

Journal Subline

LNCS 7400

Wil M.P. van der Aalst · Marco Ajmone Marsan  
Giuliana Franceschinis · Jetty Kleijn  
Lars Michael Kristensen  
Guest Editors

# Transactions on **Petri Nets** and Other Models of Concurrency VI

Kurt Jensen  
Editor-in-Chief

 Springer

*Commenced Publication in 1973*

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

## Editorial Board

David Hutchison

*Lancaster University, UK*

Takeo Kanade

*Carnegie Mellon University, Pittsburgh, PA, USA*

Josef Kittler

*University of Surrey, Guildford, UK*

Jon M. Kleinberg

*Cornell University, Ithaca, NY, USA*

Friedemann Mattern

*ETH Zurich, Switzerland*

John C. Mitchell

*Stanford University, CA, USA*

Moni Naor

*Weizmann Institute of Science, Rehovot, Israel*

Oscar Nierstrasz

*University of Bern, Switzerland*

C. Pandu Rangan

*Indian Institute of Technology, Madras, India*

Bernhard Steffen

*TU Dortmund University, Germany*

Madhu Sudan

*Microsoft Research, Cambridge, MA, USA*

Demetri Terzopoulos

*University of California, Los Angeles, CA, USA*

Doug Tygar

*University of California, Berkeley, CA, USA*

Moshe Y. Vardi

*Rice University, Houston, TX, USA*

Gerhard Weikum

*Max Planck Institute for Informatics, Saarbruecken, Germany*

Kurt Jensen Wil M.P. van der Aalst  
Marco Ajmone Marsan  
Giuliana Franceschinis Jetty Kleijn  
Lars Michael Kristensen (Eds.)

# Transactions on Petri Nets and Other Models of Concurrency VI

## Editor-in-Chief

Kurt Jensen  
University of Aarhus  
Faculty of Science, Department of Computer Science  
IT-parken, Aabogade 34, 8200 Aarhus N, Denmark  
E-mail: [kjensen@cs.au.dk](mailto:kjensen@cs.au.dk)

## Guest Editors

Wil M.P. van der Aalst  
Eindhoven University of Technology, The Netherlands  
E-mail: [w.m.p.v.d.aalst@tue.nl](mailto:w.m.p.v.d.aalst@tue.nl)

Marco Ajmone Marsan  
Politecnico di Torino, Italy  
E-mail: [ajmone@polito.it](mailto:ajmone@polito.it)

Giuliana Franceschinis  
Università del Piemonte Orientale "Amedeo Avogadro", Italy  
E-mail: [giuliana.franceschinis@di.unipmn.it](mailto:giuliana.franceschinis@di.unipmn.it)

Jetty Kleijn  
LIACS, Leiden University, The Netherlands  
E-mail: [kleijn@liacs.nl](mailto:kleijn@liacs.nl)

Lars Michael Kristensen  
Bergen University College, Norway  
E-mail: [lmkr@hib.no](mailto:lmkr@hib.no)

ISSN 0302-9743 (LNCS)	e-ISSN 1611-3349 (LNCS)
ISSN 1867-7193 (ToPNoC)	e-ISSN 1867-7746 (ToPNoC)
ISBN 978-3-642-35178-5	e-ISBN 978-3-642-35179-2
DOI 10.1007/978-3-642-35179-2	
Springer Heidelberg Dordrecht London New York	

Library of Congress Control Number: 2012952593

CR Subject Classification (1998): D.2.2-4, F.1.1, D.3.2, C.2.1-4, I.2.2, C.4, H.3.5

© Springer-Verlag Berlin Heidelberg 2012

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

The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

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

Printed on acid-free paper

Springer is part of Springer Science+Business Media ([www.springer.com](http://www.springer.com))



## Preface by Editor-in-Chief

The sixth issue of LNCS Transactions on Petri Nets and Other Models of Concurrency (ToPNoC) contains revised and extended versions of a selection of the best papers from the workshops and tutorials held at the 32nd International Conference on Application and Theory of Petri Nets and Concurrency, in Newcastle upon Tyne, UK, June 20–24, 2011, edited by Wil van der Aalst and Jetty Kleijn, and a special section on Networks, Protocols, and Services, edited by Giuliana Franceschinis, Lars Michael Kristensen, and Marco Ajmone Marsan. It also contains a paper that was submitted to ToPNoC directly through the regular submission track.

I would like to thank the five guest editors of this special issue: Wil van der Aalst, Jetty Kleijn, Giuliana Franceschinis, Lars Michael Kristensen, and Marco Ajmone Marsan. Moreover, I would like to thank all authors, reviewers, and the organizers of the Petri net conference satellite workshops, without whom this issue of ToPNoC would not have been possible.

August 2012

Kurt Jensen  
Editor-in-Chief

LNCS Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)

# LNCS Transactions on Petri Nets and Other Models of Concurrency: Aims and Scope

ToPNoC aims to publish papers from all areas of Petri nets and other models of concurrency ranging from theoretical work to tool support and industrial applications. The foundation of Petri nets was laid by the pioneering work of Carl Adam Petri and his colleagues in the early 1960s. Since then, an enormous amount of material has been developed and published in journals and books and presented at workshops and conferences.

The annual International Conference on Application and Theory of Petri Nets and Concurrency started in 1980. The International Petri Net Bibliography maintained by the Petri Net Newsletter contains close to 10,000 different entries, and the International Petri Net Mailing List has 1,500 subscribers.

For more information on the International Petri Net community, see:  
<http://www.informatik.uni-hamburg.de/TGI/PetriNets/>

All issues of ToPNoC are LNCS volumes. Hence they appear in all large libraries and are also accessible in LNCS Online (electronically). It is possible to subscribe to ToPNoC without subscribing to the rest of LNCS.

ToPNoC contains:

- revised versions of a selection of the best papers from workshops and tutorials concerned with Petri nets and concurrency;
- special issues related to particular subareas (similar to those published in the *Advances in Petri Nets* series);
- other papers invited for publication in ToPNoC; and
- papers submitted directly to ToPNoC by their authors.

Like all other journals, ToPNoC has an Editorial Board, which is responsible for the quality of the journal. The members of the board assist in the reviewing of papers submitted or invited for publication in ToPNoC. Moreover, they may make recommendations concerning collections of papers for special issues. The Editorial Board consists of prominent researchers within the Petri net community and in related fields.

## Topics

System design and verification using nets; analysis and synthesis, structure and behavior of nets; relationships between net theory and other approaches; causality/partial order theory of concurrency; net-based semantical, logical and algebraic calculi; symbolic net representation (graphical or textual); computer tools for nets; experience with using nets, case studies; educational issues related to nets; higher level net models; timed and stochastic nets; and standardization of nets.

Applications of nets to: biological systems, defence systems, e-commerce and trading, embedded systems, environmental systems, flexible manufacturing systems, hardware structures, health and medical systems, office automation, operations research, performance evaluation, programming languages, protocols and networks, railway networks, real-time systems, supervisory control, telecommunications, and workflow.

For more information about ToPNoC, please see: [www.springer.com/lncs/topnoc](http://www.springer.com/lncs/topnoc)

## **Submission of Manuscripts**

Manuscripts should follow LNCS formatting guidelines, and should be submitted as PDF or zipped PostScript files to [ToPNoC@cs.au.dk](mailto:ToPNoC@cs.au.dk). All queries should be addressed to the same e-mail address.

# LNCS Transactions on Petri Nets and Other Models of Concurrency: Editorial Board

## Editor-in-Chief

Kurt Jensen, Denmark (<http://person.au.dk/en/kjensen@cs.au.dk>)

## Associate Editors

Grzegorz Rozenberg, The Netherlands  
Jonathan Billington, Australia  
Susanna Donatelli, Italy  
Wil van der Aalst, The Netherlands

## Editorial Board

Didier Buchs, Switzerland  
Gianfranco Ciardo, USA  
José-Manuel Colom, Spain  
Jörg Desel, Germany  
Michel Diaz, France  
Hartmut Ehrig, Germany  
Jorge C.A. de Figueiredo, Brazil  
Luis Gomes, Portugal  
Roberto Gorrieri, Italy  
Serge Haddad, France  
Xudong He, USA  
Kees van Hee, The Netherlands  
Kunihiko Hiraishi, Japan  
Gabriel Juhas, Slovak Republic  
Jetty Kleijn, The Netherlands  
Maciej Koutny, UK

Lars M. Kristensen, Norway  
Charles Lakos, Australia  
Johan Lilius, Finland  
Chuang Lin, China  
Satoru Miyano, Japan  
Madhavan Mukund, India  
Wojciech Penczek, Poland  
Laure Petrucci, France  
Lucia Pomello, Italy  
Wolfgang Reisig, Germany  
Manuel Silva, Spain  
P.S. Thiagarajan, Singapore  
Glynn Winskel, UK  
Karsten Wolf, Germany  
Alex Yakovlev, UK

# Preface by Guest Editors

This issue of ToPNoC consists of three parts:

1. the first part comprises the revised versions of a selection of the best papers from the workshops and tutorials held at the 32nd International Conference on Application and Theory of Petri Nets and Concurrency, in Newcastle upon Tyne, UK, June 20–24, 2011, and has been edited by Wil van der Aalst and Jetty Kleijn;
2. the second part consists of papers selected for a special section on Networks, Protocols, and Services and has been edited by Giuliana Franceschinis, Lars Michael Kristensen, and Marco Ajmone Marsan;
3. the third part is formed by the paper “Aggregating Causal Runs into Workflow Nets” by Boudewijn van Dongen, Jörg Desel, and Wil van der Aalst, submitted to ToPNoC directly through the regular submission track.

The remainder of this preface introduces these three parts.

## Best Workshop Papers from Petri Nets 2011

This part contains revised and extended versions of a selection of the best workshop papers presented at the 32nd International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets 2011).

We, Wil van der Aalst and Jetty Kleijn, are indebted to the program committees of the workshops and in particular their chairs. Without their enthusiastic work this volume would not have been possible. Many members of the program committees participated in reviewing the extended versions of the papers selected for this issue. The following workshops were asked for their strongest contributions:

- PNSE 2011: International Workshop on Petri Nets and Software Engineering (chairs: Michael Duvigneau, Kunihiko Hiraishi, and Daniel Moldt),
- BioPPN 2011: International Workshop on Biological Processes and Petri Nets (chairs: Monika Heiner and Hiroshi Matsuno),
- ART 2011: Applications of Region Theory (chairs: Jörg Desel and Alex Yakovlev),
- CompoNet 2011: International Workshop on Petri Nets Compositions (chairs: Hanna Kludel and Franck Pommereau),
- SUMo 2011: Scalable and Usable Model Checking for Petri Nets and Other Models of Concurrency (chair: Didier Buchs)

The best papers of these workshops were selected in close cooperation with their chairs. The authors were invited to improve and extend their results where possible, based on the comments received before and during the workshop. The

resulting revised submissions were reviewed by three to five referees. We followed the principle of also asking for fresh reviews of the revised papers, i.e., from referees who had not been involved initially in reviewing the original workshop contribution. All papers went through the standard two-stage journal reviewing process and eventually eight were accepted after rigorous reviewing and revising. Presented are a variety of high-quality contributions, ranging from model checking and system verification to synthesis, and from work on Petri-net-based standards and frameworks to innovative applications of Petri nets and other models of concurrency.

The paper by Josep Carmona, *The Label Splitting Problem*, revisits label splitting, a technique to satisfy the synthesis conditions through renaming of problematic labels. To be applicable, the classical theory of regions relies on stringent conditions on the input automaton. Although some relaxations on these restrictions were proposed earlier, in general not every automaton can be synthesized while preserving its behavior using classical approaches. The paper formalizes the problem of label splitting and proposes extensions that improve the applicability of the theory of regions.

The paper *Distributed Control of Discrete-Event Systems: A First Step*, by Philippe Darondeau and Laurie Ricker, is concerned with the synthesis of distributed control implemented by asynchronous message passing automata. A survey of discrete-event systems control is provided. Also distributed Petri nets and their synthesis and translation to asynchronous communicating automata are discussed. Then distributed Petri net synthesis techniques are applied to synthesize distributed supervisory controllers that avoid deadlocks or enforce home states. An algorithm is proposed and its limitations are discussed. As an illustration of the method the paper experiments using the 3-dining philosophers problem, which leads to three (new) distributed solutions of this problem.

The third paper, *Extending PNML Scope: A Framework to Combine Petri Nets Types* by Lom-Messan Hillah, Fabrice Kordon, Charles Lakos, and Laure Petrucci, is concerned with Petri net extensions in the context of the International Standard on Petri nets, ISO/IEC 15909, which comprises three parts. ISO/IEC 15909-3 aims at defining extensions on the whole family of Petri nets. This paper elaborates on an extension framework for the third part of the standard and shows how priorities, times, and inhibitor arcs can be added in the context of an interleaving semantics.

Ekkart Kindler, in his paper *Modelling Local and Global Behaviour: Petri Nets and Event Coordination*, introduces the general idea of Event Coordination, Notation (ECNO) and of ECNO nets. ENCO can be used to define the global behavior of a software system on top of existing class diagrams. One of the major objectives of this notation was to make it easy to integrate model-based code generation with existing structural models, with existing code, and other behavioral models. Basically, the ENCO net describes how the local behavior of the individual parts of the software is coordinated. ECNO nets have been implemented as a Petri net type for the ePNK tool, together with a code

generator that produces code that can be executed by the ECNO execution engine.

In *Model Checking Using Generalized Testing Automata*, Ala-Eddine Ben Salem, Alexandre Duret-Lutz, and Fabrice Kordon continue earlier work on LTL model checking of stuttering-invariant properties. The automata-theoretic approach to model checking of linear-time properties relies on  $\omega$ -automata to represent infinite executions of a model. Different types of automata have been used and the current paper proposes to combine features from Testing Automata, and Transition-Based Generalized Büchi Automata, which leads to the introduction of Transition-Based Generalized Testing Automata. Experiments on benchmark models show that TGTA outperform the other approaches in most of the cases.

The paper *A Domain Specific Language Approach for Genetic Regulatory Mechanisms Analysis*, by Nicolas Sedlmajer, Didier Buchs, Steve Hostettler, Alban Linard, Edmundo López Bóbeda, and Alexis Marechal, describes an approach based on Domain Specific Languages (DSLs). The authors provide a language called GReg that aims to describe genetic regulatory mechanisms and their properties. The language is designed to enable model checking. GReg's objective is to shield the user from the complexity of those underlying techniques. The resulting models can be used to discover emerging properties arising from the complex interactions between biological components.

In his paper *Verifying Parallel Algorithms and Programs Using Coloured Petri Nets*, Michael Westergaard describes an approach for the automatic extraction of Coloured Petri Net models from parallel algorithms and programs where processes communicate via shared memory. This makes it possible to verify software using a formal model obtained from runnable code. An implementation of the translation is presented. Moreover, the technique proposed also supports model-driven development. Consequently, extraction of a model from an abstract description and generation of correct implementation code can be combined.

The last paper based on the best papers from the workshops held at the 32nd International Conference on Application and Theory of Petri Nets and Other Models of Concurrency is of a different flavor as it is based on a competition held in the context of the SUMo 2011 workshop. The paper *Report on the Model Checking Contest at Petri Nets 2011*, by Fabrice Kordon, Alban Linard, Didier Buchs, Maximilien Colange, Sami Evangelista, Kai Lampka, Niels Lohmann, Emmanuel Paviot-Adet, Yann Thierry-Mieg, and Harro Wimmel, presents the results of this competition. The participating tools were compared on several tests (state space generation, deadlock detection and reachability analysis) run on a set of common models (Place/Transition and Symmetric Petri nets). The collected data gave some hints about the way techniques scale up depending on both the property investigated and the characteristics of the model. This paper also presents the lessons learned from the organizers' point of view and lists enhancements required for future Model Checking Contests.

## Special Section on Networks, Protocols, and Services

This part of the present ToPNoC issue is dedicated to papers that focus on Petri net-based techniques and technologies, as well as other models of concurrency, and their applications to the analysis and design of networks, protocols, and services. Computer and telecommunication networks, together with their protocols, constitute key building blocks of most modern IT systems, since they define the infrastructures and the services that make possible the cooperation of users, be they human or machine, through the exchange of information. The engineering of networks, protocols, and services supporting today's advanced use of information technology is a challenging discipline, which requires careful behavioral modelling and validation. This makes networks, protocols, and services an important application domain for the use of Petri net techniques, as well as other models of concurrency.

This special section is based on papers submitted through an open call for contributions and invitation to selected researchers in the application domain. All papers went through a two-stage reviewing process and five papers were accepted for publication.

In *Modelling and Formal Verification of the NEO Protocol*, Christine Choppy, Anna Dedova, Sami Evangelista, Kaïs Klai, Laure Petrucci, and Samir Youcef present their work on the practical application of high-level Petri nets and a suite of supporting computer tools for the modelling and verification of a protocol for the management of large distributed databases. The Petri nets models are constructed based on a reverse-engineering approach from source code, and state space exploration is being used to analyze reliability properties of the election and bootstrap phases of the NEO protocol. One important finding is the identification of several aspects where the present NEO protocol implementation can be improved.

The paper by Sonya Arnold and Jonathan Billington, *An Initial Coloured Petri Net Model of the Hypertext Transfer Protocol Operating Over the Transmission Control Protocol*, concentrates on the use of Coloured Petri Nets (CPNs) for modelling essential features of the Hypertext Transfer Protocol (HTTP), which is currently undergoing revision by the Internet Engineering Task Force. A CPN model of the HTTP protocol is presented that relies on an explicit and rigorous modelling of the service provided by the underlying transport protocol. State spaces and standard behavioral properties of Petri nets are being used to verify liveness and termination properties of HTTP and determine tight upper bounds on interface buffers.

The paper *Privacy Compliance Verification in Cryptographic Protocols* by Suriadi Suriadi, Chun Ouyang, and Ernest Foo focuses on the use of CPNs for constructing executable formal models of privacy enhancing protocols. A representative protocol in the form of the Private Information Escrow Bound to Multiple Conditions Protocol (PIEMCP) is considered. A CPN model of the PIEMCP protocol together with associated modelling techniques is presented and then temporal logic and model checking techniques are used to formulate and verify privacy compliance properties under a range of attack scenarios.



Dario Bruneo, Francesco Longo, and Antonio Puliafito, in their paper *Modeling Energy-Aware Cloud Federations with SRNs*, develop a methodology based on stochastic reward nets to evaluate management policies in the context of infrastructure as a service clouds. Models are developed for cloud infrastructure components and it is shown how these can be used to reason analytically about energy efficiency in hybrid clouds consisting of cooperating private and public clouds. In this context, federation policies allow clouds to cooperate to handle peak request periods for virtual machines while virtual server consolidation policies allow infrastructure shutdown of data centre services.

In their paper *A SAN-Based Modeling Approach to Performance Evaluation of an IMS-Compliant Conferencing Framework*, Stefano Marrone, Nicola Mazzocca, Roberto Nardone, Roberta Presta, Simon Pietro Romano, and Valeria Vittorini propose a component- and template oriented modelling approach based on Stochastic Activity Networks (SANs) for reasoning about the performance of a distributed IP-based multimedia conferencing framework. Performance models built using the proposed approach are validated via comparison with performance measures extracted from a deployed implementation, demonstrating the accuracy of the result obtained with the approach.

## Regular Paper

An earlier version of the paper *Aggregating Causal Runs into Workflow Nets* by B.F. van Dongen, J. Desel, and W.M.P. van der Aalst was submitted to the ART workshop at Petri Nets 2011 and was suggested as a paper for ToPNoC by the workshop organizers. However, to avoid any conflict of interest, the paper was submitted to ToPNoC directly through the regular submission track. This was done because one of the authors (Wil van der Aalst) was also editor of the special issue based on Petri Nets 2011. The reviewing process was handled by the Editor-in-Chief, Kurt Jensen.

