copy would be invalidated after calling `annotate`. Therefore, permissions that denote mutable portions of the heap must be uniquely owned. We say that the permission t @ mtree string is *exclusive*. The type system enforces this policy, by preventing exclusive permissions from being duplicated.

Conversely, x @ (int, string) denotes read-only knowledge, as our tuples, integers and strings are immutable. This knowledge is permanent, as the type of x will never change. Hence, it is sound to share that information. We say that the permission is *duplicable*, and the type-checker will allow the user to obtain as many copies of the permission as desired.

A permission x @ t therefore states not only that "x has type t", but also that "we own x at type t". The user (and the type-checker) can, by looking at t, infer whether t is exclusive or duplicable, i.e. whether they have an exclusive, read-write access, or a shared, read-only access to the object. The details for this procedure, called *mode inference*, are available [7].

Some permissions are neither exclusive nor duplicable; they are said to be *affine*. Such a permission is x @ a, where a is an abstract type variable, which may occur in the body of a function polymorphic in a. We have to be conservative and make no assumptions on a.

2.3 Permissions track aliasing

At any given program point, a current permission is available, granting us ownership of a part of the heap. Combining *atomic* permissions of the form x @ t into a composite permission is achieved using the $*$ connective; we say that the conjunction of p and q is $p * q$. This conjunction is reminiscent of separation logic. Indeed, if t and t' are both exclusive, the conjunction x @ $t * y$ @ t' implies that, because one cannot hold two exclusive permissions for the same variable, x and y must be distinct. This is a *must-not-alias constraint* and we state that our $*$ conjunction is separating on exclusive portions of the heap.

Moreover, $*$ extends the conjunction of separation logic to non-exclusive portions of the heap. If t' is duplicable, then x @ $t * y$ @ t' yields no information: x and y may or may not be aliases, and the conjunction just has to be consistent. The same situation holds if both t and t' are duplicable. Normally, conjunctions are consistent: if Nil denotes the empty list cell, x @ list int $*$ x @ Nil is a conjunction that is always consistent. However, inconsistent conjunctions exist, such as x @ mtree int $*$ x @ mtree int. Our system has been proved sound, meaning that a program cannot reach a configuration where this conjunction holds. This point in the program is unreachable: it is statically ruled out as "dead code".

These must-not-alias constraints, expressed implicitly in a conjunction, are completed by *must-alias constraints*, which are expressed using *singleton types*. A singleton type is of the form $=y$, where y is a program variable. This type has exactly one inhabitant: y itself. Therefore, having x @ $=y$ means that x and y are actually equal; in particular, if they are pointers, they point to the same value. We write this using syntactic sugar as $x = y$.

A singleton type appears whenever one creates an alias. If x @ t holds, then writing `let y = x in ...` yields a new permission $x = y$, without duplicating the original permission on x, which greatly simplifies reasoning. Singleton types are pervasive in *Mezzo*; they are particularly useful when combined with *structural types*.

Listing 1 Definition of mutable, binary trees

```
data mutable mtree a =
  | Null
  | Node { left: mtree a; value: a; right: mtree a }
```

Listing 1 above shows how to define an algebraic data type in *Mezzo*. Defining such a type allows one to obtain permissions of the form x @ mtree a. However, the type of x may be refined using a match expression; one may trade this permission for a more precise one, of the form x @ Node {left : mtree a; value : a; right : mtree a}. To understand what it means to own a value with such a type, let us rewrite this compact permission, introducing names for the three fields of x, as: x @ Node {left : $=l$; value : $=v$; right : $=r$} $*$ l @ mtree a $*$ v @ a $*$ r @ mtree a.

The ownership semantics of the compact permission can now be understood as stating that we own a memory block at address x of size four, containing a tag Node and three fields. We also own two mutable trees located at addresses l and r, along with a value of type a named v. The points-to relationships are represented by the singleton types. Similarly, the meaning of a nominal permission, such as x @ mtree a, is the disjunction of the meaning of its unfoldings x @ Null and x @ Node {left : mtree a; value : a; right : mtree a}.

3 A *Mezzo* case study

We now discuss a motivating example that, by using all the mechanisms described above, avoids several pitfalls. This example, as shown in listing 2, consists of splitting a mutable binary search tree. The function `split` "steals" the ownership of its argument t from its caller, and returns two binary search trees: the first one containing all values $v \leq k$ and the second one containing all values $v > k$. The `split` function abstracts over the comparison function `cmp`. We omit the re-balancing of the tree and focus on a self-contained example.

There are several pitfalls that await the programmer when writing such code. The user may inadvertently copy a key: this would violate the invariant that the two sub-trees form a partition. The user may return, as the right sub-tree, a pointer into the left sub-tree: this would create undesired sharing, leading to subtle bugs when two concurrent threads will want to access the two sub-trees, assuming that they are distinct. The permissions mechanism, by ensuring that exclusive knowledge cannot be duplicated, enforces these invariants statically.

Listing 2 In-place splitting of a mutable binary search tree

```
1  val rec split_right [a] (
2    consumes parent: Node { left: mtree a; value: a; right = child},
3    consumes child: mtree a,
4    k: a,
5    cmp: (a, a) -> int
6  ): (mtree a | parent @ mtree a) =
7    match child with
8    | Null -> Null
9    | Node ->
10       if cmp (child.value, k) <= 0 then
11         split_right (child, child.right, k, cmp)
12       else begin
13         let left_leq, left_gt = split (child.left, k, cmp) in
14         parent.right <- left_leq;
15         child.left <- left_gt;
16         child
17       end end
18
```

```
19 and split [a] (consumes t: mtree a, k: a, cmp: (a, a) -> int)
20   : (mtree a, mtree a) =
21   match t with
22   | Null -> Null, Null
23   | Node ->
24       if cmp (t.value, k) <= 0 then begin
25         let right_gt = split_right (t, t.right, k, cmp) in
26         t, right_gt
27       end else begin
28         let left_leq, left_gt = split (t.left, k, cmp) in
29         t.left <- left_gt;
30         left_leq, t
31       end end
```

3.1 The `split` function

The `split` function is the entry point. At the start of the function body, the permission is:

$$t \mathbin{@} \mathsf{mtree}\ a * k \mathbin{@} a * \mathsf{cmp} \mathbin{@} (a, a) \to \mathsf{int}$$

The function begins by matching on its argument t. If t is Null, two empty trees are returned. In the converse case, the permission on t is refined to:

$$t \mathbin{@} \mathsf{Node}\,\{\mathsf{left} : = l; \mathsf{value} : = v; \mathsf{right} : = r\} * l \mathbin{@} \mathsf{mtree}\ a * v \mathbin{@} a * r \mathbin{@} \mathsf{mtree}\ a$$

In the case that $t.\mathsf{value} \leq k$ holds, a call to `split_right`, whose meaning we will explain in the next section, is made. In the case that $k < t.\mathsf{value}$ (line 29), values greater than k may be found in the left sub-tree. The recursive call yields a partition of left sub-tree, consuming $l \mathbin{@} \mathsf{mtree}\ a$, while producing $\mathsf{left\_leq} \mathbin{@} \mathsf{mtree}\ a * \mathsf{left\_gt} \mathbin{@} \mathsf{mtree}\ a$. We re-attach values greater than k into $t.\mathsf{left}$, thus changing the permission of t into $t \mathbin{@} \mathsf{Node}\,\{\mathsf{left} : = \mathsf{left\_gt}; \mathsf{value} : = v; \mathsf{right} : = r\}$ Values lesser or equal to k are returned, along with t, which now contains the set of all possible values greater than k.

Which mistakes could an absent-minded programmer do? A first one would be forgetting to perform the assignment "`t.left <- left_gt`", at line 30. The permission for t would still be $t \mathbin{@} \mathsf{Node}\,\{\mathsf{left} : = l; \mathsf{value} : = v; \mathsf{right} : = r\}$. However, the call to `split`, at line 29, consumed the permission for l: it is no longer available, thus preventing this type from being folded back to $\mathsf{mtree}\ a$, when exiting the function. *Mezzo* would reject this program.

Another beginner's mistake would be to return the value `(left_gt, t)`. Following the return type `(mtree a, mtree a)`, *Mezzo* would consume $\mathsf{left\_gt} \mathbin{@} \mathsf{mtree}\ a$ to prove that `left_gt` is a tree. Next, *Mezzo* would have to prove that `t` is a tree, using permission $t \mathbin{@} \mathsf{Node}\,\{\mathsf{left} : = \mathsf{left\_gt}; \mathsf{value} : = v; \mathsf{right} : = r\}$. Specifically, this implies proving that `t.left`, also known as `left_gt`, is also a tree. Unfortunately, that exclusive permission was already consumed. This situation is therefore rejected.

3.2 The `split_right` function

The `split_right` function is written in a different style, as it takes a non-null `parent` tree, along with its right `child`. After the function call, the parent is still a tree, holding all values lesser or equal to k, while the returned tree contains all values greater than k.

The call to `split_right` at line 11 is legal, as we know that `child` is a `Node`, which justifies using it as the first argument. *Mezzo* statically checks that the second argument is indeed

the right child of the first: this information is known statically, due to the use of singleton types. This contrasts with the usage in traditional imperative languages, where typical code would rely on a loop and two mutable variables, with the implicit invariant that one is the right child of the other. Here, the invariant is made explicit and *Mezzo* can rule out misuses.

The type-checker applies recursive reasoning. After the call to `split_right` at line 11, if ret denotes the return value of the recursive call, the remaining permission is:

$$\mathsf{parent} \mathbin{@} \mathsf{Node}\,\{\mathsf{left} : \mathsf{mtree}\ a; \mathsf{value} : a; \mathsf{right} : {=}\mathsf{child}\} * \mathsf{child} \mathbin{@} \mathsf{mtree}\ a * \mathsf{ret} \mathbin{@} \mathsf{mtree}\ a$$

The type-checker then performs one last folding, to obtain the desired return type for the function. Note that the function call is tail-recursive, while the reasoning is *not*. Indeed, the use of recursive functions with distinct pre- and post-conditions yields more expressiveness than the use of traditional loops. This allows for stronger invariants.

4 Conclusion

Due to its *permissions* formalism, the *Mezzo* language manages to state precise invariants for programs that rely on mutable state, thus preventing several programming mistakes. The key mechanisms enforcing this rely on ownership, linearity and singleton types.

Permissions, as presented here, cannot account for non tree-shaped aliasing patterns. However, *Mezzo* offers several mechanisms for evading this restriction, e.g. locks, Boyland's nesting [3] and our own adoption/abandon mechanism. A more thorough discussion can be found [7], which details the language with typing rules and a formal definition of permissions. A gallery of programs along with extended material are available on the website [6].

The soundness of *Mezzo* has been machine-checked [5]. In the future, we wish to extend the language and its soundness proof with concurrency, to guarantee data-race freeness.

References

1 Amal Ahmed, Matthew Fluet, and Greg Morrisett. L^3: A linear language with locations. *Fundamenta Informaticæ*, 77(4):397–449, 2007.

2 Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 301–320, 2007.

3 John Tang Boyland. Semantics of fractional permissions with nesting. *ACM Transactions on Programming Languages and Systems*, 32(6), 2010.

4 Peter W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.

5 François Pottier. Type soundness for Core Mezzo. Unpublished, January 2013.

6 François Pottier and Jonathan Protzenko. *Mezzo*. `http://gallium.inria.fr/~protzenk/mezzo-lang/`, January 2013.

7 François Pottier and Jonathan Protzenko. Programming with permissions in *Mezzo* (long version). Unpublished, March 2013.

8 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, pages 55–74, 2002.

9 Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *European Symposium on Programming (ESOP)*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381. Springer, 2000.

Improving System-Level Verification of SystemC Models with SPIN*

Martin Elshuber, Susanne Kandl, and Peter Puschner

Institute of Computer Engineering
Vienna University of Technology
Treitlstr. 3, 1040 Wien, Austria
{martine,susanne,peter}@vmars.tuwien.ac.at

Abstract

SystemC is a de-facto industry standard for developing, modelling, and simulating embedded systems. As embedded systems become more and more integrated into many aspects of human lives (e.g., transportation, surveillance systems, ...), failures of embedded systems might cause dangerous hazards to individuals or groups. Guaranteeing safety of such systems makes formal verification crucial. In this paper we present a novel approach for verifying SystemC models with SPIN. Focusing on system-level verification we reuse compiled and executable code from the original model and embed it into the verifier generated by SPIN. In contrast to most other approaches, which require a complete model transformation, in our approach the transformation focuses only on the relevant parts of the model while leaving functional blocks untransformed. Our technique aims at reducing the state vector size managed by the verifier of SPIN, at improving state exploration performance by avoiding unnecessary model transformation steps, and at concentrating on verifying properties that emerge from the composition of multiple functional units.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases SystemC, SPIN, Promela, System-Level Verification

Digital Object Identifier 10.4230/OASIcs.FSFMA.2013.74

1 Introduction

Nowadays computer systems are more and more introduced into many aspects of human lives. Especially when they contain safety-relevant features, failing may cause dangerous hazards. Consequently, formally verifying that certain properties of the system hold under all circumstances becomes a central task in system design.

With the growing complexity of state-of-the-art computer systems, manual proofs often turn out to be infeasible and error prone. To circumvent this problem, tools have been developed that analyse models at different abstraction levels (e.g., system specification, system implementation, ...) in order to formally prove that system properties match the desired behaviour of the developed product.

SystemC: *SystemC* is a de-facto industry standard for modelling systems at system level, and can be used to model software and hardware aspects in a single language. *SystemC* is an add-on library to C++. *SystemC* extends C++ by constructs similar to Hardware

* This work has been partially funded by the ARTEMIS Joint Undertaking and the National Funding Agency of Austria for the project VeTeSS under the funding ID ARTEMIS-2011-1-295311.

Description Language (HDL) languages and a scheduler. Such models can be compiled to native machine code for most of the existing hardware architectures, thus allowing fast and accurate simulation of the system.

State-of-the-art verification: Although simulation is a proper method for detecting many bugs in a system, it cannot be used to verify whether a property of a system holds for every possible system state or not. Formal verification, on the other hand, guarantees the validity of a property for all possible system states.

Amongst others, we emphasise two reasons that make it difficult to formally verify *SystemC* models: *(1)* the constructs introduced by *SystemC* use a scheduler to execute and schedule processes activated on specific events. *(2) SystemC* allows to freely use C++ constructs like class inheritance, library functions, or Standard Template Library (STL), ...

State-of-the-art techniques address these problems by transforming the model to another language and use existing tools. This transformation requires *(ad 1)* to model the scheduler explicitly, thus increasing the overall state space, and *(ad 2)* to restrict the *SystemC* model to a specific subset of *SystemC* (e.g., prohibiting class inheritance).

This paragraph gives a short summary of existing verification tools. The *SystemC* category in the 2nd International Competition on Software Verification (SV-COMP) 2013 [2] provided the *SystemC* benchmarks already transformed to *C*. Cimatti et al. [6] describe the transformation of *SystemC* to *C*, thus reducing *SystemC* verification to software verification. The implementation presented in this work uses the tool *Pinapa* [11]. *Pinapa* is a predecessor of *PinaVM* [10], the tool we are referring to in this work. The *SystemC* category of SV-COMP 2013 was won by *UFO* [1], a framework for software verification working on LLVM Bitcode (LLVM BC). Second and third place were assigned to two *CPAchecker*-based verifiers [3]. Also worth to mention are Bounded Model Checking (BMC) [4] approaches used for example by *CBMC* [7]. *Scoot* [5] is an extension to *CBMC* allowing *SystemC* verification. SPIN [8] is a popular tool for proving properties of asynchronous distributed systems specified in the language *Promela*. PAT (Process Analysis Toolkit) [12] is a modular toolkit for verification and simulation of concurrent systems.

A glance at our approach: In system-level verification we concentrate on the composability aspects of systems consisting of several functional blocks. Assuming that each functional block works as specified, we are interested in verifying properties that emerge from the composition of those blocks. The approaches mentioned above aim to exhaustively verify the system with all implementation aspects included.

Our approach solely transforms the interaction of functional blocks into the formal language *Promela* and executes code within a functional block as a single transition. Based on a model analysis done by *PinaVM* at LLVM BC level, we split the model into several functions which are embedded into the SPIN verifier and executed atomically. With this technique we can use the model checking capability of SPIN on a model that represents the relevant aspects of the system, whereas details within single blocks of the system are hidden. Thus it is possible to focus on the verification of system properties without considering functional details which may easily cause a state space explosion during the verification. In [13] a similar approach for multi-threaded C programs, where SPIN orchestrates the search, is proposed.

The remainder of this paper is structured in four further sections. Section 2 gives a more detailed overview on existing technology reused during this work. Section 3 describes the verification process of our approach. In the following sections we discuss the advantages and disadvantages of the concept and conclude with a summary of the paper.