The paper *Aggregating Causal Runs into Workflow Nets* provides three algorithms for deriving marked Petri nets from sets of partially-ordered causal runs. The three aggregation algorithms differ with respect to the assumptions about the information contained in the causal runs. Unlike most papers on process mining, the authors use the assumption that events are logged as partial orders instead of linear traces. Although the work is inspired by applications in the process mining and workflow domains, the results are generic and can be applied in other application domains.

## Thanks

As guest editors, we would like to thank all authors and referees who have contributed to this issue: not only is the quality of this volume the result of the high scientific value of their work, but we would also like to acknowledge the excellent cooperation throughout the whole process that has made our work a pleasant task. Finally, we would like to pay special tribute to Lars Madsen of

Aarhus University and Ine van der Ligt of Eindhoven University of Technology, who provided technical support for the composition of this volume and interacted with the authors. We are also grateful to the Springer/ToPNoC team for the final production of this issue.

August 2012

Wil van der Aalst  
Jetty Kleijn  
Giuliana Franceschinis  
Lars Michael Kristensen  
Marco Ajmone Marsan

Guest Editors, Sixth Issue of ToPNoC

# Organization

## Guest Editors

Wil van der Aalst, The Netherlands  
Jetty Kleijn, The Netherlands  
Giuliana Franceschinis, Italy  
Lars Michael Kristensen, Norway  
Marco Ajmone Marsan, Italy

## Co-chairs of the Workshops

Didier Buchs (Switzerland)  
Jörg Desel (Germany)  
Michael Duvigneau (Germany)  
Monika Heiner (Germany)  
Kunihiko Hiraishi (Japan)  
Hanna Klaudel (France)  
Hiroshi Matsuno (Japan)  
Daniel Moldt (Germany)  
Franck Pommereau (France)  
Alex Yakovlev (UK)

## Referees

Arseniy Alekseyev  
David Angeli  
Alessio Angius  
Gianfranco Balbo  
Kamel Barkaoui  
Robin Bergenthum  
Marco Bernardo  
Marcello Bonsangue  
Dragan Bosnacki  
Hanifa Boucheneb  
Marco Beccuti  
Didier Buchs  
Lawrence Cabac  
Massimo Canonico  
Claudine Chaouiya  
Silvano Chiaradonna  
Carla Fabiana Chiasserini

Piotr Chrzastowski-Wachtel  
Gianfranco Ciardo  
Daniele Codetta-Raiteri  
David D'Aprile  
Jörg Desel  
Boudewijn van Dongen  
Claude Dutheillet  
Sami Evangelista  
Guy Gallasch  
Steven Gordon  
Boudewijn Haverkort  
Xudong He  
Kunihiko Hiraishi  
Tommi Junttila  
Hanna Klaudel  
Jetty Kleijn  
Michał Knapik

XVIII Organization

Fabrice Kordon  
Marta Koutny  
Lars Michael Kristensen  
Luciano Lavagno  
Didier Le Botlan  
Charles Lakos  
Chen Li  
Johan Lilius  
Robert Lorenz  
Michela Meo  
Artur Meski  
Patrice Moreaux  
Berndt Müller  
Aurélien Naldi  
Meriem Ouederni

Laure Petrucci  
Franck Pommereau  
Louchka Popova-Zeugmann  
Jean François Pradat-Peyre  
Jean-Francois Raskin  
Markus Siegle  
Kent Inge F. Simonsen  
Andrea Turrini  
Antti Valmari  
Michael Westergaard  
Manuel Wimmer  
Shingo Yamaguchi  
Shaofa Yang  
Mengchu Zhou

# Table of Contents

## Best Workshop Papers from Petri Nets 2011

The Label Splitting Problem . . . . .	1
<i>Josep Carmona</i>	
Distributed Control of Discrete-Event Systems: A First Step . . . . .	24
<i>Philippe Darondeau and Laurie Ricker</i>	
Extending PNML Scope: A Framework to Combine Petri Nets Types . . .	46
<i>Lom-Messan Hillah, Fabrice Kordon, Charles Lakos, and Laure Petrucci</i>	
Modelling Local and Global Behaviour: Petri Nets and Event Coordination . . . . .	71
<i>Ekkart Kindler</i>	
Model Checking Using Generalized Testing Automata . . . . .	94
<i>Ala-Eddine Ben Salem, Alexandre Duret-Lutz, and Fabrice Kordon</i>	
A Domain Specific Language Approach for Genetic Regulatory Mechanisms Analysis . . . . .	123
<i>Nicolas Sedlmajer, Didier Buchs, Steve Hostettler, Alban Linard, Edmundo López Bóbeda, and Alexis Marechal</i>	
Verifying Parallel Algorithms and Programs Using Coloured Petri Nets . . . . .	146
<i>Michael Westergaard</i>	
Report on the Model Checking Contest at Petri Nets 2011 . . . . .	169
<i>Fabrice Kordon, Alban Linard, Didier Buchs, Maximilien Colange, Sami Evangelista, Kai Lampka, Niels Lohmann, Emmanuel Paviot-Adet, Yann Thierry-Mieg, and Harro Wimmel</i>	

## Special Section on Networks, Protocols, and Services

Modelling and Formal Verification of the NEO Protocol . . . . .	197
<i>Christine Choppy, Anna Dedova, Sami Evangelista, Kais Klai, Laure Petrucci, and Samir Youcef</i>	
An Initial Coloured Petri Net Model of the Hypertext Transfer Protocol Operating over the Transmission Control Protocol . . . . .	226
<i>Sonya Arnold and Jonathan Billington</i>	

Privacy Compliance Verification in Cryptographic Protocols . . . . . 251  
*Suriadi Suriadi, Chun Ouyang, and Ernest Foo*

Modeling Energy-Aware Cloud Federations with SRNs . . . . . 277  
*Dario Bruneo, Francesco Longo, and Antonio Puliafito*

A SAN-Based Modeling Approach to Performance Evaluation  
of an IMS-Compliant Conferencing Framework . . . . . 308  
*Stefano Marrone, Nicola Mazzocca, Roberto Nardone,  
Roberta Presta, Simon Pietro Romano, and Valeria Vittorini*

**Regular Paper**

Aggregating Causal Runs into Workflow Nets . . . . . 334  
*Boudewijn F. van Dongen, Jörg Desel, and Wil M.P. van der Aalst*

**Author Index** . . . . . 365

# The Label Splitting Problem

Josep Carmona

Universitat Politècnica de Catalunya, Spain  
jcarmona@lsi.upc.edu

**Abstract.** The theory of regions was introduced by Ehrenfeucht and Rozenberg in the early nineties to explain how to derive (synthesize) an event-based model (e.g., a Petri net) from an automaton. To be applicable, the theory relies on stringent conditions on the input automaton. Although some relaxation on these restrictions was done in the last decade, in general not every automaton can be synthesized while preserving its behavior. A crucial step for a non-synthesizable automaton is to transform it in order to satisfy the synthesis conditions. This paper revisits label splitting, a technique to satisfy the synthesis conditions through renaming of problematic labels. For the first time, the problem is formally characterized and its optimality addressed. Some extensions and applications of the label splitting are presented to illustrate the significance of this technique.

## 1 Introduction

The use of *formal models* as mathematical descriptions of software or hardware systems opens the door for the adoption of formal methods in many phases of the design of a complex system. This in turn allows incorporating computer-aided techniques in the process, including verification and performance evaluation, disciplines that nowadays resort heavily in simulation. Formal models can be either state-based (i.e., automata-like) or event-based (e.g., Petri nets), and the correspondence between models is a challenging problem that we address in this paper.

The *synthesis problem* [9] consists in building a Petri net [12,14,15] that has a behavior equivalent to a given automaton (*transition system*). The problem was first addressed by Ehrenfeucht and Rozenberg [10] introducing *regions* to model the sets of states that characterize places in the Petri net. The theory is applicable to *elementary transition systems*, a proper subclass of transition systems where additional conditions are required, and for which the synthesis produces a Petri net with isomorphic behavior. These restrictions were significantly relaxed in [7], introducing the subclass of *excitation-closed transition systems*, where not isomorphism but *bisimilarity* is guaranteed. In an excitation-closed transition system, for every event the set of states where the event is enabled should be equal to the intersection of *pre-regions* (regions where the event exits) of the event. The theory of this paper relates to the subclass of excitation-closed transition systems.

When synthesis conditions do not hold, the Petri net derived might have a behavior in general incomparable to the initial behavior [5], and therefore any faithful use of such Petri net may be hampered. To overcome this problem, one might force the synthesis conditions by transforming the initial transition system. The work in [7] was the first one in addressing this problem, introducing *label splitting* as a technique that can be applied when excitation-closure is not satisfied. The technique is based on renaming the labels of a particular event  $e$  in the transition system: given the whole set of occurrences of the event, these are relabeled into different labels  $e_1, \dots, e_k$ , thus preserving the event name but considering each new copy as a new event with respect to the synthesis conditions.

The new events produced by the label splitting technique increase the complexity of the Petri net derived: each new copy will be transformed into a transition, and hence the *label splitting problem* is to find a sequence of splittings that induces the minimal number of transitions in the derived Petri net. The motivation for this minimization is twofold: first, in many applications the Petri net derived is a valuable graphical description of (part of) a system, and therefore its visualization will benefit from having the minimal number of nodes. Second, by deriving a simpler model, the complexity of algorithms that take this model as input may be alleviated when the size is minimal. The technique presented in [7] only presented the label splitting technique as a heuristic to progress into excitation-closure, i.e., it never considered the optimal application of the technique.

The label splitting technique presented in this paper is a particular one: it is defined on the sets of states computed when searching for regions in state-based synthesis methods [5,7]. These sets, called *essential*, are the building blocks used in this paper to decide which labels to split. The methods for label splitting in the aforementioned papers also use the essential sets for label splitting, but as described previously, only in a heuristic manner.

In summary, this paper presents a novel view on the label splitting technique. First, we show how label splitting for excitation closure is nothing else than coloring a graph using a minimal number (the *chromatic number* of the graph) of colors. Second, we characterize the conditions under which an optimal label splitting can be derived to accomplish excitation closure. Finally, we present an algorithm that can be used when excitation closure cannot be attained by a single application of the label splitting technique presented in this paper. This algorithm is based on a relaxation of the label splitting problem that can be mapped into the *weighted set cover* problem.

For the sake of clarity, the theory of this paper will be presented for the class of safe (1-bounded) Petri nets. The contribution can be extended with no substantial change for the class of general ( $k$ -bounded) Petri nets.

The organization of the paper is the following: in Section 2 we provide the reader the necessary background to understand the contents of this paper. Then in Section 3 and Section 4 the core contributions of the paper are presented,



describing techniques for obtaining regions and forcing (if possible) the synthesis conditions, respectively. In Section 5 a technique to iteratively apply the methods described in the previous sections and thus guaranteeing always a solution are described. Then, in Section 6 we provide applications and extensions of the presented technique.

## 2 Preliminaries

In this section we describe the basic elements necessary to understand the theory of this paper.

### 2.1 Finite Transition Systems and Petri Nets

**Definition 1 (Transition system).** A transition system is a tuple  $(S, E, A, s_{in})$ , where  $S$  is a set of states,  $E$  is an alphabet of events, such that  $S \cap E = \emptyset$ ,  $A \subseteq S \times E \times S$  is a set of (labeled) transitions, and  $s_{in} \in S$  is the initial state.

We use  $s \xrightarrow{e} s'$  as a shorthand for  $(s, e, s') \in A$ , and we write  $s \xrightarrow{*} s'$  if  $s = s'$  or there exist a path of labeled transitions  $(s_{i-1}, e_i, s_i) \in A$  where  $1 \leq i \leq n$ , for some  $n \geq 1$ ,  $s_0 = s$ ,  $s_n = s'$ . Let  $\text{TS} = (S, E, A, s_{in})$  be a transition system. We consider connected transition systems that satisfy the following axioms: i)  $S$  and  $E$  are finite sets, ii) every event has an occurrence:  $\forall e \in E \exists s, s' \in S : (s, e, s') \in A$ , and iii) every state is reachable from the initial state:  $\forall s \in S : s_{in} \xrightarrow{*} s$ .

In some parts of the paper it will be required to compare transitions systems. The following definition formalizes a well-known relation between transition systems.

**Definition 2 (Simulation, Bisimulation [1]).** Let  $\text{TS}_1 = (S_1, E, A_1, s_{in_1})$  and  $\text{TS}_2 = (S_2, E, A_2, s_{in_2})$  be two TSs with the same set of events. A simulation of  $\text{TS}_1$  by  $\text{TS}_2$  is a relation  $\pi$  between  $S_1$  and  $S_2$  such that  $s_1 \pi s_2$  and

- for every  $s_1 \in S_1$ , there exists  $s_2 \in S_2$  such that  $s_1 \pi s_2$ .
- for every  $(s_1, e, s'_1) \in A_1$  and for every  $s_2 \in S_2$  such that  $s_1 \pi s_2$ , there exists  $(s_2, e, s'_2) \in A_2$  such that  $s'_1 \pi s'_2$ .

When  $\text{TS}_1$  is simulated by  $\text{TS}_2$  with relation  $\pi$ , and vice versa with relation  $\pi^{-1}$ ,  $\text{TS}_1$  and  $\text{TS}_2$  are *bisimilar* [1].

**Definition 3 (Petri net [12, 14, 15]).** A Petri net is a tuple  $\text{PN} = (P, T, F, M_0)$  where  $P$  and  $T$  represent finite disjoint sets of places and transitions, respectively,  $F \subseteq (P \times T) \cup (T \times P)$  is the flow relation. Let markings be functions  $M : P \rightarrow \mathbb{N}$  assigning a number of tokens to each place. Marking  $M_0$  defines the initial state of the system.

For a node  $n$  (place or transition) of a Petri net,  $\bullet n = \{n' | (n', n) \in F\}$ , and  $n\bullet = \{n' | (n, n') \in F\}$ , are called the predecessor and successor set of  $n$  in  $F$ , respectively. A transition  $t \in T$  is *enabled* at marking  $M$  if  $\forall p \in \bullet t : M(p) \geq 1$ . The *firing* of an enabled transition  $t$  at  $M$  results in a marking  $M'$  (denoted  $M[t]M'$ ) such that for each place  $p$ :

$$M'(p) = \begin{cases} M(p) - 1 & \text{if } p \in \bullet t - t\bullet \\ M(p) + 1 & \text{if } p \in t\bullet - \bullet t \\ M(p) & \text{otherwise.} \end{cases}$$

A marking  $M'$  is *reachable* from  $M$  if there is a sequence of firings  $\sigma = t_1 t_2 \dots t_n$  that transforms  $M$  into  $M'$ , denoted by  $M[\sigma]M'$ . A sequence of transitions  $\sigma = t_1 t_2 \dots t_n$  is a *feasible sequence* if  $M_0[\sigma]M$ , for some  $M$ . The set of all markings reachable from the initial marking  $M_0$  is called its Reachability Set. The *Reachability Graph* of a Petri net PN (RG(PN)) is a transition system in which the set of states is the Reachability Set, the events are the transitions of the net and a transition  $(M_1, t, M_2)$  exists if and only if  $M_1[t]M_2$ . The initial state of the Reachability Graph is  $M_0$ . PN is  $k$ -bounded if  $M(p) \leq k$  for all places  $p$  and for all reachable markings  $M$ .

## 2.2 Regions and Synthesis

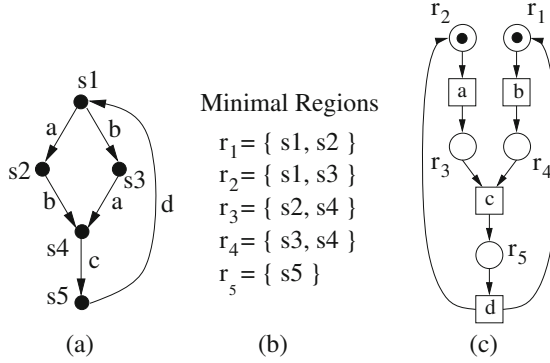
The theory of regions provides a path between transition systems and Petri nets. We now review this theory (the interested reader can refer to [5, 7, 9, 10, 13] for a complete overview). Given two states  $s, s' \in S$  and a subset of states  $S' \subseteq S$  of a transition system  $(S, E, A, s_{in})$ , if  $s \notin S'$  and  $s' \in S'$ , then we say that transition  $(s, e, s')$  *enters*  $S'$ . If  $s \in S'$  and  $s' \notin S'$ , then transition  $(s, e, s')$  *exits*  $S'$ . Otherwise, transition  $(s, e, s')$  *does not cross*  $S'$ .

**Definition 4.** Let  $\text{TS} = (S, E, A, s_{in})$  be a transition system. Let  $S' \subseteq S$  be a subset of states and  $e \in E$  be an event. The following conditions (in the form of predicates) are defined for  $S'$  and  $e$ :

$$\begin{aligned} \text{in}(e, S') &\equiv \exists (s, e, s') \in A : s, s' \in S' \\ \text{out}(e, S') &\equiv \exists (s, e, s') \in A : s, s' \notin S' \\ \text{nocross}(e, S') &\equiv \exists (s, e, s') \in A : s \in S' \Leftrightarrow s' \in S' \\ \text{enter}(e, S') &\equiv \exists (s, e, s') \in A : s \notin S' \wedge s' \in S' \\ \text{exit}(e, S') &\equiv \exists (s, e, s') \in A : s \in S' \wedge s' \notin S' \end{aligned}$$

Note that  $\text{nocross}(e, S') = \text{in}(e, S') \vee \text{out}(e, S')$ . We will abuse the notation and will use  $\text{nocross}(e, S', (s, e, s'))$  to denote a transition  $(s, e, s')$  that makes the predicate  $\text{nocross}(e, S')$  to hold, and the same for the rest of predicates.

The notion of a *region* is central for the synthesis of Petri nets. Intuitively, each region is a set of states and corresponds to a place in the synthesized Petri net.



**Fig. 1.** (a) transition system, (b) its minimal regions, (c) Petri net obtained applying Algorithm of Figure 2

**Definition 5 (Region).** A set of states  $r \subseteq S$  in the transition system  $\text{TS} = (S, E, A, s_{in})$  is called a region if the following two conditions are satisfied for each event  $e \in E$ :

- (i)  $\text{enter}(e, r) \Rightarrow \neg \text{nocross}(e, r) \wedge \neg \text{exit}(e, r)$
- (ii)  $\text{exit}(e, r) \Rightarrow \neg \text{nocross}(e, r) \wedge \neg \text{enter}(e, r)$

A region is a subset of states in which for *all* events, all transitions labeled with that event have exactly the same “entry/exit” relation. This relation will become the successor/predecessor relation in the Petri net. The event may always be either an *enter* event for the region (case (i) in the definition above), or always be an *exit* event (case (ii)), or never “cross” the region’s boundaries, i.e. each transition labeled with  $e$  is *internal* or *external* to the region, when the antecedents of neither (i) nor (ii) hold. The transition corresponding to the event will be predecessor, successor or unrelated with the corresponding place respectively. Examples of regions are shown in Figure 1 from the transition system of Figure 1(a), some regions are enumerated in Figure 1(b). For instance, for region  $r_2$ , event  $a$  is an exit event, event  $d$  is an entry event while the rest of events do not cross the region. Let  $r$  and  $r'$  be regions of a transition system. A region  $r'$  is said to be a *subregion* of  $r$  if  $r' \subset r$ . A region  $r$  is a *minimal* region if there is no other non-empty region  $r'$  which is a subregion of  $r$ . Going back to the example of Figure 1, the regions shown in Figure 1(b) are all minimal regions of the transition system of Figure 1(a). On the other hand, the region  $\{s_1, s_2, s_3, s_4\}$  is non-minimal. Each transition system  $\text{TS} = (S, E, A, s_{in})$  has two *trivial regions*: the set of all states,  $S$ , and the empty set. The set of non-trivial regions of  $\text{TS}$  will be denoted by  $R_{\text{TS}}$ .

A region  $r$  is a *pre-region* of event  $e$  if there is a transition labeled with  $e$  which exits  $r$ . A region  $r$  is a *post-region* of event  $e$  if there is a transition labeled with  $e$  which enters  $r$ . The sets of all pre-regions and post-regions of  $e$  are denoted  ${}^\circ e$  and  $e^\circ$ , respectively. By definition it follows that if  $r \in {}^\circ e$ ,

**Algorithm: Petri net synthesis**

- For each event  $e \in E$  generate a transition labeled with  $e$  in the Petri net;
- For each minimal region  $r \in R_{\text{TS}}$  generate a place  $\hat{r}$ ;
- Place  $\hat{r}$  contains a token in the initial marking iff the corresponding region  $r$  contains the initial state of  $\text{TS}$   $s_{in}$ ;
- The flow relation is as follows:  $e \in \hat{r} \bullet$  iff  $r$  is a pre-region of  $e$  and  $e \in \bullet \hat{r}$  iff  $r$  is a post-region of  $e$ , i.e.,

$$F_{\text{TS}} \stackrel{\text{def}}{=} \{(\hat{r}, e) \mid r \in R_{\text{TS}} \wedge e \in E \wedge r \in {}^\circ e\} \\ \cup \{(e, \hat{r}) \mid r \in R_{\text{TS}} \wedge e \in E \wedge r \in e^\circ\}$$

**Fig. 2.** Algorithm for Petri net synthesis from [13]

then all transitions labeled with  $e$  exit  $r$ . Similarly, if  $r \in e^\circ$ , then all transitions labeled with  $e$  enter  $r$ .

The procedure given by [13] to synthesize a Petri net,  $N_{\text{TS}} = (R_{\text{TS}}, E, F_{\text{TS}}, R_{s_{in}})$ , from a transition system  $\text{TS} = (S, E, A, s_{in})$  is illustrated in Figure 2. Notice that only minimal regions are required in the algorithm [9]. Depending on the class of transition systems considered, the algorithm provides different guarantees: for an *elementary transition system* [1], the algorithm derives a Petri net with behavior isomorphic to the initial transition system. For *excitation-closed transition systems* (see Definition 8 below), the algorithm derives a Petri net with behavior bisimilar to the initial transition system. For this latter class, the algorithm generates 1-bounded Petri nets without self-loops [7]. The generalization of the synthesis algorithm to  $k$ -bounded Petri nets can be found in [5]. An example of the application of the algorithm is shown in Figure 1. The initial transition system and the set of its minimal regions is given in Figures 1(a) and (b), respectively. The synthesized Petri net is shown in Figure 1(c).

The computation of the minimal regions is crucial for the synthesis methods in [5, 7]. It is based on the notion of *excitation region* [11].

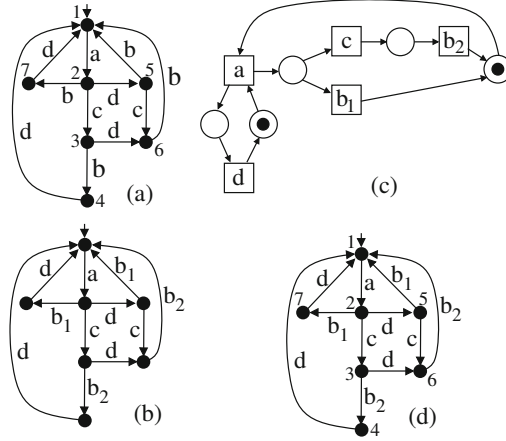
**Definition 6 (Excitation region).** *The excitation region of an event  $e$ ,  $\text{ER}(e)$ , is the set of states at which  $e$  is enabled, i.e.*

$$\text{ER}(e) = \{s \mid \exists s' : (s, e, s') \in A\}$$

In Figure 1(a), the set  $\text{ER}(c) = \{s_4\}$  is an example of an excitation region [2]. The set of minimal regions, needed in the synthesis algorithm of Figure 2, can be

<sup>1</sup> Elementary transition systems are a proper subclass of the transition systems considered in this paper, where additional conditions to the ones presented in Section 2.1 are required.

<sup>2</sup> Excitation regions are not regions in the terms of Definition 5. The term is used for historical reasons. For instance,  $\text{ER}(c)$  is not a region.



**Fig. 3.** (a) transition system (only numbers of states are shown) with minimal set of regions  $\{s1, s4, s7\}$ ,  $\{s1, s5, s6\}$ ,  $\{s2, s3, s4, s7\}$  and  $\{s2, s3, s5, s6\}$  (b) ECTS by label-splitting, (c) synthesized Petri net, (d) reachability graph of the Petri net

generated from the ERs of the events in a transition system in the following way: starting from the ER of each event, set expansion is performed on those events that violate the region condition. A pseudo-code of the expansion algorithm is given at the end of this subsection (as Algorithm [1](#)).

The following lemma from [7](#) characterizes the states to be added in the expansion of ERs:

**Lemma 1 (Set of states to become a region [7](#)).** *Let  $TS = (S, E, A, s_{in})$  be a transition system. Let  $r \subset S$  be a set of states such that  $r$  is not a region. Let  $r' \subseteq S$  be a region such that  $r \subset r'$ . Let  $e \in E$  be an event that violates some of the conditions for  $r$  to be a region. The following predicates hold for the sets  $r$  and  $r'$ :*

1.  $\text{in}(e, r) \wedge (\text{enter}(e, r) \vee \text{exit}(e, r)) \implies \{s \mid \exists s' \in r : (s, e, s') \in A \vee (s', e, s) \in A\} \subseteq r'$
2.  $\text{enter}(e, r) \wedge \text{exit}(e, r) \implies \{s \mid \exists s' \in r : (s, e, s') \in A \vee (s', e, s) \in A\} \subseteq r'$
3.  $\text{out}(e, r) \wedge \text{enter}(e, r) \implies (\{s \mid \exists s' \in r : (s, e, s') \in A\} \subseteq r') \vee (\{s \mid \exists s' \notin r : (s', e, s) \in A\} \subseteq r')$
4.  $\text{out}(e, r) \wedge \text{exit}(e, r) \implies (\{s \mid \exists s' \in r : (s', e, s) \in A\} \subseteq r') \vee (\{s \mid \exists s' \notin r : (s, e, s') \in A\} \subseteq r')$

In cases 1 and 2 above, the violating event  $e$  is converted into a **nocross** event, where only one way of expanding  $r$  is possible. However, in case 3 (4) there are two possibilities for expansion, depending on whether the violating event will be converted into a **nocross** or **enter** (**nocross** or **exit**) event.

Lemma [1](#) provides the basis for the derivation of the set of essential sets of states that will be the basis for the theory of this paper. Algorithm [1](#) presents

---

**Algorithm 1.** Expand\_States 7


---

**Input:**  $r$  is the set of states to be expanded,  
*explored* is the set of expansions already generated

**Output:**  $R$  collects all regions

```

1 begin
2   if  $r \in \text{explored}$  then return;           // avoid repeating computations
3   if  $r$  is a region then
4      $R = R - \{r_i | r \subset r_i\}$ ;           // remove supersets (non-minimal) of  $r$ 
5     if  $\neg \exists r_j \in R : r_j \subseteq r$  then  $R = R \cup \{r\}$ ;           //  $r$  added if minimal
6     return;
7   end
8   find  $e \in E$  violating some region condition in  $r$ ;
9    $r' = r \cup \{\text{set of states to legalize } e\}$ ;           // Lemma 1: all cases
10  Expand_States ( $r'$ ,explored, $R$ );           // Recursion
11  explored = explored  $\cup \{r'\}$ ;
12  if another expansion is needed then           // Case 3 or 4 in Lemma 1
13     $r' = r \cup \{\text{another set of states to legalize } e\}$ 
14    Expand_States ( $r'$ ,explored, $R$ );           // Recursion
15    explored = explored  $\cup \{r'\}$ 
16  end
17 end

```

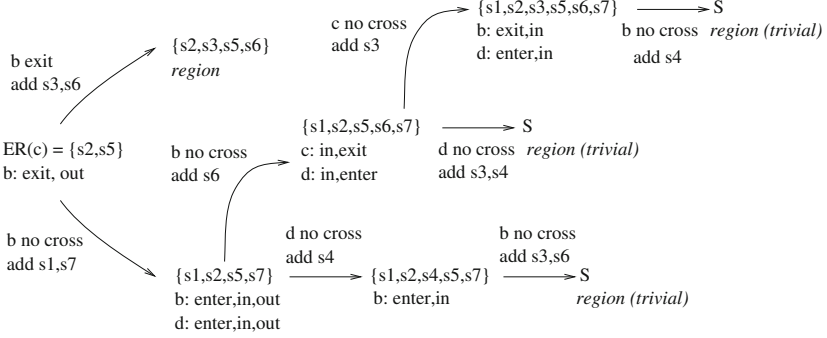
---

the pseudo-code for this procedure. The idea of the procedure is, starting from a given set of states  $r$  (in our setting it will be the excitation region of an event, as formalized below), if  $r$  is a region then update the region set found so far ( $R$ ) to delete superset regions of  $r$ , and insert  $r$  into  $R$  if it is minimal according to  $R$  (lines 11-12). If  $r$  is not a region, the algorithm expands  $r$  by adding states that are listed in some of the cases that hold from Lemma 1 (lines 13-15). For cases 3 or 4 of Lemma 1, the algorithm performs the other expansion needed, in lines 13-15. Since initially, the set of states to start with is not a region, at least one case will provide a set of states for the expansion. The new set obtained is recursively processed in the same manner, to derive a new set and so on. Moreover, since in cases 3 and 4 there are two possibilities for expansion, this recursive procedure branches in each option. Importantly, the procedure generates all the minimal pre-regions of an event 7.

In summary, set expansion to legalize violating events in a set of states generates a binary exploration tree, whose leafs are the regions found and internal nodes are non-regions. An example of such tree can be found in Figure 4. The following definition formalizes the notion of essential set:

**Definition 7 (Essential set of an event).** Let  $\text{TS} = (S, E, A, s_{in})$  be a transition system, and let  $e \in E$ .  $\text{Essential}(e, \text{TS})$  is the set of sets of states found by Algorithm 1, with initial set  $\text{ER}(e)$ . Formally:

- $\text{ER}(e) \in \text{Essential}(e, \text{TS})$ ,
- Let  $B \subseteq S$  with  $B \in \text{Essential}(e, \text{TS})$ . Then every set  $B'$  constructed from  $r = B$  as described in Lemma 1 is included in  $\text{Essential}(e, \text{TS})$ .



**Fig. 4.** Computation of essential sets ( $S = \{s1, s2, s3, s4, s5, s6, s7\}$ )

Notice that  $\text{Essential}(e, \text{TS}) \subseteq \mathcal{P}(S)$ , and  $\forall S' \in \text{Essential}(e, \text{TS}) : ER(e) \subseteq S'$ . For instance, in Figure 4 we show the computation of  $\text{Essential}(c, \text{TS})$  for the TS of Figure 3(a). Hence,  $\text{Essential}(c, \text{TS}) = \{\{s2, s5\}, \{s1, s2, s5, s7\}, \{s2, s3, s5, s6\}, \{s1, s2, s4, s5, s7\}, \{s1, s2, s5, s6, s7\}, \{s1, s2, s3, s5, s6, s7\}, \{s1, s2, s3, s4, s5, s6, s7\}\}$ . For instance, the three sets of states  $\{s1, s2, s5, s7\}$ ,  $\{s1, s2, s5, s6, s7\}$  and  $\{s2, s3, s5, s6\}$  in  $\text{Essential}(c, \text{TS})$  are computed according to case 4 in Lemma 11 on event  $b$ , while the set  $\{s1, s2, s4, s5, s7\}$  is computed from set  $\{s1, s2, s5, s7\}$  to deal with the violation caused by event  $d$  (case 3 in Lemma 11). Notice that the essential sets found are partitioned into regions and non-regions. Regions are in turn partitioned into minimal and non-minimal regions. In the example above, the sets  $\{s2, s5\}$ ,  $\{s1, s2, s5, s7\}$ ,  $\{s1, s2, s4, s5, s7\}$  and  $\{s1, s2, s3, s5, s6, s7\}$  are essential sets that are not regions, whereas the sets  $\{s2, s3, s5, s6\}$ , and  $\{s1, s2, s3, s4, s5, s6, s7\}$  (the leaves of the tree shown in Figure 4) are essential sets that are regions, being the last one the trivial region.

### 2.3 Excitation-Closed Transition Systems

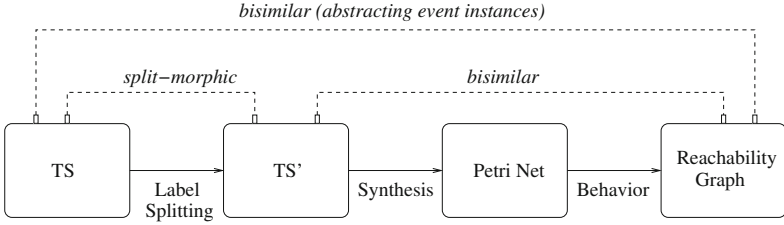
In this section we define formally the class of transition systems that will be considered in this paper.

**Definition 8 (Excitation-closed transition systems).** *A transition system  $\text{TS} = (S, E, A, s_{in})$  is an excitation-closed transition system (ECTS) if it satisfies the following two axioms:*

- *Excitation closure (EC): For each event  $e$ :  $\bigcap_{r \in \circ_e} r = ER(e)$*
- *Event effectiveness (EE): For each event  $e$ :  $\circ_e \neq \emptyset$*

ECTSs include the class of elementary transition systems [7]. Interestingly, the excitation closure axiom (EC) for ECTSs is equivalent to the *forward closure* condition [10, 13] required for elementary transition systems.

<sup>3</sup>  $\mathcal{P}(S)$  denotes the set of all subsets of  $S$ .



**Fig. 5.** Relationship between the different objects if label splitting is applied

The synthesis algorithm in Figure 2 applied to an ECTS produces a Petri net with reachability graph *bisimilar* to the initial transition system 7. When the transition system is not excitation-closed, then it must be transformed to enforce that property. One possible strategy is to represent every event by multiple transitions labeled with different instances of the same label. This technique is called *label splitting*. Figure 3 illustrates the technique. The initial transition system, shown in Figure 3(a), is not an ECTS: the event  $c$  does not satisfy the EE axiom nor the EC axiom, since no pre-regions of  $c$  exist. In order for the EC/EE axioms on  $c$  to be satisfied, one can force the set of states  $\{s2, s5\}$  (the excitation region of event  $c$ ) to be a region by applying label splitting. The set  $\{s2, s5\}$  is not a region because event  $b$  violates the region condition (some  $b$ -transitions **exit** and some others **nocross**). Hence, following this partition on the  $b$ -transitions with respect to  $\{s2, s5\}$ , the transition system is transformed by splitting the event  $b$  into the events  $b_1$  and  $b_2$ , as shown in Figure 3(b), resulting in an ECTS. The synthesized Petri net, with two transitions for event  $b$  is shown in Figure 3(c). Transition system of Figure 3(b) is *split-morphic* 7 to the transition system of Figure 3(a): there exists a surjective mapping between the sets of events, where different instances  $(a_1, a_2, \dots)$  of the single event  $a$  are mapped to  $a$ . The reachability graph of the Petri net of Figure 3(c), shown in Figure 3(d), is bisimilar to transition system of Figure 3(b). Moreover, if we abstract away the label indexes in the reachability graph of Figure 3(d), the equivalence relation between the transition system of Figure 3(a) and the reachability graph is bisimilarity: the relation  $\pi = (si, si)$ , for  $1 \leq i \leq 7$  is a bisimulation if event instances are abstracted. Figure 5 shows the relationships between the original transition system, the transformed one obtained through label splitting, and the reachability graph of the synthesized Petri net.

Hence in Petri net synthesis label splitting might be crucial for the existence of a Petri net with bisimilar behavior. The following definition describes the general application of label splitting:

**Definition 9 (Label splitting).** Let  $TS = (S, E, A, s_{in})$  be a transition system. The splitting of event  $e \in E$  produces a transition system  $TS' = (S, E', A', s_{in})$ ,



with  $E' = E - \{e\} \cup \{e_1, \dots, e_n\}$ , and such that every transition  $(s_1, e, s_2) \in A$  corresponds to exactly one transition  $(s_1, e_i, s_2)$ , and the rest of transitions for events different from  $e$  in  $A$  are preserved in  $A'$ .

Label splitting is a powerful transformation which always guarantees excitation closure: any TS can be converted into one where every transition has a different label. By definition, the obtained TS is ECTS but the size of the derived Petri net is equal to the size of the obtained ECTS. In this paper we aim at reducing the number of labels, thus reducing the size of the Petri net derived.

The work presented in this paper considers a particular application of the label splitting technique which is based on converting a set of states into a region, described in the next section.

### 3 Optimal Label Splitting to Obtain a Region

In this section the following problem is addressed: given a transition system  $\text{TS} = (S, E, A, s_{in})$  and a set of states  $S' \subseteq S$  which is not a region, determine the minimal number of label splittings to be applied in order that  $S'$  becomes a region. This is a crucial step for the technique presented in the following section to satisfy the ECTS property for a transition system. The main contribution of this section is to show that the problem might be reduced to computing the *chromatic number* of a graph [17].

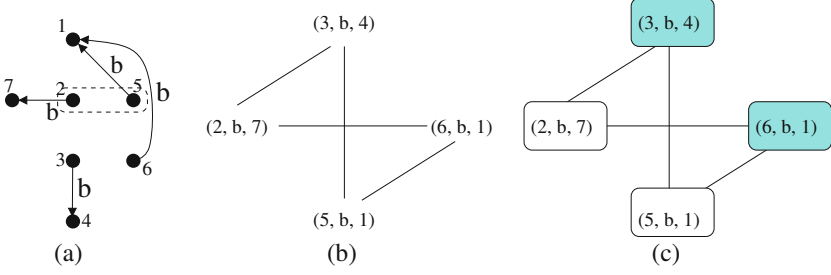
First, let us introduce the concept of *gradient graph*:

**Definition 10 (Gradient Graph).** *Given a transition system  $\text{TS} = (S, E, A, s_{in})$ , a set  $S' \subseteq S$  and an event  $e \in E$ , the gradient graph of  $e$  with respect to  $S'$  in TS, denoted as  $\mathcal{GG}(e, S') = (A_e, M)$  is an undirected graph defined as:*

- $A_e = \{(s, x, s') \mid (s, x, s') \in A \wedge x = e\}$ , is the set of nodes, and
- $M = \{\{v, v'\} \mid v, v' \in A_e \wedge [(\text{enter}(e, S', v) \wedge (\text{nocross}(e, S', v') \vee \text{exit}(e, S', v')))) \vee (\text{exit}(e, S', v) \wedge (\text{nocross}(e, S', v') \vee \text{enter}(e, S', v')))]\}$  is the set of edges.

Informally, the gradient graph contains as nodes the transitions of an event  $e$ , and an edge exists between two nodes which satisfy different predicates on set  $S'$ , like one transition **enter**  $S'$  and the other does not, or one transition **exit**  $S'$  and the other does not. For instance, the gradient graph on event  $b$  and set of states  $S' = \{s_2, s_5\}$  in the transition system of Figure 3(a) is shown in Figure 6(b) (for the sake of clarity we show in Figure 6(a) only the transitions on event  $b$  from Figure 3(a)).