2 Prerequisites

This section gives a brief introduction to existing technology (namely *SystemC*, SPIN, and *PinaVM*) we build our approach on top.

SystemC: *SystemC* is a library on top of C++. For hardware aspects *SystemC* modules are defined. They contain input-, output-, bidirectional ports for communication, and processes acting on these ports. Software aspects can be implemented using classic C++. The important fact is that *SystemC* processes are executed non-preemptively. The effect is that all modifications to the system state semantically take place at the instant when the process preempts itself. As already mentioned above, state-of-the-art verification of *SystemC* models often requires a transformation from *SystemC* to another language, and to model the scheduler separately.

SPIN: SPIN is an on-the-fly model checker, which can be used to run and verify models described in the language *Promela*. The interesting part is the way SPIN verifies a model. It first translates the *Promela* model into a C verifier which has to be compiled and run to execute the verification. In the verifier the *Promela* model is translated into a transition system represented as a `switch/case` statement. The verifier then searches the transition system for errors and reports paths if a problem was found. To allow backtracking the state vector is stored and compared against states which have already been investigated. SPIN also implements various performance features, like partial order reduction, and techniques reducing the memory requirements for storing the state vector. *Promela* provides constructs for in-lining C-Code into the verifiers transition system.

PinaVM: *PinaVM* is a tool, developed by Verimag [10], able to analyse the structure *SystemC* models. It detects which *SystemC* modules are created and how they interact with each other. Thus simplifying the translation into arbitrary languages. *PinaVM* roughly works in several phases:

Phase 1: Use LLVM to create an LLVM BC of the model.

Phase 2: Analyse the created functions in Phase 1 and find out where each *SystemC* construct is used.

Phase 3: Execute the models initialisation code generated in Phase 1 detecting the instantiated *SystemC* classes.

Phase 4: In this instant it is known what the *SystemC* model looks like (instantiated modules; Phase 2), how the interact (instantiated ports; Phase 2) and which code parts manipulate the structures (Phase 1). This information is passed to a back-end, which transforms the model to the desired format usable by existing model checker infrastructure.

PinaVM also provides a back-end for *Promela* including efficient encodings [9] for *SystemC* constructs like `wait`, `notify`, ... The *Promela* back-end translates every instruction of the LLVM BC into a corresponding *Promela* construct. We, on the other hand, only transform the *SystemC* constructs to *Promela*, and in-line the rest of the code directly into the SPIN verifier.

3 Our Approach

Our verification process is based on the given infrastructure described in Section 2. We plan to use *PinaVM* for analysing *SystemC* code, to add our own transformation technique, and to use SPIN for verifying the resulting *Promela* model. The difference to *PinaVM* using the

Promela back-end, is that we do not translate the whole model to *Promela*. We only model the *SystemC* constructs in *Promela* and include calls to the compiled *SystemC* model. These calls also modify the verifier's state vector.

Figure 1 depicts the steps executed to create a verifier binary with our approach. The gray blocks denote the steps that can be done by existing tools. These are SPIN, *PinaVM* or *LLVM*. Implementations for the white boxes are currently missing and are subject for the work to be done during the thesis. Most steps (except *Create Verifier C-Code* and *Compile & Link Verifier*) are done by PinaVM or an extension of it. PinaVM itself is based LLVM BC and uses *LLVM* libraries to handle the code.

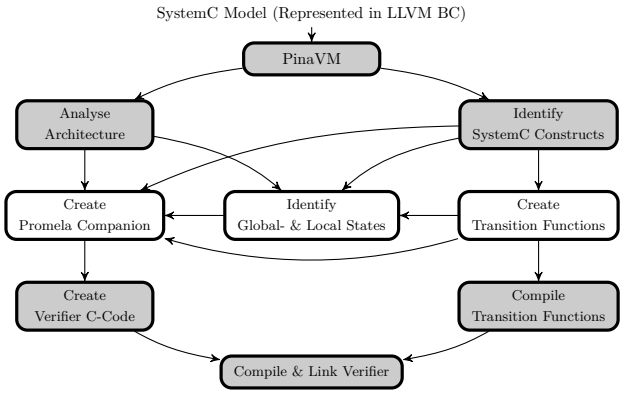


Figure 1 From *SystemC* to the SPIN verifier executable.

The output of *PinaVM* to the back-end is the LLVM BC, enriched with information on *SystemC* constructs, as well as the system architecture instantiated during model initialisation. The text below describes the actions taken within each block:

- During **Analyse Architecture** the initialisation code of the model is analysed by *PinaVM* in order to detect the kind, number, and interaction channels and the *SystemC* modules.
- During **Identify SystemC Constructs** all functions are analysed by *PinaVM* to detect and mark *SystemC* constructs such as `wait` and `notify`, but also `write` and `read`.
- **Create Transition Function:** The *SystemC* model is split into transition functions callable by the verifier. These functions are compiled separately and finally linked to the verifiers binary.
  The original functions are divided into code regions, such that each code region either contains no *SystemC* construct at all, or it consists solely of a single *SystemC* construct. Each code region that contains no *SystemC* construct is converted into a separate function. This function returns a reference to the next code region to be executed, and it receives parameters according to the values read or modified during execution.
  The resulting functions have the same semantics as the original functions when called in a proper order. Furthermore each function returns at each point the *SystemC* scheduler might preempt the execution of the original thread.
- **Identify Global- & Local States:** Depending on the model the states of the system have to be identified. *Promela* distinguishes three kinds of state variables. *Global* state variables are instantiated only once, *Local* state variables are instantiated per process and thus can be stored multiple times in the state vector, and finally *hidden* states are never stored in the state vector, thus they cannot be restored on backtracking.
  In our approach variables that are read and written solely within a code region, can be totally hidden from the verifier and are stored on the function's stack or optimised into a processor register. Variables that are written in a code region, but possibly read by another code region have to be instantiated somewhere in the verifier. To decide which type of variable has to be used, control flow analysis has to be done.
  A variable can be declared *hidden*, if and only if all statements between (and including) writing and reading are executable in an atomic manner. Because *SystemC* processes

are non-preemptive, this is the case if no blocking *SystemC* construct can be executed between writing and reading.

The rest of the variables have to be stored in the state vector, and are *global* or *local*. Variables are declared local if they are declared locally in the *SystemC* process parenting the code region. They are declared globally otherwise. Member variables of *SystemC* classes can also be declared globally, because from the architecture analysis phase it is known how many objects are instantiated.

- **Create *Promela* Companion:** The *Promela* Companion is the actual input file to SPIN. It encodes
  - *SystemC* constructs similar to [10, 9],
  - definitions of all *hidden*, *local* and *global* state variables accessible by other Promela constructs like LTL formulas, and
  - calls to the transition functions as well as the control flow among them (`goto` statements, and C in-lining).
- Finally the verifier executable can be generated by using SPIN to **create the verifier C-Code** and LLVM to compile the transition functions, the verifier, and link all together.

4 Discussion

The main aim is to create a source-code driven verification system, allowing us to verify properties on system level while disregarding details within functional blocks. A requirement of the implementation of each functional block is that it is free of memory bugs such as buffer under- and overflow, access violations, and so on.

Furthermore, the structure of the model architecture has to be static. This means that no dynamic *SystemC* constructs must be created, except during the initialisation code. This requirement stems from they way *PinaVM* works.

SystemC verification is often driven by translating the model into plain C or similar languages and by adding a scheduler to the translated model. This has the disadvantage of introducing additional states and thus adding complexity to the verification process. With our approach we reuse at least parts of the process scheduler of SPIN, thus aiming at an improvement of the verification process.

We also expect improvements in the memory requirements, by reducing the size of the state vector. The expected effect is mainly caused by hiding states from the verifier, thus disallowing it to backtrack to system states that are irrelevant for system-level verification. Assume a model does some computation inside a loop, whereas the result is passed to other functional blocks solely after the loop. The intermediate results (e.g., temporary variables) do not have to be included in the system states. Thus we expect an improvement in both the state vector size and the number of states that have to be investigated. The effect of the latter one is expected to be smaller as partial order reduction also can be done by SPIN.

By avoiding the complete transformation of the model from C++ to *Promela* and then back to C again, we think that we can improve the search speed of the verifier, and thus increase the number of states investigated per second. This is because each transition in a *Promela* model is selected by a switch statement. The size of the switch statement and the number of entries within it is expected to be reduced.