A graph  $G = (V, E)$  is *k-colourable* if there exists an assignment  $\alpha : V \rightarrow \{1, 2, \dots, k\}$  for which any pair of nodes  $v, v' \in V$  such that  $\{v, v'\} \in E$  satisfy  $\alpha(v) \neq \alpha(v')$ . The *chromatic number*,  $\chi(G)$ , of a graph  $G$  is the minimum  $k$  for which  $G$  is *k-colourable* [17]. The rest of the section shows the relation between the chromatic number and the optimal label splitting to obtain a region.



**Fig. 6.** (a) Projection of the transition system of Figure 3(a) showing only the transitions involving event  $b$  with the encircled set of states  $\{s_2, s_5\}$ , (b)  $\mathcal{GG}(b, \{s_2, s_5\})$ , (c) coloring

**Definition 11 (Label splitting as gradient graph coloring).** *Given a transition system  $\text{TS} = (S, E, A, s_{in})$ , the label splitting of event  $e$  according to a coloring  $\alpha$  of the gradient graph  $\mathcal{GG}(e, S')$  produces the transition system  $\text{TS}' = (S, E', A', s_{in})$  where  $E' = E - \{e\} \cup \{e_1, \dots, e_n\}$ , with  $\{e_1, \dots, e_n\}$  being the colors defined by the coloring of  $\mathcal{GG}(e, S')$ . Every transition  $(s, e, s')$  of event  $e$  is transformed into  $(s, e_{\alpha((s, e, s'))}, s')$ , whilst the rest of transitions of events in  $E - \{e\}$  are preserved in  $A'$ .*

For instance, the label splitting of the transition system of Figure 3(a) according to the coloring shown on Figure 6(c) of  $\mathcal{GG}(b, \{s_2, s_5\})$  is shown on Figure 3(b).

**Proposition 1.** *Given a transition system  $\text{TS} = (S, E, A, s_{in})$  and the gradient graph  $\mathcal{GG}(e, S')$ . If event  $e$  is split in accordance with a  $\chi(\mathcal{GG}(e, S'))$ -coloring of  $\mathcal{GG}(e, S')$  then the new inserted events  $\{e_1, \dots, e_{\chi(\mathcal{GG}(e, S'))}\}$  satisfy the region conditions for  $S'$  (cf., Definition 5)).*

*Proof.* By contradiction: assume that there exists  $e_i \in \{e_1, \dots, e_{\chi(\mathcal{GG}(e, S'))}\}$  such that conditions of Definition 5 do not hold. Without loss of generality, we assume that there exist  $(s_1, e_i, s_2)$  and  $(s'_1, e_i, s'_2)$  for which predicates  $\text{enter}(e_i, S', (s_1, e_i, s_2))$  and  $\text{nocross}(e_i, S', (s'_1, e_i, s'_2))$  hold (the other cases can be proven similarly). But then the nodes  $(s_1, e, s_2)$  and  $(s'_1, e, s'_2)$  are connected by an edge in  $\mathcal{GG}(e, S')$ , but they are assigned the same color  $e_i$ . This is a contradiction.  $\square$

Notice that, when the graph  $\mathcal{GG}(e, S')$  contains no arcs it means that for the set  $S'$  all the  $e$ -transitions satisfy the region predicates of Definition 5. This graph can be trivially colored with one color, i.e.,  $\chi(\mathcal{GG}(e, S')) = 1$ . Hence, in that case, according to Definition 11, no increase in the number of labels arises due to the event splitting, e.g.,  $|E'| = |E - \{e\} \cup \{e_1\}| = |E| - 1 + 1 = |E|$ , with  $E, E'$  being the alphabets of Definition 11. In this case the transformation is denoted *renaming*.

**Definition 12 (Renaming).** *If  $\chi(\mathcal{GG}(e, S')) = 1$  then the application of label splitting transformation on event  $e$  for  $S'$  is called renaming.*

Clearly, renaming an event  $e$  is a cosmetic transformation which does not change any of the predicates of Definition 5 for the label of the new event on the set of states considered.

The following corollary provides the first main result of this section that relates coloring of the gradient graph to obtaining regions in a transition system.

**Corollary 1.** *Given a transition system  $\text{TS} = (S, E, A, s_{in})$  and a set  $S' \subseteq S$ . If every event  $e$  is split according to the colors required for achieving  $\chi(\mathcal{GG}(e, S'))$ -coloring, then  $S'$  is a region in the resulting transition system.*

*Proof.* It follows from iterative application of Proposition 11. □

In the following theorem we abuse the notation and extend the  $\chi$  operator to sets of states, and define a lower bound to the number of label splittings:

**Theorem 1.** *Given a transition system  $\text{TS} = (S, E, A, s_{in})$ ,  $E = \{e_1, \dots, e_n\}$  and a set  $S' \subseteq S$ . Denote  $\chi(S') = \chi(\mathcal{GG}(e_1, S')) + \dots + \chi(\mathcal{GG}(e_n, S'))$ . To make  $S'$  a region, the minimum number of labels needed including renaming is  $\chi(S')$ , and the minimum number of labels needed excluding renaming is  $\chi(S') - n$ .*

*Proof.* By contradiction: if there is a  $k < \chi(S')$  such that only  $k$  labels are needed to convert  $S'$  into region, then there is an event  $e \in E$  for which less than  $\chi(\mathcal{GG}(e, S'))$  labels are used to split  $e$  for satisfying the conditions of Definition 5. This leads to a contradiction to the chromatic number of the graph  $\mathcal{GG}(e, S')$ . Finally, notice that formally the application of the transformation in Definition 11 removes all the original events (some of them are simply renamed into a new event). Hence it follows that the minimum number of labels excluding renaming is  $\chi(S') - n$ . □

Hence Theorem 11 establishes that at least  $\chi(S') - n$  extra labels are needed in order for  $S'$  to become a region. Clearly, if apart from the splittings derived from  $\chi(S')$ , other splittings are additionally done,  $S'$  will still be a region. The theory presented in this section represents the core idea for the label splitting technique of this paper. The next section shows how to apply it to obtain ECTSs.

## 4 Optimal Label Splitting on Essential Sets for Synthesis

Given a non-ECTS, the following question arises: is there an algorithm to transform it into an ECTS with a minimal number of extra labels? This section addresses this problem, deriving sufficient conditions under which a positive answer can be given. As was done in previous work [7], in this paper we will restrict the theory to a particular application of the label splitting: instead of an arbitrary instantiation of Definition 9 which may split an event of a transition system in an

arbitrary way, we will only consider the splittings used to convert essential sets into regions (Definition 11), a technique which has been shown in the previous section.

We tackle the problem in two phases: first, we show how the ECTS conditions for an event  $e$  (Definition 8) can be achieved by using the essential sets found in the expansion (see Lemma 11) of the excitation region  $\text{ER}(e)$ , described in Definition 7. Then we show the conditions under which the strategy can be applied in the general case, i.e., considering all the events not satisfying some of the conditions required in Definition 8.

For the definitions and theorems in this section, we assume in the formalizations a transition system  $\text{TS} = (S, E, A, s_{in})$ . We use  $\text{Witness}(e, \text{TS})$  to denote the sets of essential sets of an event  $e$  that are not regions and, if they are converted into regions, both the EC and EE axioms from Definition 8 on  $e$  will hold. Formally:

**Definition 13 (Witness sets of an event).** *Let  $e \in E$ .  $\text{Witness}(e, \text{TS})$  is defined as follows:*

$$\begin{aligned} \mathcal{C} = \{S_1, \dots, S_k\} \in \text{Witness}(e, \text{TS}) \iff & (\bigcap_{q \in (\circ_e \cup \mathcal{C})} q) = \text{ER}(e) \wedge (\forall S_i \in \mathcal{C} : \\ & S_i \in \text{Essential}(e, \text{TS}) \wedge S_i \notin R_{\text{TS}} \\ & \wedge \neg \text{nocross}(e, S_i) \wedge \neg \text{enter}(e, S_i)) \end{aligned}$$

A witness set contains non-regions  $S_i$  that are candidates to be pre-regions of the event (since both  $\neg \text{nocross}(e, S_i)$  and  $\neg \text{enter}(e, S_i)$  hold, and therefore the only possibility for  $e$  in  $S_i$  is to exit). The intersection of the sets forming a witness, together with the existing pre-regions of the event, is the excitation region of the event. Hence, if event  $e$  does not satisfy the EC or EE conditions from Definition 8, the singleton  $\{\text{ER}(e)\}$  will always be in  $\text{Witness}(e, \text{TS})$ <sup>4</sup>. For instance, for the TS of Figure 3(a), a witness for event  $c$  is  $\{\{s2, s5\}\}$ . Notice that if an event satisfies both the EC and EE axioms from Definition 8, then its witness set is empty.

Finally, if  $\mathcal{C} = \{S_1, \dots, S_k\}$ , we abuse the notation and use  $\chi(\mathcal{C})$  to denote  $\chi(\mathcal{GG}(e_1, S_1) \cup \dots \cup \mathcal{GG}(e_1, S_k)) + \dots + \chi(\mathcal{GG}(e_n, S_1) \cup \dots \cup \mathcal{GG}(e_n, S_k))$ , with  $E = \{e_1, \dots, e_n\}$ . The union operator on gradient graphs is defined as  $\mathcal{GG}(e, S_1) \cup \dots \cup \mathcal{GG}(e, S_k) = (A_e, M_1 \cup \dots \cup M_k)$ , with  $A_e$  and  $M_i$  being the nodes and edges of the graph  $\mathcal{GG}(e, S_i)$ , for  $i = 1, \dots, k$  cf., Definition 10<sup>5</sup>. For the example of Figure 3(a), we have  $\chi(\{\{s2, s5\}\}) = \chi(\mathcal{GG}(a, \{s2, s5\})) + \chi(\mathcal{GG}(b, \{s2, s5\})) + \chi(\mathcal{GG}(c, \{s2, s5\})) + \chi(\mathcal{GG}(d, \{s2, s5\})) = 1 + 2 + 1 + 1 = 5$ .

First, we start by describing the minimal strategy to make an event satisfy the conditions of Definition 8.

<sup>4</sup> There is a special case where some states in  $\text{ER}(e)$  are connected through  $e$ -transitions, which can then be considered in a generalization of Definition 13. For the sake of readability, we use this simple version of witness.

<sup>5</sup> We only consider the union of gradient graphs of the same event.

**Proposition 2.** *Let  $\mathcal{C} = \{S_1, \dots, S_k\} \in \text{Witness}(e, \text{TS})$  such that  $\chi(\mathcal{C})$  is minimal, i.e.,  $\forall \mathcal{C}' \in \text{Witness}(e, \text{TS}) : \chi(\mathcal{C}) \leq \chi(\mathcal{C}')$ . Then, if only label splitting on essential sets (Definition 17) is considered,  $\chi(\mathcal{C})$  is the minimal number of labels needed to make the EC and EE axioms of Definition 8 on  $e$  to hold.*

*Proof.* First, it is clear that after applying label splitting on essential sets from a non-empty witness, the set of pre-regions of  $e$  becomes non-empty. We now prove the minimality of  $\mathcal{C}$  by contradiction. Assume the EC/EE axioms can hold by the sole application of label splitting on essential sets and with less labels than  $\chi(\mathcal{C})$ , where  $\chi(\mathcal{C})$  is minimal. Then this means that there is a set  $\mathcal{C}'$  that can be used instead of  $\mathcal{C}$  to make the EC and EE axioms to hold, and which requires fewer label splittings than  $\chi(\mathcal{C})$ . Since  $\mathcal{C}' \notin \text{Witness}(e, \text{TS})$  (otherwise  $\mathcal{C}'$  will be the minimal witness instead of  $\mathcal{C}$ ), then at least one set  $S'_i \in \mathcal{C}'$  satisfies that  $S'_i \notin \text{Essential}(e, \text{TS})$  or  $S'_i \in R_{\text{TS}}$ , since for the rest of witness conditions that can be violated (intersection not yielding  $\text{ER}(e)$  or  $e$  entering/crossing  $S'_i$ ,  $S'_i$  cannot be used to make the EC and EE axioms to hold). In any of the two situations, we reach a contradiction: if  $S'_i \notin \text{Essential}(e, \text{TS})$ , then the label splitting is not applied on essential sets, and if  $S'_i \in R_{\text{TS}}$ , then  $\mathcal{C}' \setminus \{S'_i\} \in \text{Witness}(e, \text{TS})$ , and clearly  $\chi(\mathcal{C}) > \chi(\mathcal{C}' \setminus \{S'_i\})$ .  $\square$

Given a non-ECTS, the optimal label splitting problem on essential sets is to determine which sets to convert into regions in order to satisfy, for each event, the EE and EC axioms, using the minimal number of labels.

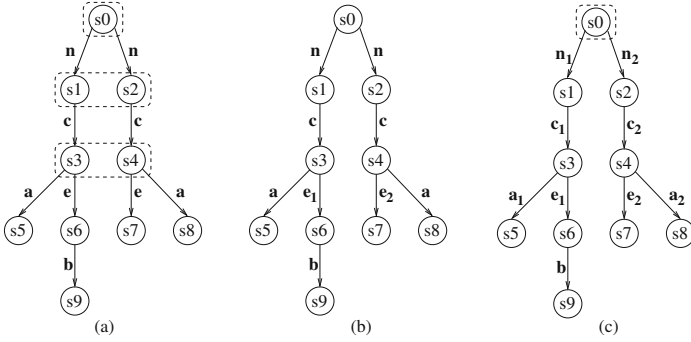
**Definition 14 (Optimal label splitting on essential sets).** *Given the events  $e_1, \dots, e_k \in E$  violating the EC or EE axioms from Definition 8, define the universe  $\mathcal{U}$  as  $\text{Witness}(e_1, \text{TS}) \cup \dots \cup \text{Witness}(e_k, \text{TS})$ . The optimal label splitting problem is to determine sets  $S_1, \dots, S_n \in \mathcal{U}$  such that  $\forall 1 \leq i \leq k : \exists \mathcal{C} \in \text{Witness}(e_i, \text{TS}) : \mathcal{C} \subseteq \{S_1, \dots, S_n\}$  and where  $\chi(\{S_1, \dots, S_n\})$  is minimal.*

Notice that Definition 14 is defined on the set of events violating the EC/EE condition, selecting a set of essential sets which both requires minimal number of labels and ensures the EC/EE axioms to hold for these events. An interesting result guarantees EC/EE preservation for those events that both satisfy initially the EC/EE axioms and were not split:

**Proposition 3.** *Given an event  $e \in E$  such that the EC/EE axioms from Definition 8 hold, and  $e$  has not been selected for splitting. Then these axioms hold in the new transition system obtained after label splitting.*

*Proof.* Label splitting preserves regions: the predicates of Definition 5 that hold on each region will also hold if some event is split. Hence, the non-empty set of regions ensuring  $\bigcap_{r \in \circ_e} r = \text{ER}(e)$  and  $\circ_e \neq \emptyset$  is still valid after label splitting.  $\square$

However, label splitting may split events for which the EC/EE axioms were satisfied or attained. Unfortunately, the new events appearing might not satisfy these axioms, as the following example demonstrates.



**Fig. 7.** (a) Initial transition system where minimal regions are drawn within dashed lines: all the events but  $b$  satisfy the EC axiom, (b) the splitting of  $e$  leads to new events  $e_1$  and  $e_2$  not satisfying the EC axiom, (c) the final ECTS, where all events have been split

*Example 1.* In the transition system of Figure 7(a) events  $n$ ,  $c$ ,  $a$  and  $e$  satisfy both the EC and EE axioms, since all these events have non-empty set of pre-regions whose intersection is the excitation region of the event. However,  $b$  does not satisfy neither the EC nor the EE axioms, and the splitting required for  $b$  to satisfy these axioms is done on the essential set  $\{s6\}$ , which requires to split the event  $e$ , resulting in the transition system of Figure 7(b). The new events  $e_1$  and  $e_2$  arising from the splitting of  $e$  do not satisfy the EC axiom (but do satisfy the EE axiom). This requires further splittings, which as in the case of  $b$ , force the splitting of events resulting in new events that do not satisfy the EC axiom. Four iterations are required to obtain the ECTS shown in Figure 7(c).

This example invalidates any label splitting strategy that aims at reaching excitation closure in just one iteration by only inspecting the violations to the EC/EE axioms: in general, when the splitting of some event is applied, its ER is divided into several ERs for which there might be no set of pre-regions which guarantees the satisfaction of the EC/EE axioms on these new events arising. Importantly, the label splitting technique preserves the regions, but new regions might be necessary for the new events arising from a splitting. Therefore, any label splitting technique that focuses on EC/EE violations must be an iterative method (see next section for such a method). However, if the new labels inserted do not incur violations to the EC/EE axioms, the presented technique guarantees the optimal label splitting:

**Theorem 2.** *Let  $TS' = (S, E', A', s_{in})$  be the transition system reached after splitting labels on events  $e_1, \dots, e_k$  that violate the EC/EE axioms of Definition 8 in transition system  $TS$ , using the minimal combination of witness  $S_1, \dots, S_n$  from Definition 14. Then, if the new events appearing satisfy the EC/EE axioms, the number of splittings performed is minimal and  $TS'$  is ECTS.*

*Proof.* The minimality of the witness according to Definition 14 ensures that no fewer splittings are possible to make  $e_1, \dots, e_k$  to satisfy the EC/EE axioms: if less splittings were possible by a different set of essential sets, then the essential sets chosen by Definition 14 will be that set. Moreover, the assumption implies that the new events arising from the splitting satisfy the EC/EE axioms. Finally, Proposition 3 guarantees that events that were not split still satisfy the EC/EE axioms. The set of events  $E'$  is partitioned into these two sets, and therefore  $TS'$  is ECTS.  $\square$

Notice that the current characterization of the label splitting problem, as seen from Theorem 2, is dynamic, i.e., the new labels appearing when problematic labels are split must satisfy the axioms required in an ECTS in order to guarantee a minimal number of labels. In contrast, a static characterization will simply impose constraints on the initial transition system in order to determine the minimal number of labels required for reaching an ECTS. Static techniques for solving the label splitting problem may represent an alternative to the techniques described in this paper.

## 5 A Greedy Algorithm for Iterative Label Splitting

The optimal label splitting problem presented in the previous section considers the minimal application of the label splitting technique to satisfy the EE/EC conditions for the initial set of events. Moreover, it was shown that in general it may not be possible to reach an ECTS by a single application of the technique.

The label splitting problem is similar to the *weighted set cover* (WSC) problem [6]. The WSC problem can be informally described as follows: given a finite set  $Y$  and a family  $\mathcal{F}$  of subsets of  $Y$ , such that every element of  $Y$  belongs to at least one subset in  $\mathcal{F}$ , and each set  $S_i$  in  $\mathcal{F}$  has an associated weight  $w_i$ , determine a minimum-weight cover, i.e. a set  $\mathcal{C} \subseteq \mathcal{F}$  that contains all the elements in  $Y$  and the sum of the weights of the elements in  $\mathcal{C}$  is minimal. The WSC problem is known to be NP-hard, and can be formulated as an *integer linear programming* (ILP) model [6]. This section is devoted to show how to encode in an ILP model the label splitting problem, inspired by the corresponding ILP formulation for the WSC problem. Moreover, an algorithm for label splitting that iterates until excitation closure is achieved is presented.

Clearly, the label splitting problem is akin to the WSC problem: if  $e_1, \dots, e_k$  are the events that do not satisfy the EC or EE axioms, then  $\mathcal{F} = \text{Witness}(e_1, \text{TS}) \cup \dots \cup \text{Witness}(e_k, \text{TS})$  is the family of subsets to consider, and the problem is to find a set of sets of states  $\{S_1, \dots, S_n\}$  that i) at least one witness is covered for each event  $e_i$ ,  $1 \leq i \leq k$ , and ii) requires the minimal number of labels.

The ILP model for the WSC problem simply minimizes the sum of individual costs of each element  $S_i$  while covering the set  $Y$ . In our setting, we would like to minimize the number of labels, and hence will use the function  $\chi$  in the cost function of the ILP model. Then, based on the ILP model for the WSC problem, the resulting ILP model for the label splitting problem is:

$$\begin{aligned}
& \min \sum_{\mathcal{C} \in \mathcal{W}} \chi(\mathcal{C}) \cdot X_{\mathcal{C}} & (1) \\
& \text{subject to} \\
& \forall e \in \{e_1, \dots, e_k\} : \sum_{\mathcal{C} \in \text{Witness}(e, \text{TS})} X_{\mathcal{C}} \geq 1 \\
& X_{\mathcal{C}} \in \{0, 1\}
\end{aligned}$$

where  $e_1, \dots, e_k, \in E$  are the set of events that do not satisfy the EC/EE axioms,  $\mathcal{W} = \text{Witness}(e_1, \text{TS}) \cup \dots \cup \text{Witness}(e_k, \text{TS})$  and  $X_{\mathcal{C}}$  denotes the binary variable that selects witness  $\mathcal{C}$  to be or not in the solution. A solution to the ILP model (II) will then minimize the sum of labels needed to convert the witnesses selected for each event violating the EE/EC axioms into regions<sup>6</sup>.

Imagine that a solution to model (II) contains a pair of witnesses  $\mathcal{C}_1 = \{S_1, S_2, S_3\}$  and  $\mathcal{C}_2 = \{S_2, S_3, S_4\}$ . Unfortunately, the equality

$$\chi(\mathcal{C}_1 \cup \mathcal{C}_2) = \chi(\mathcal{C}_1) + \chi(\mathcal{C}_2) \quad (2)$$

does not hold since the fact that  $\mathcal{C}_1$  and  $\mathcal{C}_2$  share some set (e.g., share sets  $S_2$  and  $S_3$ ) implies that at the right hand side of the equality colors are counted in every gradient graph of an event, while in the left hand side only one gradient graph (which is the union of these gradient graphs) is considered. In that situation one may be counting twice the labels (colors) needed in the right part of the equality of (2). Notice that ILP model (II) minimizes the sum of labels of individual witnesses (as done in the right part of equality (2)), and hence it may provide non-optimal solutions to the label splitting problem. In spite of this, by using an ILP solver to find an optimal solution for model (II), one may have a strategy to proceed into deriving an ECTS with reasonable (although maybe not-optimal) number of label splittings. Hence, our proposed strategy based on the ILP model (II) is greedy.

Algorithm 2 presents the iterative strategy to derive an ECTS. It first computes the minimal regions of the initial transition system. The algorithm iterates when the EE/EC axioms do not hold for at least one event (function `ExcitationClosed(TS', R)` will return false). Then the main loop of the technique starts by collecting the witnesses for events violating the EE/EC axioms of the current transition system, which are provided in the set  $\mathcal{W}$  (line 5). Then model (II) based on  $\mathcal{W}$  is created and an ILP solver is invoked in line 6 that will provide a solution (a set of witnesses that ensure the satisfaction of the EE/EC axioms for the problematic events). Notice that for the sake of clarity of the algorithm, we provide in line 6 the sets that form the cover instead of providing the particular witness selected for each event (i.e., given a solution  $\mathcal{C}_1, \dots, \mathcal{C}_k$  of model (II),  $\{S_1, \dots, S_n\} = \bigcup_{1 \leq i \leq k} \mathcal{C}_i$ ). In line 7 the splitting of labels corresponding to  $\chi(\{S_1, \dots, S_n\})$  is performed, ensuring that sets  $S_1, \dots, S_n$  become regions. The new regions are appended to the regions found so far (which are still

<sup>6</sup> In model (II) it is important to use the  $\geq$  in the constraints part since in an optimal solution it may happen that an event is covered by more than one witness.



**Algorithm 2.** GreedyIterativeSplittingAlgorithm

---

**Input:** Transition system  $\text{TS} = (S, E, A, s_{in})$   
**Output:** Excitation-closed transition system  $\text{TS}' = (S, E', A', s_{in})$  bisimilar to  $\text{TS}$

```

1 begin
2    $\text{TS}' = \text{TS}$ 
3    $\mathcal{R} = \text{GenerateMinimalRegions}(\text{TS})$ 
4   while not (ExcitationClosed( $\text{TS}'$ ,  $\mathcal{R}$ )) do
5      $\mathcal{W} = \text{CollectWitnesses}(\text{TS}')$ 
6      $(S_1, \dots, S_n) = \text{Solution ILP model (II)}$ 
7      $\text{TS}' = \text{SplitLabels}(\text{TS}', S_1, \dots, S_n)$ 
8      $\mathcal{R} = \mathcal{R} \cup \{S_1, \dots, S_n\}$ 
9   end
10 end

```

---

regions, see proof of Proposition 3), and the excitation closure is re-evaluated to check convergence.

Although in general the presented iterative technique is not guaranteed to provide an ECTS that has the minimal number of labels, in terms of convergence the improvements with respect to the previous (also non-optimal) approaches [5, 7] are:

- the whole set of events violating the EE/EC axioms are considered in every iteration: in previous work only one event is considered at a time, and
- for every event violating the EE/EC axioms, the necessary splittings are applied to attain excitation closure: on the previous work, only one of the splittings was applied.

Hence the *macro* technique presented in this section is meant to speed-up the achievement of the excitation closure, when compared to the *micro* techniques presented in the literature.

## 6 Extensions and Applications of Label Splitting

The theory presented in this paper may be extended into several directions. Here we informally enumerate some possible extensions.

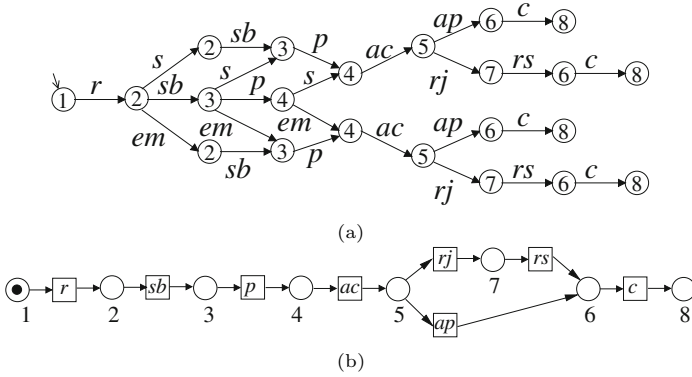
**Extension to  $k$ -Bounded Synthesis.** For the sake of clarity, we have restricted the theory to safe Petri nets. The extension to  $k$ -bounded Petri nets can be done by adapting the notion of gradient graph to use *gradients* instead of the predicates required in Definition 5, and lift the notion of region to multisets instead of sets. Informally, the gradient of an event is an integer value which represents the modification made by the event on the multiplicity of the region (see [2, 5] for the formal definition of gradient). When more than one gradient is assigned to a given event in a multiset, then the multiset is not a region. Given a *multiset*  $r$  which is not a region, the essential sets will be those multisets  $r' \geq r$

such that the number of gradients for some event has decreased, i.e.,  $r'$  is a sound step towards deriving a region from multiset  $r$ . The excitation closure definition (see Definition 8) for the general case should be the one described in 5: a multiset which is a region will be a pre-region of an event if its *support* (set of states with multiplicity different than 0) includes the excitation region of the event. The support sets of pre-regions will be intersected and for the event to satisfy the EC/EE axioms this intersection should be equal to the excitation region of the event. The concept of the witness set of an event can be naturally adapted to use multisets instead of sets by using the support of each multiset accordingly.

**Label Splitting for Petri Net Classes.** Maybe one of the strongest points of the theory of regions is the capability to guide the synthesis for particular Petri net classes 4,7. The core idea is to select, among the set of all regions, those that satisfy particular conditions. Here we list some interesting classes that may be obtained by restricting the label splitting technique presented in this paper.

- In a *marked graph* 12, every region (place) must have exactly one event satisfying the **enter** predicate (input transition) and one satisfying the **exit** predicate (output transition), and the rest of events should be **nocross**. In general, when the goal is a particular Petri net class defined by conditions on the structure with respect to the connections to the places, one can simply select the essential sets that, if transformed into regions by using the theory of Section 3, will still satisfy these conditions.
- In a *state machine* 12, every transition has exactly one input and one output place. One important feature of the regions of a state machine is that they form a *partitioning* of the state space of the net 3. Figure 8(a) shows an example, where eight regions form the partitioning. The corresponding state machine is depicted in Figure 8(b). Hence, the selection of essential sets may be guided to construct partitionings of the state space and therefore the derivation of a state machine will be guaranteed. The generalization to *conservative Petri nets* is also possible by lifting the notion of partitioning to *multi-partitioning* 3.
- In a *Free-choice* net 8, if two different transitions share a place of their pre-sets, then their pre-sets should be equal (i.e.,  $\bullet t_1 \cap \bullet t_2 \neq \emptyset \implies \bullet t_1 = \bullet t_2$ ). When transforming essential sets into regions by label splitting, an extra condition is required for selecting the sets to split: for each pair of regions derived, either none or the whole set of **exit** events should be shared between the two. This crucial restriction will guarantee to derive Free-choice nets.
- In an *Ordinary* Petri net, arcs have always weight one. The extension to  $k$ -bounded Petri nets described before can be restricted to only consider gradients in the range  $\{-1, 0, +1\}$ , while allowing any multiplicity in the multisets derived. This will ensure the derivation of ordinary Petri nets.

Clearly, by restricting the class of Petri nets to be derived, one can no longer ensure bisimilarity between the derived Petri net and the initial transition system. This is sometimes acceptable, specially in the context of *Process Mining*.



**Fig. 8.** (a) Example of region partitioning. The eight regions shown in the figure are identified by natural numbers, i.e., the region where label  $r$  exits has states with number 1, the region where  $r$  enters has states with number 2 (three states), and so on, (b) State machine corresponding to the partitioning shown in (a) (places contain the region number to which they correspond). Notice that this state machine does not contain the transitions  $em$  and  $s$ , since they are concurrent with some transitions in the net of Figure 8(b).

**Application to Process Mining.** The *discovery* of a formal process model from a set of executions (called *log*) of a system, the *conformance* of a model in describing a log, and the *enhancement* of a process model are the main disciplines in the novel area of Process Mining [16]. Petri nets are one of the most popular models used in Process Mining. Importantly, the synthesis conditions of this paper are relaxed in the context of Process Mining [5], since the derivation of a *simple* model is often preferred even if some extra traces are possible in the model while not observed in the log (this phenomenon is called *overapproximation*). Simplicity here may have various meanings, but it usually implies that the underlying graph of the derived Petri net can be understood by a human. By requiring the minimal number of labels to be split, the approach presented in this paper may be interesting in the context of Process Mining, since the by-product of this optimization is the derivation of Petri nets with minimal number of transitions, thus being more readable. Hence, in Process Mining the synthesis conditions may be required for particular events in order to control the degree of overapproximation that the derived Petri net will have. When a particular event is forced to satisfy the synthesis conditions, it can be guaranteed with the theory of this paper that the number of labels will be minimal if label splitting on essential sets is applied.

## 7 Conclusions

This paper has presented a fresh look at the problem of label splitting, by relating it to some well-known NP-complete problems like chromatic number or

set covering. In a restricted application of label splitting based on essential sets, optimality is guaranteed if certain conditions hold.

Extensions of the technique have been discussed, motivating the appropriateness of using essential sets for label splitting since it allows to control the derived Petri net: the theory can be adapted to produce  $k$ -bounded Petri nets and interesting Petri net classes. Also, one potential application of the technique is presented in the area of Process Mining.

As future work, there are some research lines to follow. First, regarding the technique presented in this paper, it will be very important to study stronger constraints on the transition system that makes the label splitting on essential sets technique not to be an iterative process, as happens with the one presented in this paper. Second, addressing the general problem with unrestricted application of splitting (i.e., not using essential sets but instead arbitrary selections of labels to split) might be an interesting direction to follow. Moreover, incorporating the presented techniques (e.g., Algorithm 2) into our synthesis tool 5 will be considered.

**Acknowledgements.** We would like to thank anonymous reviewers for their comments that help improving the final version of the paper. This work has been supported by the projects FORMALISM (TIN2007-66523) and TIN2011-22484.

## References

1. Arnold, A.: Finite Transition Systems. Prentice Hall (1994)
2. Bernardinello, L., Michelis, G.D., Petruni, K., Vigna, S.: On synchronic structure of transition systems. In: Proceedings of the International Workshop on Structures in Concurrency Theory (STRICT), pp. 69–84 (May 1995)
3. Carmona, J.: Projection approaches to process mining using region-based techniques. *Data Min. Knowl. Discov.* 24(1), 218–246 (2012)
4. Carmona, J., Cortadella, J., Kishinevsky, M.: A Region-Based Algorithm for Discovering Petri Nets from Event Logs. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 358–373. Springer, Heidelberg (2008)
5. Carmona, J., Cortadella, J., Kishinevsky, M.: New region-based algorithms for deriving bounded Petri nets. *IEEE Transactions on Computers* 59(3), 371–384 (2009)
6. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms. McGraw-Hill Higher Education (2001)
7. Cortadella, J., Kishinevsky, M., Lavagno, L., Yakovlev, A.: Deriving Petri nets from finite transition systems. *IEEE Transactions on Computers* 47(8), 859–882 (1998)
8. Desel, J., Esparza, J.: Free-choice Petri Nets. Cambridge Tracts in Theoretical Computer Science, vol. 40. Cambridge University Press (1995)
9. Desel, J., Reisig, W.: The synthesis problem of Petri nets. *Acta Inf.* 33(4), 297–315 (1996)
10. Ehrenfeucht, A., Rozenberg, G.: Partial (Set) 2-Structures. Part I, II. *Acta Informatica* 27, 315–368 (1990)
11. Kishinevsky, M., Kondratyev, A., Taubin, A., Varshavsky, V.: Concurrent Hardware: The Theory and Practice of Self-Timed Design. John Wiley and Sons, London (1993)

12. Murata, T.: Petri Nets: Properties, analysis and applications. Proceedings of the IEEE, 541–580 (April 1989)
13. Nielsen, M., Rozenberg, G., Thiagarajan, P.: Elementary transition systems. Theoretical Computer Science 96, 3–33 (1992)
14. Petri, C.A.: Kommunikation mit Automaten. PhD thesis, Bonn, Institut für Instrumentelle Mathematik (1962) (technical report Schriften des IIM Nr. 3)
15. Reisig, W., Rozenberg, G. (eds.): APN 1998. LNCS, vol. 1491. Springer, Heidelberg (1998)
16. van der Aalst, W.M.P.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer (May 2011)
17. West, D.B.: Introduction to Graph Theory. Prentice-Hall (1996)

# Distributed Control of Discrete-Event Systems: A First Step<sup>\*</sup>

Philippe Darondeau<sup>1</sup> and Laurie Ricker<sup>2</sup>

<sup>1</sup> INRIA Rennes-Bretagne Atlantique, Campus de Beaulieu, Rennes, France  
darondeau@irisa.fr

<sup>2</sup> Department of Mathematics & Computer Science, Mount Allison University,  
Sackville NB, Canada  
lricker@mta.ca

**Abstract.** Distributed control of discrete-event systems (DES) means control implemented by asynchronous message passing automata that can neither perform synchronized actions nor can read one another's state. We explain some significant differences between this emerging area and earlier forms of control. Distributed control synthesis is challenging. To initiate a discussion on this topic, we outline a methodology based on the synthesis of distributable Petri nets (PN). The methodology is illustrated via a well-known example from distributed computing, the dining philosophers, for which three distributed solutions are produced. The paper provides a survey of DES control for PN researchers and a survey of distributed PN synthesis for DES researchers, with the intent to create a common basis for further investigation of this research track.

## 1 Introduction

Developing control strategies for plants composed of a number of distributed components that communicate via an asynchronous network, e.g., power grids and traffic systems, is a significant challenge. In contrast to the control of centralized systems, communication in a distributed system takes time, and there is no immediate way to synchronize events in different locations. In this paper, we focus on the control of discrete-event systems (DES) in a distributed setting, namely, synthesizing a set of controllers that communicate via an asynchronous network. Distributed control has largely been neglected in the literature on DES: we provide a brief survey of DES control in section 2 for the benefit of Petri net researchers. In this introduction, we preface the literature review with a comparison between *monolithic* control and *distributed* control, the two extremes of the evolution outlined in section 2.

A DES  $G$  over set of events  $\Sigma$  may be represented by a deterministic automaton  $(Q, \Sigma, \delta, q_0, Q_F)$  called the *plant*, where  $\delta : Q \times \Sigma \rightarrow Q$  is the partial transition function,  $q_0$  is the initial state, and  $Q_F$  is the set of final (or “marked”)

---

<sup>\*</sup> Work partially supported by the EC-FP7 under project DISC (Grant Agreement INFSO-ICT-224498) and NSERC.

states. Given a specification  $Spec$  of the desired behavior of  $G$ , the goal of control is to enforce this specification on  $G$  and to ensure that final states can always be reached. In this paper, we ignore final states and consider simplified plants of the form  $G = (Q, \Sigma, \delta, q_0)$ . A parameter of the control problem is the data of the observable and controllable events. For the remainder of the paper, we consider disjoint partitions of the alphabet  $\Sigma = \Sigma_o \uplus \Sigma_{uc} = \Sigma_c \uplus \Sigma_{uc}$ , where  $\Sigma_o$  and  $\Sigma_c$  are the sets of observable and controllable events, respectively, and we assume that every controllable event is observable, i.e.,  $\Sigma_c \subseteq \Sigma_o$ . The centralized supervisory control problem consists of constructing a *monolithic supervisor*  $S$  over the set of events  $\Sigma_o$ . That is, we want a deterministic automaton  $S = (X, \Sigma_o, \delta, x_o)$ , such that the partially synchronized product of  $S$  and  $G$ , usually denoted by  $S \times G$  but here denoted by  $(S/G)$ , satisfies  $Spec$  and the following *admissibility* requirement is met: for every reachable state  $(x, q)$  of  $(S/G)$ , if an uncontrollable event  $e \in \Sigma_{uc}$  is enabled at state  $q$  in  $G$ , then  $e$  should also be enabled at state  $x$  in  $S$ .

By *distributed control*, we mean a situation in which several local supervisors and a DES cooperate as follows. First, the set of events of the DES is partitioned over the different locations ( $\Sigma = \uplus_{\ell \in \mathcal{L}} \Sigma_\ell$ ), and for each location  $\ell \in \mathcal{L}$ , every event in  $\Sigma_\ell$  results from the synchronized firing of two  $e$ -labelled transitions, in the DES and in the local supervisor  $S_\ell$ , respectively. Second, the set of events of each local supervisor  $S_\ell$  is the union of  $\Sigma_\ell$  and a set  $X_\ell$  of auxiliary *send* and *receive* events, used to communicate with the other local supervisors in *asynchronous message passing mode*. Messages sent from one location to another are never lost; however, they may overtake one another and the arrival times cannot be predicted. Distributed supervisors of this type can be implemented almost directly on *any* distributed architecture.