So far we implemented a prototype that automatically extracts the transition functions of simple *SystemC* models. First experiments showed that the size of the state vector is reduced and the number of explored states is kept at a similar amount compared to the *Promela* backend of PinaVM.

5 Conclusion

In this paper we presented an idea for a novel approach to improve the formal verification process on system-level of a *SystemC* model. The *SystemC* model is transformed into a formal automaton model by interpreting the *SystemC* constructs and assigning precise semantics to them. By a straight-forward transformation the whole functionality of the *SystemC* model is represented in the resulting formal model with the consequence that all the complexity of the system description is part of the verification process. In our approach the model transformation is realized in such a way that functional details within a block of the system model are hidden and only the aspects of the model that are relevant for system-level verification are considered for the verification process. This principle should enhance the verification process by saving time and memory within the model checking process by SPIN.

References

1 Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. Ufo: A Framework for Abstraction- and Interpolation-Based Software Verification. In *CAV*, pages 672–678, 2012.

2 Dirk Beyer. Second Competition on Software Verification - (Summary of SV-COMP 2013). In *TACAS*, pages 594–609, 2013.

3 Dirk Beyer and M.Erkan Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer Berlin Heidelberg, 2011.

4 Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded Model Checking, 2003.

5 Nicolas Blanc, Daniel Kroening, and Natasha Sharygina. Scoot: A Tool for the Analysis of SystemC Models. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 467–470. Springer Berlin Heidelberg, 2008.

6 A. Cimatti, A. Micheli, I. Narasamdya, and M. Roveri. Verifying SystemC: A software model checking approach. In *Formal Methods in Computer-Aided Design (FMCAD), 2010*, pages 51–59, 2010.

7 Edmund Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

8 Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

9 Kevin Marquet, Jeannet Bertrand, and Matthieu Moy. Efficient Encoding of SystemC/TLM in Promela. In *Proceedings of the International MultiConference of Engineers and Computer Scientists 2011*, pages 1039–1044, 2011.

10 Kevin Marquet and Matthieu Moy. PinaVM: A SystemC front-end based on an executable intermediate representation. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '10, pages 79–88. ACM, 2010.

11 Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. Pinapa: an extraction tool for SystemC descriptions of systems-on-a-chip. In *Proceedings of the 5th ACM international conference on Embedded software*, EMSOFT '05, pages 317–324. ACM, 2005.

12 Jun Sun, Yang Liu, JinSong Dong, and Jun Pang. PAT: Towards Flexible Verification under Fairness. In *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 709–714. Springer Berlin Heidelberg, 2009.

13 Anna Zaks and Rajeev Joshi. Verifying Multi-threaded C Programs with SPIN. In *Proceedings of the 15th international workshop on Model Checking Software*, SPIN '08, pages 325–342. Springer-Verlag, 2008.

Modelling and Reasoning about Dynamic Networks as Concurrent Systems

Yanti Rusmawati[1] and David Rydeheard[2]

1 **PhD student, School of Computer Science, The University of Manchester**
Oxford Road, Manchester M13 9PL, UK
rusmaway@cs.man.ac.uk

2 **School of Computer Science, The University of Manchester**
Oxford Road, Manchester M13 9PL, UK
david@cs.man.ac.uk

Abstract

We propose a new approach to modelling and reasoning about dynamic networks. Dynamic networks consist of nodes and edges whose operating status may change over time (for example, the edges may be unreliable and operate intermittently). Message-passing in such networks is inherently difficult and reasoning about the behaviour of message-passing algorithms is also difficult. We develop a series of abstract models which allow us to focus on the correctness of routing methods. We model the dynamic network as a "demonic" process which runs concurrently with routing updates and message-passing. This allows us to use temporal logic and fairness constraints to reason about dynamic networks. The models are implemented as multi-threaded programs and, to validate them, we use an experimental run-time verification tool called RuleR.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases dynamic networks, temporal logic, concurrent systems

Digital Object Identifier 10.4230/OASIcs.FSFMA.2013.80

1 Introduction

We are increasingly reliant on highly dynamic and complex computing systems. In communication, dynamic networks are widespread these include the: Internet, peer-to-peer networks, mobile networks and wireless networks. Networks may have edges down, nodes may move, or there may be routing instability due to the changing of networks. These systems are very difficult to analyse, and their behaviour and correctness are hard to formulate and establish. To undertake formal reasoning about such systems, abstract models are essential in order to separate the general reasoning about message routing and updating of routing tables from the details of how these are implemented in particular networks. We show we can establish correctness of dynamic networks at suitable levels of abstraction.

At its simplest level, a network consists of a collection of nodes connecting to each other through edges. In a message-passing network, each node communicates by exchanging messages in an attempt to deliver messages to their destinations. In a dynamic network, nodes and/or edges may become inoperative or operative. This representation of the dynamics of a network clearly models unreliable networks. It also models mobile and wireless networks by considering edges as possible communication links and operative edges as the links established at a particular time.

The two problems of message-passing in high-level models [2, 3, 4] and self-stabilising systems [5] in dynamic networks have been widely studied [1]. Numerous models and algorithms

1st French Singaporean Workshop on Formal Methods and Applications 2013 (FSFMA'13).
Editors: Christine Choppy and Jun Sun; pp. 80–85

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

have been proposed but proofs of correctness (especially liveness) have tended to need the assumption that changes in networks eventually cease, i.e., they are no longer dynamic.

Here, the correctness of dynamic networks can be established without the termination requirement. We develop the correctness of dynamic networks in terms of ensuring that: even when routing tables do not reflect the actual network connections, the routing information is correct sufficiently often; messages eventually get delivered; the network is sufficiently connected for sufficiently often; and there is no persistent livelock. The main contributions of this paper are: modelling dynamic networks using concurrent systems; factorisation of proof; and run-time verification of the implementation of dynamic network models.

We introduce a new approach to proof techniques for dynamic networks in which using ideas from concurrent systems [11] to analyse message-passing. To do so, we use Linear Temporal Logic and formulate concepts of fairness which capture network properties. In order to express dynamic networks as concurrent systems [6], we consider the dynamic changes to be the result of a "demonic" process which runs concurrently with routing updates and message-passing. By the correctness of dynamic networks, we mean that, under certain conditions, all messages will eventually be delivered. By formulating networks as concurrent systems, we can establish correctness for networks that never cease to change. By modelling at this level of abstraction, we are able to prove the properties of networks independently of the mechanisms in actual networks and therefore provide "a factorisation" of proofs of correctness for actual dynamic networks. We have implemented two abstract models as concurrent systems and then adapted a run-time verification systems RuleR [8], to analyse execution traces to test whether model instances satisfy the modal correctness for message delivery.

2 Abstract models

Consider a graph $\mathsf{G} = (\mathsf{N}, \mathsf{E})$ where N is a set of nodes and E is a set of edges. This provides the basic connectivity of a network. To introduce dynamics (for edges) we consider the edges to be in an *on* or *off* state. Therefore we introduce edge status L, which changes value whenever a disruption occurs for an edge. Routing update information is determined by L. There is a set of messages M, where each message is at a node defining a function $\mathsf{M} \to \mathsf{N}$.

We develop dynamic network models as concurrent systems. We introduce two models. Firstly, *Model 1*, as shown in Figure 1, is a two-process model with instantaneous updates, in which the routing tables are always correct. The two processes are a "demonic" *Disrupter* (which disrupts the connectivity of dynamic networks) and an *Organizer* (which attempts to deliver messages). In *Model 2*, we introduce a more realistic routing table update, adding a third process called *Updater*, as shown in Figure 2. Here, the routing tables may not be correct at any time but routing is still possible. The *Disrupter* process can disrupt the connection of an edge (so it becomes 'on' or 'off'). The *Updater* runs concurrently recalculating the

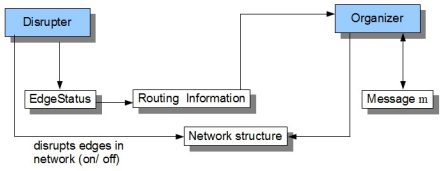


Figure 1 Two processes dynamic network model.

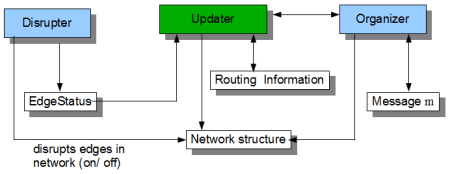


Figure 2 Three processes dynamic network model.

routing update information to obtain actual available paths. If there is an available path, the *Organizer* process runs concurrently can send a message to the next node along a path.

3 Proving correctness of message-passing in dynamic networks

We use discrete-time Linear Temporal Logic (LTL) [10] to describe the properties of execution traces of this multi process systems. Some of the key properties which enable us to reason about network correctness are expressed as *fairness* constraints [9] in concurrent process models. We use strong fairness at the process level to express, for example, the relative frequency of network change to message motion and of routing table updates to network change.

Our aim is to formulate and prove the correctness of message-passing using the two abstract models above. We need to prove that, under certain conditions, all messages eventually reach their destination. We introduce a colouring of message according to their states. This is inspired by Gries [13] and Dijkstra [12]'s reasoning about on-the-fly garbage collection. Here Black means a message is at its destination; green means a message is progressing along the route; and red means no route is allocated at present. Notation (with ϕ being any formula): $\Box\ \phi$ means ϕ always holds in every state; $\Diamond\ \phi$ means ϕ eventually holds in some state; $\bigcirc\ \phi$ means ϕ holds in the next state; $pathX(\mathsf{n}_1, \mathsf{n}_2)$ means that there is an available path between node n_1 and node n_2; $\mathsf{P}(\mathsf{n}_1, \mathsf{n}_2)$ being a set of paths between node n_1 and node n_2; path(m, p) means that message m is allocated path p; $\mathsf{rt}(\mathsf{n}_1, \mathsf{n}_2, \mathsf{p})$ being the routing table entry saying p is a path from n_1 to n_2; and $moved(\mathsf{m}, \mathbf{z})$ means that the location of message m is changed to position $\mathbf{z}$.

For *Model 1*, the modal properties we need are:

1. Paths exist infinitely often:

   P1: $\forall\ \mathsf{n}_1, \mathsf{n}_2 \in \mathsf{N}.\ \Box\Diamond\ pathX(\mathsf{n}_1, \mathsf{n}_2)$

2. Red messages eventually become green (messages are looked at sufficiently often):

   P2: $\forall \mathsf{m} \in \mathsf{M}.\ \Box\ ((\Box\Diamond\ pathX(at(\mathsf{m}), dest(\mathsf{m})))\ \wedge\ red(\mathsf{m})$
   $\Rightarrow\ \Diamond\ (green(\mathsf{m})\ \wedge\ (\exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m}).\ path(\mathsf{m}, \mathsf{p}))))$

▶ **Lemma 1.**

$\forall \mathsf{m} \in \mathsf{M}.\ \Box\ ((P1\ \wedge\ P2\ \wedge\ red(\mathsf{m}))\ \Rightarrow\ \Diamond\ green(\mathsf{m}))$

Proof. Trivial: by modus ponens using P1 and P2. ◀

Notice the formulation expressing the properties of dynamic networks as trace properties, some in terms of fairness of process interaction, others as connectivity properties of graphs.

Now we prove that all messages reach their destination under suitable conditions. We need to establish the following:

(A.) messages eventually move, which means that the *Organizer* should access each message sufficiently often, and when it is accessed, it can be moved. Therefore, we modify P2 to P2″ as follows to include a fairness requirement.

P2″: $\forall \mathsf{m} \in \mathsf{M}.\ \Box\ ((\Box\Diamond\ pathX(at(\mathsf{m}), dest(\mathsf{m}))) \Rightarrow\ \Diamond\ (black(\mathsf{m})\ \wedge$
$(red(\mathsf{m})\ \Rightarrow\ \bigcirc(green(\mathsf{m})\ \wedge\ (\exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).\ path(\mathsf{m}, \mathsf{p}))))\ \wedge$
$(\exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).\ (green(\mathsf{m})\ \wedge\ path(\mathsf{m}, \mathsf{p})\ \wedge\ at(\mathsf{m}) \neq dest(\mathsf{m})\ \Rightarrow$
$(up(\mathrm{1st\_elmt}(\mathsf{p}))\ \wedge\ \bigcirc(green(\mathsf{m})\ \wedge\ path(\mathsf{m}, \mathrm{tail\_of\_path}(\mathsf{p}))\ \wedge$
$at(\mathsf{m}) = next\_node(\mathsf{p})))))\ \wedge ((green(\mathsf{m})\ \wedge\ at(\mathsf{m}) = dest(\mathsf{m}))\ \Rightarrow\ \bigcirc\, black(\mathsf{m}))))$

We also need:

(a) **Finiteness of paths**, which we define as:

$$\text{FP: } \forall \mathsf{m} \in \mathsf{M}.\ ((green(\mathsf{m}) \ \wedge\ \neg\Diamond\ red(\mathsf{m})) \ \Rightarrow\ (\Diamond\ black(\mathsf{m})))$$

which means that if a message m is green and there is no potential to become red eventually then message m will eventually become black. This can only hold if the *Organizer* checks message m infinitely often (P2″).

(b) No livelock. Here we define **Livelock-free** as:

$$\text{LF: } \forall \mathsf{m} \in \mathsf{M}.\ \neg\ \Box\Diamond\ (green(\mathsf{m}) \ \Rightarrow\ \bigcirc red(\mathsf{m})),$$

i.e a green message becomes red (without an assigned route) only finitely often.

(B.) each message **eventually reaches** its destination.

We show that for each m there is a point in the trace at which $\Box\ (green(\mathsf{m}) \ \vee black(\mathsf{m}))$ hence $\Diamond black(\mathsf{m})$.

▶ **Lemma 2.**

$$\forall \mathsf{m} \in \mathsf{M}.\ \Box\ (\exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).\ ((P1\ \wedge\ P2''\ \wedge\ Lemma\ 1\ \wedge\ green(\mathsf{m}) \ \wedge\ path(\mathsf{m}, \mathsf{p}))\ \Rightarrow\ \Diamond\ moved(\mathsf{m}, z)))$$

Proof. Suppose message m has not reached the destination. We then follow from the proof of Lemma 1, and by modus ponens on P1 and P2″, hence we have Lemma 2. ◀

▶ **Lemma 3.**

$$\forall \mathsf{m} \in \mathsf{M}.\ \Box\ ((FP\ \wedge\ LF\ \wedge\ Lemma\ 1\ \wedge\ Lemma\ 2 \ \wedge\ P1\ \wedge\ P2''\ \wedge\ green(\mathsf{m})\ \wedge\ \exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).\ path(\mathsf{m}, \mathsf{p}))\ \Rightarrow\ \Diamond black(\mathsf{m}))$$

Proof. By Lemma 1 and the definition of Livelock-free, and by modus ponens on P1 and P2″, as well as on Lemma 2 and finiteness of path definition, the message is $\Diamond black(\mathsf{m})$. Hence we have Lemma 3. ◀

By Lemma 1, 2, and 3, we have

▶ **Theorem** 1. $\forall \mathsf{m} \in \mathsf{M}.\ \Box\ ((\text{P1}\ \wedge\ \text{P2}\ \wedge\ \text{P2}''\ \wedge\ \text{FP}\ \wedge\ \text{LF}\ \wedge\ red(\mathsf{m}))\ \Rightarrow\ \Diamond black(\mathsf{m}))$.

For *Model 2*, we need an additional property which expresses that the routing table is populated sufficiently often. This is formulated as follows.

$$\text{P3: } \forall\ \mathsf{n}_1, \mathsf{n}_2 \in \mathsf{N}.\ \Box\ ((\Box\Diamond\ pathX(\mathsf{n}_1, \mathsf{n}_2))\ \Rightarrow\ \Diamond\ (\exists \mathsf{p} \in \mathsf{P}(\mathsf{n}_1, \mathsf{n}_2).\ \mathsf{rt}(\mathsf{n}_1, \mathsf{n}_2, \mathsf{p})))$$

We also need to modify P2 to P2′ since paths for messages are obtained from the routing table, replacing $\text{pathX}(\mathsf{n}_1, \mathsf{n}_2)$ with $\mathsf{rt}(\mathsf{n}_1, \mathsf{n}_2, \mathsf{p})$. We modify P2′ to P2‴ to extend the fairness requirement which includes the routing tables when the message m is at its position. The model also needs routing tables which are correct sufficiently often. In *Model 2*, the condition P2″ is replaced by P2‴, then finally we have a similar theorem. The proofs proceeds as for *Model 1*.

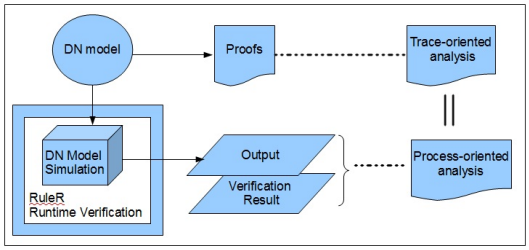


Figure 3 Run-time verification on the implementation of dynamic network models.