Computing a monolithic supervisor and distributing or desynchronizing this supervisor *a posteriori* cannot work unless special conditions guaranteeing distributability have been imposed *a priori*. Thus, special methods must be investigated for distributed supervisor synthesis. We are aware of only two efforts in this direction. Synthesis algorithms for distributed controllers avoiding forbidden states in infinite systems have been proposed in [18]. The local supervisors communicate by FIFO buffers, which might compromise the effective solution of decision problems, but this obstacle is circumvented by regular over-approximation techniques. The main drawback of the approach is an excess of communication since any local supervisor must inform directly or indirectly all other supervisors of every event occurring locally. Prior work along the same line was presented in [37] for distributed state estimation. A different approach was taken in [9], where distributed Petri net synthesis is proposed as an effective technique to solve the distributed supervisory control problem for tolerances. Given a plant  $G$  and two prefix-closed regular languages  $L_{min}$  and  $L_{max}$  such that  $L_{min} \subseteq L_{max} \subseteq L(G)$ , where  $L(G)$  is the language generated by  $G$ , the problem consists of constructing a supervisor  $S$  such that  $L_{min} \subseteq L(S/G) \subseteq L_{max}$ . The distributed version of this problem was addressed under the auxiliary assumption that supervisor  $S$  may be realized as the reachability graph of a distributed Petri net. This ensures

that  $S$  can always be expressed as an asynchronously communicating automaton, as required. The approach may help enforce safety objectives on services while guaranteeing a minimal service, but it has high algorithmic complexity (inherited from Chernikova’s algorithm [6]). However, after attempts to apply the theory to other significant examples found in the literature on DES supervisory control (e.g., train systems, manufacturing or workflow systems with shared resources), we convinced ourselves that in most systems of interest, the main control objective is to avoid deadlocks or to enforce home states or both. It is simply impossible to express such objectives by tolerance specifications because  $L_{min}$  is not known beforehand.

In this paper, we suggest a possible way to synthesize distributed supervisory controllers that avoid deadlocks or enforce home states. We apply distributed Petri net synthesis techniques as was done in [9], but instead of computing least over-approximations of regular languages by Petri net languages (which requires the use of Chernikova’s algorithm), we compute distributed monitor places induced by regions of the plant separating events from states, which can be done in polynomial time. There is no obstacle for full automation of this method, except that the algorithm is non-deterministic and one must handle the global reachability graph of the plant, even though the plant is distributed. A survey of distributed Petri nets, their synthesis and their translation to asynchronous communicating automata is given in section 3. In the framework presented here, messages and communications between components are entirely synthesized, which is novel in DES control. On this basis, our proposed algorithm for distributed controller synthesis is outlined in section 4, where the limitations of the present work are also indicated. Many variations of the algorithm are possible since every rule, based on the order in which the unwanted transitions of  $G$  are examined and removed, determines a corresponding distributed controller synthesis algorithm that runs in polynomial time in the size of  $G$ .

To illustrate the method, we have chosen the  $n = 3$  version of the classical  $n$  dining philosophers problem [17], where the local supervisors are in the forks and both hands of each philosopher may act concurrently. This example is small, but, in our opinion, not trivial. We succeeded in producing three distributed solutions to this problem that could not have been discovered in the absence of an algorithmic strategy and a software tool. This example and the subsequent solutions are presented in section 5. Our ongoing work includes extending our software tool so that we can apply our technique to much larger systems.

## 2 Decentralized and Asynchronous DES Control

In this section, we explain our understanding of the gradual shift from centralized control towards distributed control of discrete-event systems (DES), an area still in its infancy but one that may soon become vital for applications of the theory.

Ramadge and Wonham’s theory of non-blocking supervisory control for DES was introduced in [26] and developed soon after in [21,27] to cover situations in which only some of the events generated by the DES are observed, for instance “in situations involving decentralized control” [27]. The basic supervisory



control problem consists of deciding, for a DES with observable/unobservable and controllable/uncontrollable events, whether a specified behavior may be enforced on the DES by some admissible control law. This basic supervisory control problem is decidable. Moreover, if every controllable event is also observable, then one can effectively compute the largest under-approximation of the specified behavior that can be enforced by the supervisory control.

After the work done in [31] and extended in [38], *decentralized control* refers to a form of control in which several local supervisors, with different subsets of observable events and controllable events, cooperate in rounds. In each round, the local supervisors compute collaboratively the set of all authorized events, and the DES performs sequences of events taken from this set, where each sequence is comprised of unobservable events followed by at most one observable event that ends the round. In each round, the cooperation between the supervisors may result from an implicit synchronization, e.g., in conjunctive coordination [31], or be obtained by applying distributed algorithms, i.e., to reach a consensus on the set of authorized events. In any case, the controlled DES is governed by a global clock, whose ticks are the occurrences of observable events. This global clock may not be a problem for embedded system controllers, especially when they are designed using synchronous programming languages like Lustre [15]. Yet global clocks are impractical for widely-distributed systems or architectures, not to mention for distributed workflows or web services. Global clocks are also impractical in fault-tolerant distributed systems, such as automotive control systems. This explains why Globally Asynchronous Locally Synchronous (GALS) architectures have been developed [25]. The existence of non-blocking decentralized supervisory control for the basic supervisory control problem is decidable in the special case where the “closed” behavior of a DES is equal to the prefix-closure of its “marked” behavior (the set of event sequences that lead to some marked states), but this problem is undecidable in the general case without communication [19,32].

Decentralized control can be extended to the case where supervisors communicate [3]. Communication between supervisors is helpful when the correct control decision cannot be taken by any of the local supervisors in a cooperative round of decision making. The idea is to ensure that supervisors receive relevant information about system behavior that they cannot directly observe so that (at least) one local supervisor can make the correct control decision. Further, it may be desirable to synthesize an optimal communication protocol, either from a set-theoretic (e.g., [30]) or quantitative (e.g., [29]) perspective (cardinality of the information set matters in the former case, while the frequency of occurrence of elements in the information set are relevant in the latter case). For the most part, work in this area has been devoted to the study of synchronous communication protocols, i.e., protocols that specify which information each supervisor should communicate to the other supervisors within each round. There are several different approaches: build a correct communication protocol using a bottom-up approach [3,28]; given a correct communication protocol, use a top-down approach to reduce it to one that is optimal from a set-theoretic

perspective [30,34]; and apportion an optimal centralized solution among a set of communicating supervisors [22].

Decentralized control with communication has also been studied in the context of asynchronous communication protocols, i.e., under the assumption that information sent by a supervisor to another supervisor in a cooperative round may be received in a later round. Bounded and unbounded communication delays (counted in cooperative rounds) have been explored in [33,16]. Communication channels are modeled as FIFO buffers; however, there is no notion of optimality since the communication protocol for each supervisor is to simply send all of its observations to the other supervisors. In the case of unbounded delay, the decentralized control problem is undecidable [33,16].

Decentralized control with synchronous or asynchronous communication extends the basic decentralized control problem in the general direction of distributed control; however, the approach fails to reach this objective. On the one hand, it is not realistic to envisage synchronous communication in a distributed context. On the other hand, if asynchronous communication is assumed, then either *distant synchronization* or some kind of *arbitration* [20] is necessary to enforce decentralized control, but, again, distant synchronization and arbitration are not realistic in a distributed context.

In the rest of the section, we would like to focus on two specific models of decentralized control (with or without communication) proposed in the literature: one based upon asynchronous automata (requiring synchronous communication), and the other based upon the asynchronous product of automata (requiring no communication).

In [24], decentralized control is investigated in the framework of recognizable trace languages and asynchronous automata [40,23,11]. In an asynchronous automaton, states are vectors of local states, and transitions are defined on *partial* states, i.e., projections of states on a subset of locations. Transitions that concern disjoint subsets of locations produce independent events, hence asynchronous automata have *partially ordered runs* (*ergo* there is no global clock). Recognizable trace languages have a close relationship with one-safe Petri nets. Indeed, when locations are in bijective correspondence with places of nets, one-safe nets generate prefix-closed deterministic and recognizable trace languages. However, recognizable trace languages need not be prefix-closed, hence the presence of global final states in asynchronous automata. In [24], a DES is an asynchronous automaton where an uncontrollable event has exactly one location while a controllable event may have several locations. Every event, i.e., occurrence of a transition, is observed in all locations concerned in the generating transition. The goal is to construct for all locations *local supervisors* that jointly enforce a specified behavior on the DES, i.e., a specific subset of partially ordered runs. Local supervisors cooperate in two ways. First, a controllable event with multiple locations cannot occur in the controlled DES unless it is enabled by all supervisors in these locations. Second, at each occurrence of a controllable event, all local supervisors concerned with this event synchronize and communicate to one another all information that they have obtained so far by direct observation of

local events and from prior communication with other supervisors. The information available to the supervisor in location  $\ell$  is therefore the  $\ell$ -view of the partially ordered run of the DES, i.e., the causal past of the last event with location  $\ell$ . The local control in location  $\ell$  is a map from the  $\ell$ -view of runs to subsets of events with location  $\ell$ , including all uncontrollable events that are observable at this location. Implementing asynchronous supervisors, as they are defined above, on distributed architectures first requires the implementation of a protocol for synchronizing the local supervisors responsible for an event whenever the firing of this event is envisaged. One of the main contributions of [24] is the identification of conditions under which one can effectively decide upon the existence of asynchronous supervisors, which, moreover, can always be translated to *finite* asynchronous automata.

In [7], the goal is not to synthesize directly a decentralized supervisor but, rather, to transform a centralized (non-blocking and optimal) supervisor into an equivalent system of local supervisors with *disjoint* subsets of controllable events. (A non-blocking supervisor guarantees that the plant may always reach some final state.) The data for the problem are a DES  $G$  and a centralized supervisor  $S$ , defined over a set  $\Sigma$  of observable/unobservable and controllable/uncontrollable events, plus a partition  $\Sigma = \uplus_{\ell \in \mathcal{L}} \Sigma_\ell$  of  $\Sigma$  over a finite set of locations. All local supervisors have the set of events  $\Sigma$ , but the local supervisor  $S_\ell$  in location  $\ell$  is the sole supervisor that can disable controllable events at location  $\ell$ . The states of  $S_\ell$  are the cells of a corresponding *control cover*. A control cover is a covering of the set of states of  $S$  by *cells* with the following properties. First, if two transitions from a cell are labelled by the same event, then they must lead to the same cell. Second, if two states  $x$  and  $x'$  are in the same cell, then for any reachable states  $(x, q)$  and  $(x', q')$  of  $S/G$  and for any event  $e$  with location  $\ell$ , if  $e$  is enabled at  $q$  and  $q'$  in  $G$ , then  $e$  should be coherently enabled or disabled at  $x$  and  $x'$  in  $S$ . Third, two states in a cell should be coherently final or non-final. Supervisor localization based on control covers is universal in the sense that it always succeeds in transforming a centralized supervisor  $S$  into an equivalent family of local supervisors  $S_\ell$ . It is worth noting that a local supervisor  $S_\ell$  with language  $L(S_\ell)$  may be replaced equivalently by a local supervisor with the language  $\pi(L(S_\ell))$  for any natural projection operator  $\pi : \Sigma^* \rightarrow (\Sigma'_\ell)^*$  such that  $\Sigma_\ell \subseteq \Sigma'_\ell \subseteq \Sigma$  and  $L(S_\ell) = \pi^{-1}\pi(L(S_\ell))$ . In the end, for any event  $e \in \Sigma$ , whenever  $e$  occurs in the controlled DES, all local supervisors  $S_\ell$  such that  $e \in \Sigma'_\ell$  should perform local transitions labelled by  $e$  in a *synchronized way*.

The approaches taken in [24] and [7] differ significantly. The former approach is based on asynchronous automata and on the assumption that several local supervisors may be responsible for the same controllable event. The latter approach is based on asynchronous product automata and on the assumption that exactly one local supervisor is responsible for each controllable event. Both approaches, though, rely on similar requirements for the final implementation of the asynchronous supervisors. In both approaches, the only way for the local supervisors to cooperate and/or communicate with one another is by synchronization on shared events; however, as previously noted, distributed architectures

do not generally provide protocols for performing synchronized transitions at different locations.

### 3 Distributed Controller Synthesis Based on Petri Net Synthesis

In this section, we recall the background of Petri nets, as well as a framework for Petri net based distributed controller synthesis [2,9] that we will use in the remainder of the paper.

**Definition 1.** A Petri net  $N$  is a bi-partite graph  $(P, T, F)$ , where  $P$  and  $T$  are finite disjoint sets of vertices, called places and transitions, respectively, and  $F : (P \times T) \cup (T \times P) \rightarrow \mathbb{N}$  is a set of directed edges with non-negative integer weights. A marking of  $N$  is a map  $M : P \rightarrow \mathbb{N}$ . A transition  $t \in T$  is enabled at a marking  $M$  if  $M(p) \geq F(p, t)$  for all places  $p \in P$ . If  $t$  is enabled at  $M$ , then it can be fired, leading to the new marking  $M'$  (denoted by  $M[t]M'$ ) defined by  $M'(p) = M(p) + F(t, p) - F(p, t)$  for all  $p \in P$ .

**Definition 2.** A Petri net system  $N$  is a tuple  $(P, T, F, M_0)$  where  $M_0$  is a marking of the net  $(P, T, F)$ , called the initial marking. The reachability set  $RS(N)$  of  $N$  is the set of all markings reached by sequences of transitions from  $M_0$ .  $N$  is said to be bounded if  $RS(N)$  is finite. The reachability graph  $RG(N)$  of  $N$  is the transition system  $(RS(N), T, \delta, M_0)$  with the partial transition map  $\delta : RS(N) \times T \rightarrow RS(N)$  defined as  $\delta(M, t) = M'$  iff  $M[t]M'$  for all markings  $M$  and  $M'$  in  $RS(N)$ . The language  $L(N)$  of  $N$  is the set of its firing sequences, i.e., words in  $T^*$  that label sequences of transitions from  $M_0$  in  $RG(N)$ .

Recall that a supervisor for a plant  $G = (Q, \Sigma, \delta, q_0)$  is a deterministic automaton  $S = (X, \Sigma_o, \delta, x_o)$ , such that  $(S/G)$  satisfies the following condition: for every reachable state  $(x, q)$  of  $(S/G)$ , if an uncontrollable event  $e \in \Sigma_{uc}$  is enabled at state  $q$  in  $G$ , then  $e$  is also be enabled at state  $x$  in  $S$ . We say that  $S$  is *Petri net definable* (PND) if it is isomorphic to the reachability graph of a Petri net system  $N$  with the set of transitions  $T = \Sigma_o$ . Thus,  $S$  is PND if there exists a Petri net system  $N = (P, T, F, M_0)$  with the set of transitions  $T = \Sigma_o$  and a bijection  $\varphi : X \rightarrow RS(N)$  such that  $\varphi(x_0) = M_0$  and for all  $x \in X$  and  $t \in \Sigma_o$ ,  $\delta(x, t)$  is defined if and only if  $M[t]M'$  for  $M = \varphi(x)$  and for some marking  $M'$ , and then  $M' = \varphi(x')$  where  $x' = \delta(x, t)$ .

PND controllers comprised of *monitor places* for PND plants date back to the work in [10,39]. Monitor places are linear combinations of places of the net that define the plant (seen as rows of the incidence matrix of the net), and they are added to the existing places of this net in order to constrain its behavior. PND controllers for DES given as labelled transition systems proceed from a similar idea with one significant difference: since  $G$  has no places, the places of the controller net  $N$  are synthesized directly from *Spec* (the specification) using the theory of regions, which has been developed together with synthesis algorithms both for DES and for languages [11,2,8]. This controller net synthesis technique

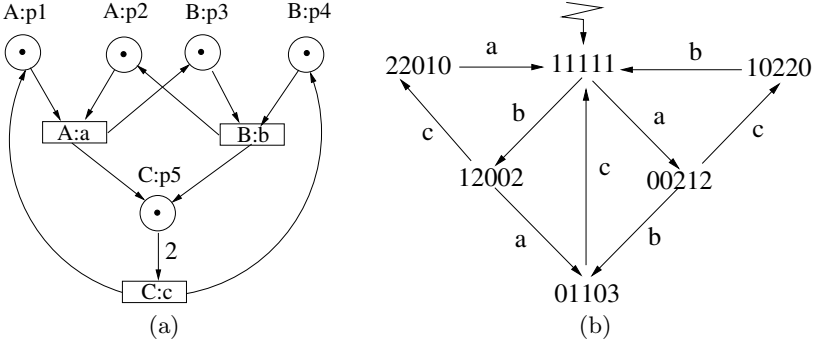
was inaugurated in [13]. In that work, the specification  $Spec$  was the induced restriction of  $G$  on a subset of “good” states. In the considered case where  $Spec$  is a DES, a region of  $Spec$  may be seen as a Petri net system  $N_p$  with a set of transitions equal to the set of transition labels in  $Spec$  and with a single place  $p$  such that  $Spec$  and  $Spec \times RG(N_p)$  are isomorphic, i.e., monitoring  $Spec$  by  $N_p$  does not affect the behaviour of  $Spec$  in any way. In the alternative case where  $Spec$  is a language over the alphabet  $T$ , a region of  $Spec$  may be seen as a Petri net system  $N_p$  with the set of transitions  $T$  and with a single place  $p$  such that  $Spec \cap L(N_p) = Spec$ . The theory of regions may thus be applied both to state-oriented specifications given by transition systems and to event-oriented specifications given by languages.

Since the Petri nets that we consider are labelled injectively (their set of transitions  $T$  is equal to the set  $\Sigma_o$  of the observable events), it may seem a bit odd to search for PND controllers instead of general controllers defined by labelled transition systems. It is well known that languages of injectively labelled and bounded Petri nets form a *strict subset* of the regular languages. So, while one loses generality, one gains only the paltry compensation of a more compact representation of controllers. Fortunately, the situation changes radically when one takes distributed control into account, because in this case controller net synthesis can be tailored to distributed Petri nets. In contrast to general deterministic automata, distributed Petri nets can always be converted to asynchronously communicating automata (as we explain below). We contend that the availability of this automated translation, defined in [2], is a fair compensation for the loss of generality, which explains the approach presented in [9] and the modified approach suggested in the end of this paper. Indeed, we do not know of any other approach where we can synthesize communications between controllers while at the same time avoiding the generation of useless communications. This aspect of the approach provides us with strong motivation for the use of distributed Petri nets. We recall now from [2] the definition of distributed Petri nets and their automated translation to asynchronous message passing automata.

**Definition 3 (Distributed Petri net system).** *A distributed Petri net system over a set of locations  $\mathcal{L}$  is a tuple  $N = (P, T, F, M_0, \lambda)$  where  $(P, T, F, M_0)$  is a Petri net system and  $\lambda : T \rightarrow \mathcal{L}$  is a map, called a location map, subject to the following constraint: for all transitions  $t_1, t_2 \in T$  and for every place  $p \in P$ ,  $F(p, t_1) \neq 0 \wedge F(p, t_2) \neq 0 \Rightarrow \lambda(t_1) = \lambda(t_2)$ .*

*Example 1.* A distributed Petri net system is depicted in Fig. 1(a). The set of locations is  $\mathcal{L} = \{A, B, C\}$  and each transition  $x \in \{a, b, c\}$  has the corresponding location  $X$ . The reachability graph of this net system is shown in Fig. 1(b).  $\diamond$

In a distributed Petri net, two transitions with different locations cannot compete for tokens, hence *distributed conflicts cannot occur*, which makes an effective implementation straightforward. Such a restriction has been mentioned in [36] in the specific context of Elementary Net Systems (where a place may contain at most one token). Two transitions with different locations may, though, *send* tokens to the same place. Implementing a distributed Petri net system  $N$  means



**Fig. 1.** (a) A distributed Petri net and (b) its reachability graph

producing an asynchronous message passing automaton (AMPA) behaving like  $N$  up to branching bisimulation (see Appendix for precise definitions). Given any non-negative integer  $B$ , let  $RG_B(N)$  denote the induced restriction of the reachability graph  $RG(N)$  of  $N$  on the subset of markings bounded by  $B$ , i.e., markings  $M$  such that  $M(p) \leq B$  for all places  $p \in P$ . If  $N$  defines a controller for plant  $G$  and  $B$  is the maximum of  $M(p)$  for all places  $p \in P$  and for all markings  $M$  of  $N$  reached in the partially synchronized product  $RG(N)/G$ , then  $RG(N)/G$  and  $RG_B(N)/G$  are isomorphic even though  $N$  may be unbounded. Transforming  $N$  into a  $B$ -bounded AMPA with a reachability graph branching bisimilar to  $RG_B(N)$  may be done as follows (see [2] for a justification). The bound  $B$  is used only in the last step of the transformation, but its role there is crucial.