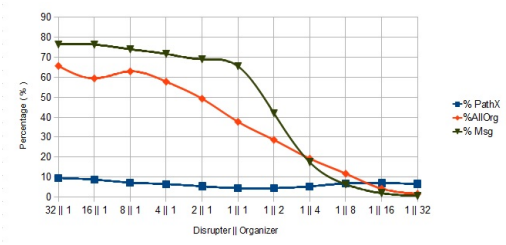


Figure 4 Dynamic network model, property P2.

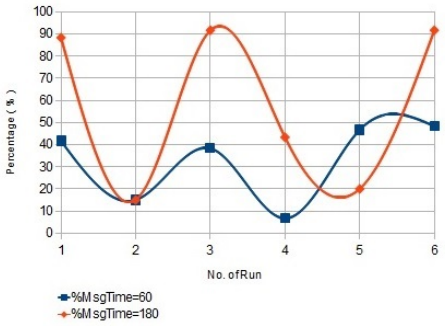


Figure 5 Dynamic network model with possible livelock.

4 Experimental validation: Using run-time verification

We now consider the following question. Suppose a dynamic network is implemented as a concurrent system using multiple Java threads. When do the network properties (expressed as properties of execution traces of concurrent systems as above) hold and therefore, by the proofs above, all messages are eventually delivered?

There are several approaches. We could prove the implementation manually or we could use a verification technique such as model checking. Here we introduce a new approach based on run-time verification (RV) [7], as pictured in Figure 3. Whether or not a system satisfies the properties required for message delivery depends on interprocess interaction and the parameters involved in this. Run-time verification is particularly suitable here as it is the relationship of these parameters with the execution traces that determine the correctness of the dynamically allocated interprocess interaction. We have implemented *Model 1* and *Model 2*, and use an experimental run-time verification tool RuleR, which has been developed by Barringer et al. [8]. RuleR is a rule-based run-time verification with dynamic rules. This is an experimental use of RV on concurrent models. Some trace properties required are properties properly of infinite traces. We show how to use RV to examine finite traces and relate this to the overall network behaviour.

Consider a result of *Model 1*, as Figure 4 shows, in which *Disrupter* process and *Organizer* process running concurrently (denoted as "Disrupter || Organizer", for example, "4 || 1" means that the *Disrupter* process sleep four times longer than the *Organizer* process) for 60 msecs. Property P2 (i.e messages are looked at sufficiently often) is depicted as a percentage of Organizer actions within the traces (called "%AllOrg"), which is recorded by RuleR. The percentage of messages reach their destination is depicted as "%Msg". "%PathX" is the percentage of path that exist. The result shows that: if path existence occurs infinitely often

and the messages are looked at sufficiently often, then the number of messages eventually reach their destinations are increased. This result supports the proof of Lemma 1, 2 and 3, hence the Theorem 1. As Figure 5 shows, when the traces are longer (time = 180 msecs), the relation between all conditions (fairness and properties) engenders more confidence and the messages eventually get delivered.

5 Conclusion

We have shown that, by introducing models of message-passing dynamic networks as concurrent systems, we can use standard proof techniques for concurrent systems based on temporal logic and properties such as fairness to establish correctness (i.e the eventual delivery of all messages) of dynamic networks at an appropriate level of abstraction. Moreover, we have employed techniques recently developed in run-time verification in order to check whether implemented models of dynamic networks satisfy the required temporal properties for correct message delivery.

References

1 Kuhn, F. and Oshman, R. *Dynamic networks: models and algorithms*. SIGACT News, ACM, Vol. 42, pp. 82-96, 2011

2 Kuhn, F., Lynch, N. and Oshman, R. *Distributed computation in dynamic networks*. Proceedings of the 42nd ACM symposium on theory of computing ACM, 2010, pp. 513-522

3 O'Dell, R. and Wattenhofer, R. *Information dissemination in highly dynamic graphs*. Proceedings of the 2005 joint workshop on foundations of mobile computing. ACM, 2005, pp. 104-110

4 Clementi, A. and Pasquale, F. *Information Spreading in Dynamic Networks: An Analytical Approach*. Nikoletseas, S. and Rolim, J. D. (eds.) Theoretical Aspects of Distributed Computing in Sensor Networks. Springer Berlin Heidelberg, 2010, pp. 591-619

5 Chen, Y. and Welch, J.L. *Self-stabilizing mutual exclusion using tokens in mobile ad hoc networks*. Proceedings of the 6th international workshop on Discrete algorithms and methods for mobile computing and communications. ACM, 2002, pp. 34-42

6 Magee, J. and Kramer, J. *Concurrency: State Models and Java Programs*. Wiley, 2006

7 Leucker, M. and Schallhart, C. *A brief account of runtime verification*. The Journal of Logic and Algebraic Programming, 2009, Vol. 78(5), pp. 293 - 303

8 Barringer, H., Havelund, K., Rydeheard, D. and Groce, A. *Rule Systems for Runtime Verification: A Short Tutorial*. Bensalem, S. and Peled, D. (eds.), Runtime Verification. Springer Berlin Heidelberg, Vol. 5779, pp. 1-24, 2009

9 Kwiatkowska, M. *Survey of fairness notions*. Information and Software Technology, 1989, Vol. 31(7), pp. 371-386

10 Emerson, E.A. *Temporal and Modal Logic*. J. van Leeuwen, ed. Handbook of Theoretical Computer Science. Elsevier, 1990, Volume B: Formal Models and Semantics, pp. 995-1072.

11 Owicki, S. and Gries, D. *An axiomatic proof technique for parallel programs I*. Acta Informatica, Vol. 6(4). Springer-Verlag, 1976, pp. 319-340.

12 E.W. Dijkstra, Leslie Lamport, A.J. Martin, and E.F.M. Steffens. *On-the-Fly Garbage Collection: An Exercise in Cooperation*. Communications of the ACM, Vol. 21(11), November 1978. pp. 966-975.

13 Gries, D. *An exercise in proving parallel programs correct*. Commun. ACM, 1977, Vol. 20, pp. 921-930

Safety of Unmanned Aircraft Systems Facing Multiple Breakdowns

Patrice Carle[1], Christine Choppy[2], Romain Kervarc[1], and Ariane Piel[1,2]

1 **ONERA – The French Aerospace Lab, 91123 Palaiseau, France**
{patrice.carle,romain.kervarc,ariane.piel}@onera.fr
2 **Université Paris 13, Sorbonne Paris Cité, LIPN, CNRS UMR 7030, 93430 Villetaneuse, France**
Christine.Choppy@lipn.univ-paris13.fr

Abstract

This work deals with data analysis issues in an aeronautics context by using a formal framework relying on activity recognition techniques which are applied to the certification and safety analysis processes of Unmanned Aircraft Systems in breakdown situations. In this paper, the behaviour of these systems is modelled, simulated and studied in case of multiple failures using a complex event processing language called chronicles to describe which combinations of events in time may lead to safety breaches, and a C++ chronicle recognition library is used to implement this method.

1998 ACM Subject Classification I.2.4 Knowledge Representation Formalisms and Methods

Keywords and phrases complex event processing, safety, aeronautics, multiple breakdowns, behaviour recognition tool

Digital Object Identifier 10.4230/OASIcs.FSFMA.2013.86

1 Introduction

The wide range of the possible civil applications to the insertion of aircrafts without pilots on board in controlled or uncontrolled airspace motivates a pronounced general will for its achievement in a near future. One of the main security issues to be solved is the global consistency of the system required to operate safely an Unmanned Aircraft (UA). In the framework of operation safety analysis, we provide the possibility to detect incoherent states between the different entities making up the system. These incoherent states are formalised so as to be able to automatically recognise them through complex event processing, and hence offer the opportunity of both a self-acting surveillance and an assistance to the improvement of the system. This work relies on a fragment of the IDEAS project in charge of the Insertion of Unmanned Aircrafts in Airspace and Security, and tackles consistency problems in breakdown handling policies for UA.

The system required to safely operate an unmanned aircraft can follow several types of architecture. Our model is based on the one presented in Fig. 1. It is composed of three entities, the UA and the Remote Pilot Station (RPS), which both make up the Unmanned Aircraft System (UAS), to which is added the Air Traffic Control (ATC). All three interact via several communication links. The RPS pilots the UA via Telecommand (TC), and the UA sends information to the pilot through Telemetry (TM). In addition, the ATC and the pilot can communicate via radio (Voice) relayed by the UA.