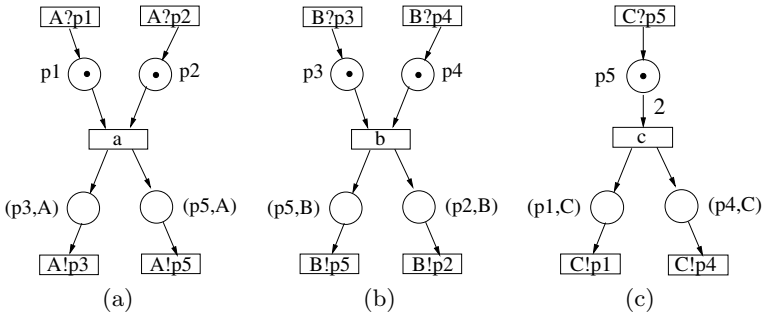
- Given  $N = (P, T, F, M_0, \lambda)$ , extend  $\lambda : T \rightarrow \mathcal{L}$  to  $\lambda : (T \cup P) \rightarrow \mathcal{L}$  such that  $\lambda(p) \neq \lambda(t) \Rightarrow F(p, t) = 0$  for all  $p \in P$  and  $t \in T$  (this is possible by Def. 3).
- For each location  $\ell \in \mathcal{L}$ , construct a net system  $N_\ell = (P_\ell, T_\ell, F_\ell, M_{\ell,0})$ , called a *local net*, as follows.
  - $P_\ell = \{p \mid p \in P \wedge \ell = \lambda(p)\} \cup \{(p, \ell) \mid p \in P \wedge \ell \neq \lambda(p)\}$ ,
  - $T_\ell = \{t \in T \mid \lambda(t) = \ell\} \cup \{\ell!p \mid p \in P \wedge \ell \neq \lambda(p)\} \cup \{\ell?p \mid p \in P \wedge \ell = \lambda(p)\}$ ,
  - $F_\ell(p, t) = F(p, t)$  and  $F_\ell(t, p) = F(t, p)$ ,
  - $F_\ell(t, (p, \ell)) = F(t, p)$ ,
  - $F_\ell((p, \ell), \ell!p) = 1$  and  $F_\ell(\ell?p, p) = 1$ ,
  - $M_{\ell,0}(p) = M_0(p)$ .

In each local net  $N_\ell$ , places  $(p, \ell)$  are local clones of places  $p$  of other local nets. Whenever a transition  $t \in T$  with location  $\ell$  produces tokens for a *distant* place  $p \in P$  ( $\lambda(t) = \ell \neq \lambda(p)$ ), the transition  $t \in T \cap T_\ell$  produces tokens in the local clone  $(p, \ell)$  of  $p$ . These tokens are removed from the local clone  $(p, \ell)$  of  $p$  by the auxiliary transition  $\ell!p$ , each firing of which models an asynchronous emission of the *message*  $p$ . Symmetrically, for any place  $p$  of  $N$  with location  $\ell$ , each firing of the auxiliary transition  $\ell?p$  models an

asynchronous reception of the *message*  $p$ , resulting in one token put in the corresponding place  $p \in P_\ell$ .

- For each location  $\ell$ , compute  $RG_B(N_\ell)$ . The desired *AMPA* is the collection of local automata  $\{A_\ell = RG_B(N_\ell) \mid \ell \in \mathcal{L}\}$ . Since any local net  $N_\ell$  may be unbounded, the bound  $B$  is needed here to guarantee that the local automata  $A_\ell$  are finite. By construction, the *AMPA* is branching bisimilar to  $RG_B(N)$ , but *not* necessarily to  $RG(N)$ . Each transition of an automaton  $A_\ell$  is labelled either with some  $t \in T \cap T_\ell$  or with some asynchronous message emission  $\ell!p$  or with some asynchronous message reception  $\ell?p$ . A message  $p$  sent from a location  $\ell'$  by a transition  $\ell'!p$  of the automaton  $A_{\ell'}$  is automatically routed towards the automaton  $A_\ell$  with the location  $\ell = \lambda(p)$ , where it is received by a transition  $\ell?p$  of the automaton  $A_\ell$ . No assumption is made on the relative speed of messages, nor on the order in which they are received.

*Example 2.* The local net systems  $N_A, N_B$  and  $N_C$  computed from the net  $N$  of Fig. 1 are displayed from left to right in Fig. 2. The bound  $B$  is equal to 3.  $\diamond$



**Fig. 2.** Three local net systems

Finally, let us return to controllers. Let  $G$  be a DES over a set of events  $\Sigma$  and  $Spec$  be a specification of the desired behavior of  $G$  (we purposely remain vague about the nature of specifications, which may be isomorphism classes of DES, languages, modal formulas, and so on). Let  $S = (X, \Sigma_o, \delta, x_o)$  be a finite-state admissible supervisor for  $G$ , such that  $(S/G)$  satisfies the specification  $Spec$ . We say that  $S$  is a *distributed Petri net definable supervisor* (DPND) if it is isomorphic to the reachability graph of a distributed Petri net  $N = (P, T, F, M_0, \lambda)$  with the set of transitions  $T = \Sigma_o$  and the location map  $\lambda : \Sigma_o \rightarrow \mathcal{L}$  defined by  $\lambda(e) = \ell$  if  $e \in \Sigma_\ell$ , where  $\lambda(e)$  indicates the locations where  $e$  may be observed and possibly controlled. In view of this isomorphism, since  $X$  is finite, there must exist a finite bound  $B$  such that  $M(p) \leq B$  for all places  $p \in P$  and for all reachable markings  $M$  of  $N$ . Therefore,  $S$  may be translated to an equivalent *AMPA*  $= \{A_\ell = RG_B(N_\ell) \mid \ell \in \mathcal{L}\}$ , realizing, in the end, *fully distributed control*.



## 4 Towards Petri Net Based Distributed Controller Synthesis

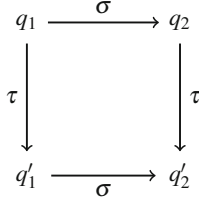
The approach to distributed control that we suggest is to search for admissible DPND controllers using Petri net synthesis techniques (see [2,8]). These techniques allow us to answer the following questions:

- Given a finite automaton over  $\Sigma_o$  and a location map  $\lambda : \Sigma_o \rightarrow \mathcal{L}$ , can this automaton be realized as the reachability graph of a distributed Petri net  $N = (P, T, F, M_0, \lambda)$  with set of transitions  $T = \Sigma_o$ ?
- Given a regular language over  $\Sigma_o$  and a location map  $\lambda : \Sigma_o \rightarrow \mathcal{L}$ , can this language be realized as the set of firing sequences of a distributed Petri net  $N = (P, T, F, M_0, \lambda)$  with set of transitions  $T = \Sigma_o$ ?

The type of net synthesis techniques that should be applied when solving the distributed supervisory control problem depends upon *Spec*, the specification of the control objective to be enforced on the plant  $G$ . As previously noted, when the control objective is to avoid deadlocks or enforce home states, the detailed method defined in [9] to derive distributed supervisors for the basic supervisory control problem cannot be applied. In fact, we do not know of any optimal method to cope with these significant and practical problems. Nevertheless, to initiate work in this direction, in this section we propose an algorithm for computing distributed supervisors that enforce such objectives, based on Petri net synthesis, that can be performed in polynomial time (w.r.t. the size of  $G$ ). The case study presented in next section shows that this algorithm produces good results when  $G$  is isomorphic to the reachability graph of a one-safe net. For arbitrary plants  $G$ , more elaborate algorithms of the same flavour should be considered and compared for efficiency and for quality of control. Moreover, some completeness results ought to be established. Before we present the algorithm, we would like to insist again on other two limitations. Because *AMPA* derived from distributed Petri nets are a *strict subclass* of *AMPA*, distributed controller synthesis techniques based on Petri net synthesis may fail even though the distributed supervisory control problem under consideration may be solved using more general *AMPA*. Second, to apply Petri net synthesis techniques,  $G$  must be given as a plain transition system, not as a symbolic transition system. A main merit of the general approach that we suggest here is that no comparable approach exists yet. The above points illustrate, if it was not yet entirely clear, that the field for research on distributed controller synthesis remains wide open.

Our algorithm applies to arbitrary plants  $G = (Q, \Sigma, \delta, q_0)$ . In the sequel,  $\Sigma = \Sigma_o \uplus \Sigma_{uo} = \Sigma_c \uplus \Sigma_{uc} = \uplus_{\ell \in \mathcal{L}} \Sigma_\ell$ , where  $\mathcal{L}$  is the set of distributed locations, and we use  $q \xrightarrow{\sigma} q'$  as an alternative representation of  $\delta(q, \sigma) = q'$ . We require two additional definitions before presenting the algorithm. Four transitions  $q_1 \xrightarrow{\sigma} q_2$ ,  $q'_1 \xrightarrow{\sigma} q'_2$  and  $q_1 \xrightarrow{\tau} q'_1$ ,  $q_2 \xrightarrow{\tau} q'_2$  with pairwise identical labels  $\sigma$  and  $\tau$  (see Fig. 3) form a *distributed diamond* in  $G$  if the events  $\sigma$  and  $\tau$  have different locations. We let  $\prec$  denote the reflexive transitive closure of the relation on transitions defined as  $q_1 \xrightarrow{\sigma} q_2 \prec q'_1 \xrightarrow{\sigma} q'_2$  and  $q_1 \xrightarrow{\tau} q'_1 \prec q_2 \xrightarrow{\tau} q'_2$  if the four transitions form a distributed diamond.



**Fig. 3.** A diamond

If the control objective is to avoid deadlocks, then we call  $q \in Q$  a *bad state* if  $\delta(q, \sigma)$  is undefined for all  $\sigma \in \Sigma$ . If the control objective is to enforce the home-state property on  $q_0$ , then  $q \in Q$  is a bad state if  $q_0$  cannot be reached inductively from  $q$ . The algorithm proceeds iteratively by switching states to *bad*, by formally removing transitions, and by computing distributed monitor places to actually block the transitions that have been removed. Formally removing a transition off means replacing  $q \xrightarrow{\sigma} q'$  by the negative assertion  $\neg(q \xrightarrow{\sigma})$ . A negative assertion  $\neg q \xrightarrow{\sigma}$  may be enforced by a *monitor place* [10]. This monitor place is computed as a *region* that *separates* event  $\sigma$  from state  $q$  in the reachable restriction of  $G$  which is obtained after removing the bad states (see [2]). Monitor places  $p$  and the inducing regions are, moreover, subject to distribution constraints that may be expressed as sign constraints. These constraints are the following. First,  $F(p, \tau) = 0 = F(\tau, p)$  for every unobservable transition  $\tau \in \Sigma \setminus \Sigma_o$ ; second,  $F(p, \tau) = 0$  for every transition  $\tau \in \Sigma$  with location  $\lambda(\tau)$  different from  $\lambda(\ell)$ . In the following algorithm, we say that  $w$  is cycle-free (CF) if  $\delta(q_0, u) \neq \delta(q_0, v)$  for all distinct prefixes  $u, v$  of  $w$ . (Note that a state that is not in  $Bad$  is considered to be *good*).

---

**Algorithm 1.**


---

```

while  $Bad \neq \emptyset$  do
  Select  $q_{bad}$  from the set of bad states
  Choose CF  $w$  s.t.  $\delta(q_0, w) = q_{bad}$ ,  $w = w_1 \sigma w_2$  and  $w_1 \in \Sigma^*$ ,  $w_2 \in \Sigma_{uc}^*$ ,  $\sigma \in \Sigma_c$ 
  Set  $\neg \delta(q_0, w_1) \xrightarrow{\sigma}$  in place of  $\delta(q_0, w_1) \xrightarrow{\sigma} \delta(q_0, w_1 \sigma)$  and
    similarly remove all predecessor transitions wrt preorder  $\prec$ 
  Remove all unreachable states
  Recalculate  $Bad$ 
  while  $\exists$  assertions  $\neg q \xrightarrow{\sigma}$  where  $q$  is a good state do
    if  $\exists$  distributed monitor place  $p$  separating  $\sigma$  from  $q$  then
      Add  $p$  to the set of monitor places
      Remove the negative assertion  $\neg q \xrightarrow{\sigma}$ 
    else
      Make  $q$  a bad state
    end if
  end while
end while

```

---

At the time when this non-deterministic algorithm stops, either  $q_0$  has been added to *Bad* and then no solution to the distributed control problem is provided, or a distributed Petri net is obtained by gathering all monitor places, and a distributed controller is then obtained by transforming this net to an *AMPA* as indicated in section 3.

Computing a distributed monitor place by separating an event  $\sigma$  from a state  $q$  takes time polynomial in the size of  $G$ , hence the above algorithm runs in polynomial time. This complexity bound applies to all possible algorithms based on computing monitor places for blocking unwanted transitions. The policy taken here is to remove—in one step—one transition and all predecessor transitions w.r.t.  $\prec$ , but more subtle policies could give better results. Note that the above algorithm is closer in spirit to the ideas in [12] than to the ideas in [9].

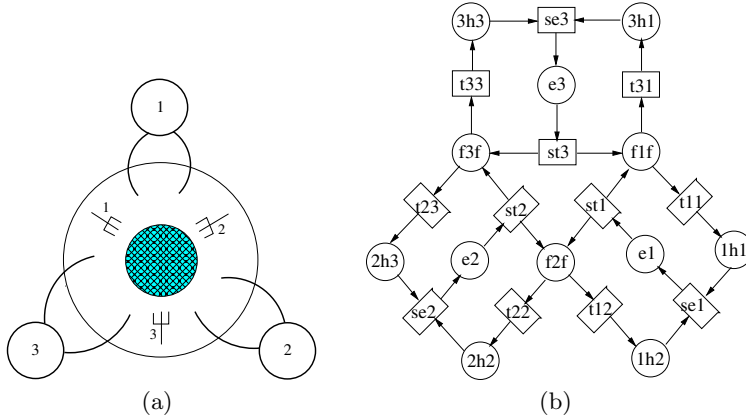
## 5 Distributed Control for Three Dining Philosophers

We would now like to illustrate our Petri net based synthesis method for distributed controllers on a toy example. We have chosen the well-known problem of the dining philosophers [17]. The reasons for this choice are twofold. On the one hand, the problem is easily understood. On the other hand, the size of this problem can be adjusted to accommodate our partially-implemented algorithm by decreasing the usual number of philosophers, which is five, to three.

*Example 3.* Three philosophers  $\varphi_1$ ,  $\varphi_2$ , and  $\varphi_3$  are sitting at a table with a bowl of spaghetti in the center. Three forks  $f_1$ ,  $f_2$ ,  $f_3$  are placed on the table, such that philosopher  $\varphi_i$  has the fork  $f_i$  on his right and the fork  $f_{i+1 \bmod 3}$  on his left (see Fig. 4(a)). A philosopher alternates periods of eating and periods of thinking. To eat, he needs both the fork to his direct left and the fork to his direct right. Therefore, he tries to grab them one after the other while thinking. A philosopher who thinks with a fork in each hand stops thinking and starts eating after a finite delay. A philosopher who eats eventually stops eating, puts down the forks, and starts thinking again after a finite delay. The basic problem is to avoid deadlock, i.e., the situation in which every philosopher has taken one fork. A classical solution is to let all philosophers but one, say  $\varphi_1$ , first take the fork to their right, and then let  $\varphi_1$  first take the fork to his left. An augmented problem is to avoid the starvation of any philosopher.  $\diamond$

The dining philosophers problem was used in [35] to illustrate the use of supervisory control techniques for removing deadlocks from multi-threaded programs with lock acquisition and release primitives. In that work, the emphasis was on optimal control, not on distribution. In this paper, we do the opposite, i.e., we give priority to distributed control over optimal control. Moreover, *we intend that local controllers will be embedded in forks*, not in philosopher threads. The idea is that resource managers have fixed locations while processes using resources are mobile. More specifically, consider the PND plant  $G$  defined by the Petri net  $N$  (shown in Fig. 4(b)).

Places  $f_1f$ ,  $f_2f$ , and  $f_3f$  are the resting places of the forks  $f_1$ ,  $f_2$ , and  $f_3$ , respectively ( $fif$  should be read as “ $f_i$  is free”), and they are initially marked



**Fig. 4.** Modeling the Dining Philosophers Problem:(a) three philosophers; (b) the uncontrolled plant

with one token each. For  $j \in \{1, 2, 3\}$  and  $i \in \{j, j + 1 \bmod 3\}$ , “philosopher  $\varphi_j$  can take fork  $f_i$  when it is free” (transition  $t_{ji}$ ). After this, “philosopher  $\varphi_j$  holds fork  $f_i$ ” (condition  $jh_i$ ). A philosopher  $\varphi_j$  with two forks may “start eating” (transition  $se_j$ ). A philosopher  $\varphi_j$  who is eating (condition  $e_j$ ) can “start thinking” (transition  $st_j$ ), and then forks  $f_j$  and  $f_{j+1 \bmod 3}$  return again to the table. With respect to distribution, there are four locations as follows. For each fork  $f_i$ ,  $i \in \{1, 2, 3\}$ , the two transitions which compete for this fork, namely  $t_{ii}$  and  $t_{ji}$  with  $j = i - 1 \bmod 3$ , have location  $i$  given by the index of this fork. All other transitions are given by default to another location 4 that does not matter.

The control objective is to avoid deadlocks. For simplicity, we assume that all transitions are observable. The controllable transitions are the transitions which consume resources, i.e., the transitions  $t_{ji}$  (philosopher  $\varphi_j$  takes fork  $f_i$ ). The control objective should be achieved by distributed control and, more precisely, by a distributed Petri net. The set of transitions  $T$  of the controller net may be any set included in  $\{se_j, st_j, t_{jj}, t_{ji} \mid 1 \leq j \leq 3 \wedge i = j + 1 \bmod 3\}$ . The location map  $\lambda : T \rightarrow \{1, 2, 3, 4\}$  is the induced restriction of the map defined by  $\lambda(t_{ij}) = j$  and  $\lambda(se_j) = \lambda(st_j) = 4$  for  $1 \leq j \leq 3$ .

The PND plant  $G$  under consideration, i.e., the reachability graph of  $N$ , has two sink states  $M_1$  and  $M_2$ , defined by  $M_1(3h_1) = M_1(1h_2) = M_1(2h_3) = 1$  and  $M_2(1h_1) = M_2(2h_2) = M_2(3h_3) = 1$ , respectively (letting  $M_1(p) = 0$  or  $M_2(p) = 0$  for all other places  $p$ ). If one disregards *distributed* control, it is quite easy to eliminate these two deadlocks by adding two monitor places  $p_1$  and  $p_2$ , the role of which is to disable the instances of the transitions  $t_{31}, t_{12}, t_{23}$  and  $t_{11}, t_{22}, t_{33}$  that reach  $M_1$  and  $M_2$  in one step, respectively. The monitor places may be chosen so that no other transition instance is disabled, hence they do not cause new deadlocks. Two such monitor places are  $p_1 = 2 + st_1 + st_2 + st_3 - t_{31} - t_{12} - t_{23}$  (i.e.,  $p_1$  is initially marked with 2 tokens, there are flow arcs with weight 1 from

$st_1$ ,  $st_2$  and  $st_3$  to  $p_1$ , and there are flow arcs with weight 1 from  $p_1$  to  $t_{31}$ ,  $t_{12}$  and  $t_{23}$ ), and  $p_2 = 2 + st_1 + st_2 + st_3 - t_{11} - t_{22} - t_{33}$ . The PND controller consisting of the monitor places  $p_1$  and  $p_2$  realizes the optimal control of  $G$  for deadlock avoidance. This optimal controller is unfortunately not a distributed controller. For one-safe nets, optimal control can always be realized by PND controllers [12], but the problem is different for DPND controllers.