Hence, the dynamic data flows between the agents of the system and between different systems if several UAS are considered are very elaborate. Moreover, each agent deduces

1st French Singaporean Workshop on Formal Methods and Applications 2013 (FSFMA'13).
Editors: Christine Choppy and Jun Sun; pp. 86–91

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

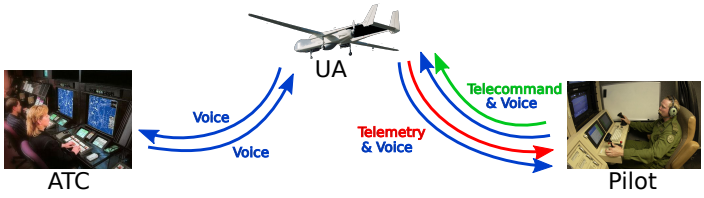


Figure 1 Architecture of the system required to operate safely an UA.

from its own observations the state of the other agents. The situation in case of a fault can therefore be very complex, and all the more so if several faults are considered. Hence, these highly automated systems are very critical, which requires the strong risk-free guarantees provided by formal methods such as our behaviour recognition technique.

In the framework of the IDEAS project, the behaviour of each entity in case of a failure has been specified [5]. In a previous paper [3], we put forward ongoing work overseeing the consistency between the three entities during the rundown urgency procedure linked to a single Telecommand (TC) failure (the pilot receives information from the UA via telemetry but cannot send out orders to it). In the present paper, we will consider the additional failure of the Voice link (the pilot and the ATC cannot communicate directly anymore), both on its own and coupled with the TC breakdown, providing a framework for an automatic monitoring of consistency in UAS. This yields an opportunity to put to use new features of the monitoring tool employed for the online analysis of the simulation.

The problem is formalised by standardising the rundown procedures from the IDEAS project into the UML language [6]. A class diagram exposed in [3] precisely describes the structure of the system. The three entities are made up of several components modelling their complex interactions precisely enough to be able to consider separately the smooth functioning of the communication links, and hence check for the consistency between the active states of the diagram. The several urgency procedures corresponding to the codified behaviours related to failures and specified in the IDEAS project are united and translated into one single state-transition diagram presented in Fig. 2. This allows to consider multiple concurrent failures which was not possible before.

This paper will start by going into some details of the preliminary semi-formal representation of the system, exposing the stakes at hand. In a second part, the system will be supervised using a behaviour recognition library, and the results will be displayed.

2 A Modelling framework

Through the class diagram and the state-transition diagram, the system has been entirely modelled. Indeed, its life cycle is mirrored by the changes in the active states of the diagram (Fig. 2). The aim is to analyse the simulation while it is running in order to draw forth certain desired or undesired behaviours.

The simulation which has been established strictly follows the behavioural guidelines specified and required by the IDEAS project. However, UA are not currently allowed to fly in controlled or uncontrolled airspace, and the regulations such as the ones studied in this paper have not yet been finalised. Our aim is to study whether certain incoherent states emerge from the simulation. Not only does behaviour recognition on running simulation allow to confirm or reject certain requirement choices, but it also provides a means to enlighten the experts in their elaboration of the regulations that should be followed, and this by bringing out and highlighting possible security breaches. In this application, the first use of our

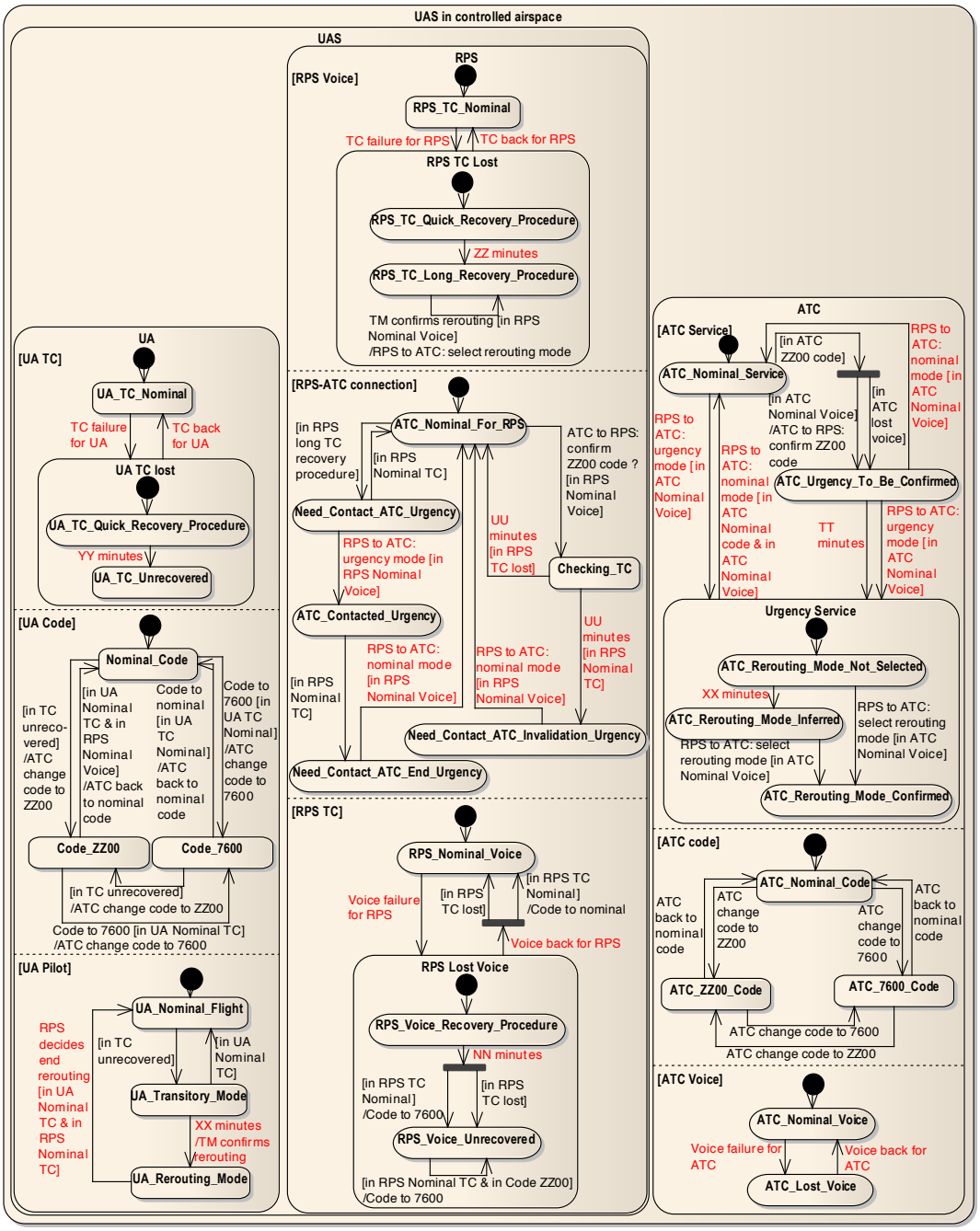


Figure 2 State-transition diagram describing the behaviour of the system.

behaviour recognition technique is to offer an assistance during the development stage of the regulations. Since multiple breakdowns are considered, the situation becomes highly complex to embrace, and such assistance is necessary.

Eventually, the remaining causes of safety breaches should only be human (i.e. due to the pilot or the air traffic controller). These have to stay in the model since they represent a reality which cannot be avoided, but they can be detected using our behaviour recognition technique. Hence, the second use of our method is the detection of the last safety breaches which cannot be prevented, in order to generate alarms and reduce the potential risks.

The first step is to specify the inconsistent states of the system which have to be averted. For example, the two following behaviours are undesired:

- *Incoherent ATC Voice*: the transponder code emitted by the UA starts indicating code 7600 at the air traffic control, which means that there is a voice failure, but the controller does not realise so, and this is expressed by the fact that the diagram does not switch to `ATC Lost Voice`.
- *Incoherent flight mode UA/ATC*: after a fault which has been solved, the UA has switched back to a nominal flight but the ATC stays in an urgency service.

Once one of these behaviours is detected, its origins have to be determined. If the cause is due to faulty behavioural guidelines, then the model has to be corrected, and, otherwise, if the source is human, it should be planned to trigger alarms warning the pilot and/or the air traffic controller of the situation.

In order to exploit these examples and to allow direct simulation, the UML diagram has been implemented in C++ using the Meta State Machine (MSM) library [4] of boost (Version 1.53.0) which provides a straightforward way to define state machines. So as to simulate the life cycle of the system, scenarios activate the red transitions of the diagram of Fig. 2 (the other transitions are triggered by events automatically generated by the diagram), thus providing a complete modelling framework. We thus obtain a direct simulation of the system that we want to supervise using failure detection.

3 Behaviour recognition with CRL (Chronicle Recognition Library)

To perform this supervision, it is necessary to be able to formally express failures as behaviours which are to be recognised. Monitoring is then needed to allow an *online* recognition of *all* the occurrences of the described behaviours during the running simulation, which are central issues linked to complex event processing. These two requirements are fulfilled by a temporal language — the chronicle language, which syntax and semantics are partly defined in [2] — and its associated recognition tool.

This language allows to formally depict system behaviours. Arrangements of events are described: a chronicle can be a single event, the conjunction of two chronicles, the disjunction of two chronicles, the sequence of two chronicles or the absence of a chronicle during another chronicle. In addition, temporal constraints between chronicles or on the length of time of the recognition of a chronicle may be specified. For instance, let `E` and `F` be single events and δ a real number, chronicle (`E` then δ) $-$ [`F`] corresponds to event `E` followed by δ units of time during which no event `F` occurs.

The first step is therefore to write down chronicles which will oversee the system. It is necessary to take into account isolated events since we want to be able to recognise the chronicles online, so the set of events considered to build these chronicles are the entrances in and exits from the different possibly active states of the diagram of Fig. 2.

The two unwanted behaviours briefly described in Sec. 2 may be formally expressed by the following chronicles:

- *Incoherent ATC Voice*

  (`to_ATC_Nominal_Code to_ATC_7600_Code` then 5) $-$ [`to_ATC_Lost_Voice`]
- *Incoherent flight mode UA/ATC*

  (`from_UA_Nominal_Flight`
  ((`to_UA_Nominal_Flight` then 10) $-$ [`from_UA_Nominal_Flight`]))
  $-$[`to_ATC_Nominal_Service`]

Once the behaviours to be recognised have been formalised in the chronicle language, the simulation is run along different scenarios which are supervised by a behaviour recognition tool in charge of bringing to light any incoherent state described by the written chronicles. Such a tool, designed on the basis of duplicating automata and called Chronicle Recognition System (CRS/ONERA), has been developed by the ONERA in the late 1990s [1]. A new recognition tool which algorithms directly result from the set semantics of the chronicle language is developed during this Ph.D. thesis. This tool, called Chronicle Recognition Library (CRL), implemented in C++, is described in greater detail in [3]. Chronicles are plugged into the program, and then, gradually as events flow in, the program gives the set of all the recognitions of each chronicle, specifying for each recognition which events lead to it.

Let us now run this tool on two scenarios, looking for recognitions of the two previously specified chronicles.

Consider, to start with, an overly simple story line as an instructive example: there is a voice failure, which is only acknowledged by the pilot (event `Voice failure for RPS`). The simulation is run with this single event. The evolution of the entrances in and exits from the active states of the diagram are then plugged into CRL which generates the following result:

```
t = 0  Engine created
t = 0  Added chronicle :
        ([(to_ATC_Nominal_Code to_ATC_7600_Code) + 5] − to_ATC_Lost_Voice)
t = 0  Added Event : Voice_failure_for_RPS
t = 0  Added Event : from_RPS_Nominal_Voice
t = 0  Added Event : to_RPS_Voice_Recovery_Procedure
t = 4  Added Event : from_RPS_Voice_Recovery_Procedure
t = 4  Added Event : to_RPS_Voice_Unrecovered
t = 4  Added Event : from_Nominal_Code
t = 4  Added Event : to_Code_7600
t = 4  Added Event : from_ATC_Nominal_Code
t = 4  Added Event : to_ATC_7600_Code
t = 9  Chronicle recognition :
        ([(to_ATC_Nominal_Code to_ATC_7600_Code) + 5] − to_ATC_Lost_Voice)
        Reco Set = {⟨⟨(to_ATC_Nominal_Code, 0), (to_ATC_7600_Code, 4)⟩, (t, 9)⟩}
```

Chronicle *Incoherent ATC Voice* has been recognised: `to_ATC_Nominal_Code` at time 0 has been followed by `to_ATC_7600_Code` at time 4, and, until time 9, the forbidden events have not occurred. Thanks to event historisation, it can be diagnosed that the source of the inconsistency is a lack of attention from the air traffic controller. An alarm should be triggered by the chronicle to warn the ATC of the situation and attempt to restore a correct situation.

Let us now consider a second scenario involving multiple breakdowns: a voice failure acknowledged both by the pilot and the ATC (events `Voice failure for RPS` and `Voice failure for ATC`) is shortly followed by a TC failure recognised both by the UA and the pilot (events `TC failure for UA` and `TC failure for RPS`). However, the TC is restored 15 minutes later (events `TC back for UA` and `TC back for RPS`), at which point the pilot decides that the situation is not too alarming (a voice failure can indeed be considered as such) and therefore orders the UA back to a nominal flight (event `RPS decides end rerouting`). When the simulation is run with these events, the following result is identified by CRL:

```
⋮
t = 65  Chronicle recognition :
            ([from_UA_Nominal_Flight ([to_UA_Nominal_Flight + 10]
            − from_UA_Nominal_Flight)] − to_ATC_Nominal_Service)
            Reco Set = {⟨(from_UA_Nominal_Flight, 35), ⟨(to_UA_Nominal_Flight, 55), (t, 65)⟩⟩}
```

Chronicle *Incoherent flight mode UA/ATC* is recognised. This time, the inconsistency is not due to human error, which means that the model has to be corrected: it is brought to light that a transition is missing in the modelling of the ATC between `Urgency service` and `ATC_Nominal_Service`. Indeed, the ATC should be able to switch back to a nominal service even though no radio communication with the pilot is available. A transition triggered by the exit of `ATC_ZZ00_Code` (indicating the end of the TC failure) has therefore to be added. Once this improvement has been completed and the system and/or the procedure have been modified, running the simulation using the new model on the same scenario does not produce any recognition anymore, ascertaining that the behaviour has been rightfully corrected.

4 Conclusion and perspectives

In conclusion, we provide in this paper a complex event processing framework applied to monitor safety for Unmanned Aircraft Systems in case of one or multiple breakdowns. The behaviour of the UAS is completely modelled in a UML diagram which is implemented in C++. A temporal language, the chronicle language, is used to specify the inconsistent states which would lead to safety breaches and which therefore have to be avoided. CRL, a C++ library, allows direct analysis of simulation data. The achievement is twofold: an assistance to regulation development is provided, and the remaining safety breaches which cannot be totally prevented can be made to trigger alarms.

Among the numerous possible future directions for this work, we plan to continue the extension of the chronicle language in order to increase its expressivity and hence be able to deal with a wider spectrum of applications. For instance, we intend to formalise a notion of actions triggered by successful recognitions which would allow, for example, the formalisation of the alarm generation in the application presented in this paper. In addition, the choice and writing of the chronicles to be recognised is currently completed by an expert by hand. It would be desirable to develop an assistance tool for the generation of chronicles, so as to get closer to an exhaustive covering of the situations to be recognised.

Acknowledgements. The authors thank J. Bourrely for his help and support for this application, and T. Lang and C. Le Tallec for their useful insights on Unmanned Aircrafts.

References

1 P. Carle, P. Benhamou, F.-X. Dolbeau, and M. Ornato. La reconnaissance d'intentions comme dynamique des organisations. In *6èmes Journées Francophones pour l'Intelligence Artificielle Distribuée et les Systèmes Multi-Agents (JFIADSMA'98)*, 1998.

2 Patrice Carle, Christine Choppy, and Romain Kervarc. Behaviour recognition using chronicles. In *Proc. 5th IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 100–107, 2011.

3 Patrice Carle, Christine Choppy, Romain Kervarc, and Ariane Piel. Handling Breakdowns in Unmanned Aircraft Systems. In *18th International Symposium on Formal Methods - Doctoral Symposium*, 2012.

4 Christophe Henry. "MSM library of boost". `www.boost.org/doc/libs/1_48_0/libs/msm/doc/HTML/index.html`, 2011.

5 Thibault Lang. IDEAS–T1.1: Architecture de système de drone et scénarios de missions. Technical report, 2009.

6 "OMG Unified Modeling Language™(OMG UML), Superstructure, Version 2.4.1", 2011.