### 5.1 A Distributed Controller That Avoids Deadlocks

The reachability graph  $G$  of  $N$  has 36 states and 78 transitions. An inspection of  $G$  allows us to identify the two subgraphs shown in Fig. 5, where the initial state is 0 and the deadlock states  $M_1$  and  $M_2$  are numbered 25 and 11, respectively. All six transitions that lead in one step to either of the deadlock states 25 and 11 are represented in Fig. 5 and similarly for all predecessors of the deadlocking transitions  $26 \xrightarrow{t_{31}} 25$  and  $16 \xrightarrow{t_{22}} 11$  w.r.t. preorder  $\prec$ . The following distributed diamonds, given by pairs of vertical edges, have been omitted:  $(0 \xrightarrow{t_{22}} 5, 1 \xrightarrow{t_{22}} 6)$ ,  $(0 \xrightarrow{t_{12}} 23, 1 \xrightarrow{t_{12}} 24)$ ,  $(30 \xrightarrow{t_{22}} 8, 29 \xrightarrow{t_{22}} 12)$  for the leftmost subgraph, and  $(0 \xrightarrow{t_{31}} 1, 5 \xrightarrow{t_{31}} 6)$ ,  $(0 \xrightarrow{t_{33}} 19, 5 \xrightarrow{t_{33}} 10)$ ,  $(19 \xrightarrow{t_{31}} 2, 10 \xrightarrow{t_{31}} 7)$  for the rightmost subgraph. Three occurrences of  $t_{31}$ , resp.  $t_{22}$  in  $G$  do moreover not appear in the leftmost, resp. rightmost subgraph but none of them precedes  $26 \xrightarrow{t_{31}} 25$  nor  $16 \xrightarrow{t_{22}} 11$  (w.r.t.  $\prec$ ).

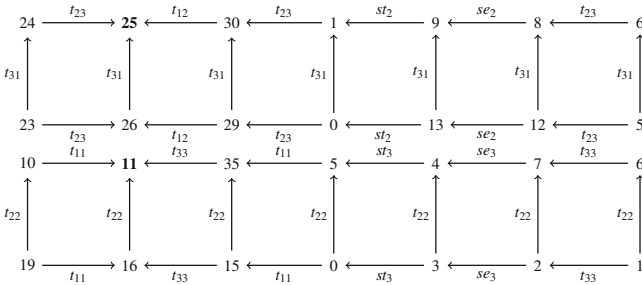
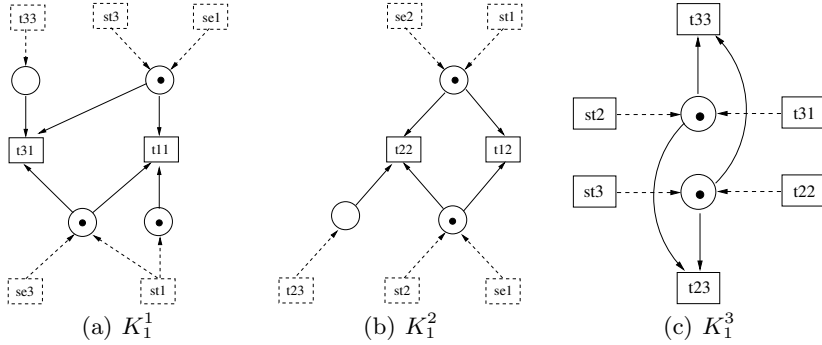


Fig. 5. Two ladder-shaped subgraphs leading to deadlock

Following Algorithm 1, to avoid reaching the deadlock states  $M_1$  and  $M_2$ , one can remove all seven instances of the transitions  $t_{31}$  and  $t_{22}$  indicated in Fig. 5. Let  $G_1$  be the reachable restriction of  $G$  defined in this way. Let  $G_1$  be the reachable restriction of  $G$  defined in this way.  $G_1$  has 23 states, none of which is a sink state. In particular 0, 2, 3, 12, 13, 15, 16, 19, 23, 26, 29 are states in  $G_1$  but 1 and 5 are no longer reachable. The states 2, 12, 19, 23 may be reached from the initial state 0 by firing in  $G_1$  the sequences of events  $t_{33} \cdot t_{31}, t_{23} \cdot t_{22}, t_{33}, t_{12}$ , respectively. All 14 transitions  $t_{33}$  or  $t_{22}$  shown in Fig. 5 can be blocked by distributed monitor places (which we computed using SYNETH [5]). The resulting DPND controller  $K_1$  has four components  $K_1^1, K_1^2, K_1^3, K_1^4$  to plug in the respective locations 1 to 4.  $K_1^4$  does nothing but send the other



**Fig. 6.** Local controllers for our example

components asynchronous signals informing them of the occurrences of the *start eating* and *start thinking* events  $se_j$  and  $st_j$ . The local controllers  $K_1^1$ ,  $K_1^2$ ,  $K_1^3$  are depicted in Fig. 6. For each controller, the asynchronous flow from other local controllers is indicated by dashed arrows.  $K_1^1$  and  $K_1^2$  send messages to  $K_1^3$  (indicating that  $t_{31}$  or  $t_{22}$  has occurred) and  $K_1^3$  sends messages to  $K_1^1$  and  $K_1^2$  (indicating that  $t_{33}$  or  $t_{23}$  has occurred) but  $K_1^1$  and  $K_1^2$  do not communicate directly. In the design of the distributed controller  $K_1$ , we have privileged the transitions  $t_{31}$  and  $t_{22}$ . As a result, in the initial state of  $RG(K_1)/G$ , philosopher  $\varphi_1$  can grab both forks  $f_1$  and  $f_2$  *in parallel* while philosophers  $\varphi_2$  and  $\varphi_3$  can only fight over fork  $f_3$ .

Similar controllers may be obtained by choosing  $t_{12}$  and  $t_{33}$ , or  $t_{23}$  and  $t_{11}$ , instead of  $t_{31}$  and  $t_{22}$ . Unfortunately, neither  $K_1$  nor its variations (arising from a focus on the different events just noted) can avoid starvation. In  $RG(K_1)/G$ , for each philosopher there actually exists a reachable state in which he may act fast enough to prevent the other two from ever eating! Finally, note that all places of the local controller nets  $K_1^1$ ,  $K_1^2$  and  $K_1^3$  stay bounded by 1 in all reachable states of  $RG(K_1)/G$ . Therefore, to transform  $K_1$  into an equivalent asynchronous message passing automaton *AMPA*, as indicated in section 3, it suffices to compute the bounded reachability graph  $RG_B(K_1)$  for the bound  $B = 1$ . The component automaton  $A_1$  of this *AMPA* operating at location 1 (and thus obtained from  $K_1^1$ ) has 16 states and 52 transitions. The large numbers of states and transitions may be explained by the protocol used to receive messages from  $K_1^3$  and  $K_1^4$ ; indeed, as this protocol is asynchronous, the 6 types of messages may be received at any time and in almost any order.

## 5.2 Another DPND Controller That Avoids Deadlocks

A quite different distributed controller  $K_2$  may be obtained by shifting the focus to the transitions  $t_{31}$  and  $t_{11}$ . By inspecting the reachability graph  $G$  of  $N$  again, one can locate the two subgraphs shown in Fig. 7, which contain all

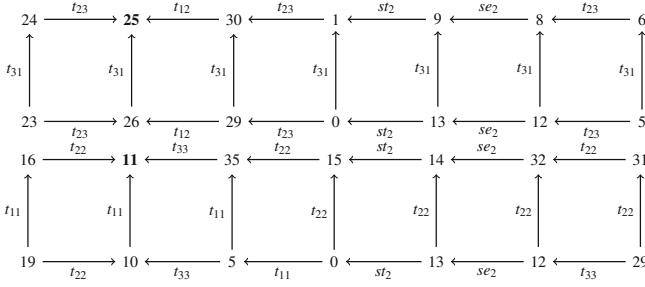


Fig. 7. Subgraphs leading to deadlock

predecessors of the deadlocking transitions  $26 \xrightarrow{t_{31}} 25$  and  $10 \xrightarrow{t_{11}} 11$ . To avoid reaching the deadlock states  $M_1$  and  $M_2$ , following Algorithm 1, one can also choose to cross off all seven instances of the transitions  $t_{31}$  and  $t_{11}$  indicated in Fig. 7. Let  $G_2$  be the reachable restriction of  $G$  defined in this way.  $G_2$  has 22 states, none of which is a sink state. In particular 0, 5, 10, 12, 13, 19, 23, 26, 29 are states in  $G_2$  but the sets of states  $\{6, 8, 9, 1, 30, 25, 24\}$  and  $\{31, 32, 14, 15, 35, 11, 16\}$  are no longer reachable. The states 5, 19, 23, 29 may be reached from the initial state 0 by firing in  $G_2$  the events  $t_{22}, t_{33}, t_{12}, t_{23}$ , respectively. All  $t_{31}$  or  $t_{11}$  transitions shown in Fig. 7 can be blocked by distributed monitor places. The resulting DPND controller  $K_2$  has four components  $K_2^1, K_2^2, K_2^3, K_2^4$  to plug in the respective locations 1 to 4.  $K_2^2, K_2^3$  and  $K_2^4$  do nothing but send  $K_2^1$  asynchronous signals informing this local controller of the occurrences of the sets of events  $\{t_{12}\}, \{t_{33}\}$  and  $\{se_1, st_1, se_3, st_3\}$ , respectively. The local controller  $K_2^1$  is depicted in Fig. 8. The component automaton  $A_1$  of the AMPA derived from  $K_2$  operating at location 1 (thus obtained from  $K_2^1$ ) has again 16 states and 52 transitions.

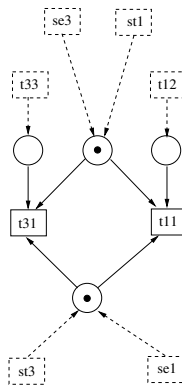


Fig. 8.  $K_2^1$

Now all control decisions are taken in location 1! In the initial state of  $RG(K_2)/G$ , philosopher  $\varphi_2$  can take both forks  $f_2$  and  $f_3$  *in parallel*, while philosophers  $\varphi_1$  and  $\varphi_3$  can only compete with  $\varphi_2$  to get forks  $f_2$  and  $f_3$ , respectively. Note that all places of the local controller nets  $K_2^1$  stay bounded by 1 in all reachable states of  $RG(K_2)/G$ . With respect to starvation,  $K_2$  and all similar controllers have exactly the same drawbacks as  $K_1$ .  $K_1$  and  $K_2$  are incomparable controllers, and we suspect, but have not verified, that they are both maximally permissive amongst DPND controllers for deadlock avoidance.

### 5.3 A Distributed Controller That Avoids Starvation

It is possible to produce a DPND controller that avoids both starvation and deadlocks without following Algorithm 1. We designed such a controller after a careful inspection of the cycles in the reachability graph of  $G$ , and confirmed that it could be implemented with a distributed Petri net. More specifically, we considered the set of cycles in  $G$  that go through the initial state 0 and contain each transition  $t_{ij}$  exactly once (for  $1 \leq i, j \leq 3$ ). We projected these cycles to words in  $\{t_{ij} \mid 1 \leq i, j \leq 3\}^6$ . We strived to extract from the resulting set of words maximal subsets  $\{w_1, \dots, w_n\}$  such that  $\{w_1, \dots, w_n\}^*$  is the language of a distributed Petri net. This worked for the set formed of the two words  $w_1 = t_{11} \cdot t_{23} \cdot t_{12} \cdot t_{31} \cdot t_{22} \cdot t_{33}$  and  $w_2 = t_{11} \cdot t_{33} \cdot t_{12} \cdot t_{22} \cdot t_{31} \cdot t_{23}$ . Coming back to  $G$ , we extracted from this graph the maximal subgraph  $G_3$  with language included in the inverse projection of  $\{w_1, w_2\}^*$ . This gave us the graph shown in Fig. 9.  $G_3$  is isomorphic to  $RG(K_3)/G$  where  $K_3$  is a distributed Petri net comprised of three components  $K_3^1$ ,  $K_3^2$  and  $K_3^3$ , depicted in Fig. 10. Note the presence of a flow arc with weight 2 in  $K_3^2$ . With this solution to the problem, parallelism disappears completely, and there remains only one state where a choice is possible. It would be interesting to examine the same problem for a larger number of philosophers, but fully-automated strategies are necessary for this purpose.

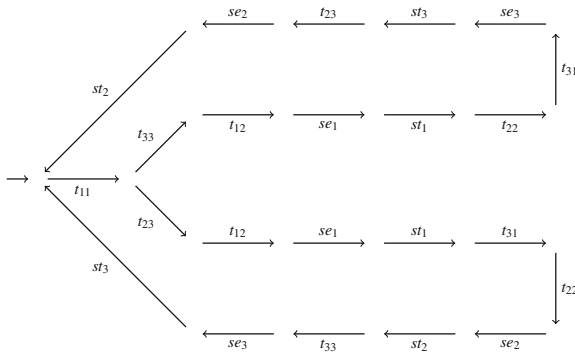


Fig. 9.  $RG(K_3)/G$

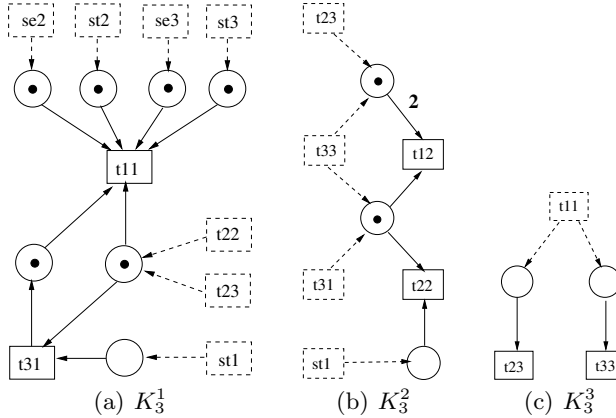


Fig. 10. A fair controller

## 6 Conclusion

To move towards a theory of distributed control of DES, we have proposed a mixture of asynchronous control and communication. A significant advantage of the proposed methodology is that the way of encoding the information to be exchanged is automated: the messages sent asynchronously are names of places of a Petri net produced by synthesis. Yet there remains a sizeable amount of work to be done: at present, a significant disadvantage of the methodology is the lack of a general theorem and a fully-automated controller synthesis algorithm (including heuristics to guide the non-deterministic choice of the transitions to be removed).

## References

1. Badouel, E., Darondeau, P.: Theory of Regions. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 529–586. Springer, Heidelberg (1998)
2. Badouel, E., Caillaud, B., Darondeau, P.: Distributing Finite Automata Through Petri Net Synthesis. *Formal Aspects of Computing* 13, 447–470 (2002)
3. Barrett, G., Lafortune, S.: Decentralized Supervisory Control with Communicating Controllers. *IEEE Trans. Autom. Control* 45(9), 1620–1638 (2000)
4. Brand, D., Zafropulo, P.: On Communicating Finite-State Machines. *J. of the ACM* 30(2), 323–342 (1983)
5. Caillaud, B.: <http://www.irisa.fr/s4/tools/synet/>
6. Chernikova, N.V.: Algorithm for finding a general formula for the non-negative solutions of a system of linear inequalities. *USSR Computational Mathematics and Mathematical Physics* 5(2), 228–233 (1965)
7. Cai, K., Wonham, W.M.: Supervisor Localization: A Top-Down Approach to Distributed Control of Discrete-Event Systems. *IEEE Trans. Autom. Control* 55(3), 605–618 (2010)



8. Darondeau, P.: Unbounded Petri Net Synthesis. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 413–438. Springer, Heidelberg (2004)
9. Darondeau, P.: Distributed Implementation of Ramadge-Wonham Supervisory Control with Petri Nets. In: CDC-ECC 2005, pp. 2107–2112 (2005)
10. Giua, A., Di Cesare, F., Silva, M.: Generalized Mutual Exclusion Constraints on Nets with Uncontrollable Transitions. In: IEEE-SMC 1992, pp. 974–979 (1992)
11. Genest, B., Gimbert, H., Muscholl, A., Walukiewicz, I.: Optimal Zielonka-Type Construction of Deterministic Asynchronous Automata. In: Abramsky, S., Gavoiile, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010, Part II. LNCS, vol. 6199, pp. 52–63. Springer, Heidelberg (2010)
12. Ghaffari, A., Rezg, N., Xie, X.: Algebraic and Geometric Characterization of Petri Net Controllers Using the Theory of Regions. In: WODES 2002, pp. 219–224 (2002)
13. Ghaffari, A., Rezg, N., Xie, X.: Design of Live and Maximally Permissive Petri Net Controller Using the Theory of Regions. *IEEE Trans. Robot. Autom.* 19, 137–142 (2003)
14. van Glabbeek, R.J., Weijland, W.P.: Branching Time and Abstraction in Bisimulation Semantics. *J. of the ACM* 43(3), 555–600 (1996)
15. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The Synchronous Dataflow Programming Language Lustre. *Proc. IEEE* 79(9), 1305–1320 (1991)
16. Hiraishi, K.: On Solvability of a Decentralized Supervisory Control Problem with Communication. *IEEE Trans. Autom. Control* 54(3), 468–480 (2009)
17. Hoare, C.A.R.: Communicating Sequential Processes. *CACM* 21(8), 666–677 (1978)
18. Kalyon, G., Le Gall, T., Marchand, H., Massart, T.: Synthesis of Communicating Controllers for Distributed Systems. In: CDC-ECC 2011, pp. 1803–1810 (2011)
19. Lamouchi, H., Thistle, J.: Effective Control Synthesis for DES under Partial Observations. In: CDC 2000, pp. 22–28 (2000)
20. Lamport, L.: Arbiter-Free Synchronization. *Distrib. Comput.* 16(2/3), 219–237 (2003)
21. Lin, F., Wonham, W.M.: On Observability of Discrete-Event Systems. *Info. Sci.* 44, 173–198 (1988)
22. Mannani, A., Gohari, P.: Decentralized Supervisory Control of Discrete-Event Systems Over Communication Networks. *IEEE Trans. Autom. Control* 53(2), 547–559 (2008)
23. Mukund, M., Sohoni, M.: Gossiping, Asynchronous Automata and Zielonka’s Theorem. Report TCS-94-2, Chennai Mathematical Institute (1994)
24. Madhusudan, P., Thiagarajan, P.S., Yang, S.: The MSO Theory of Connectedly Communicating Processes. In: Sarukkai, S., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 201–212. Springer, Heidelberg (2005)
25. Potop-Butucaru, D., Caillaud, B.: Correct-by-Construction Asynchronous Implementation of Modular Synchronous Specifications. In: ACSD 2005, pp. 48–57 (2005)
26. Ramadge, P.J., Wonham, W.M.: Supervisory Control of a Class of Discrete Event Processes. *SIAM J. Control Optim.* 25, 206–230 (1987)
27. Ramadge, P.J., Wonham, W.M.: The Control of Discrete Event Systems. *Proc. of the IEEE, Special Issue on Dynamics of Discrete Event Systems* 77, 81–98 (1989)
28. Ricker, S.L., Caillaud, B.: Mind the Gap: Expanding Communication Options in Decentralized Discrete-Event Control. In: CDC 2007, pp. 5924–5929 (2007)
29. Ricker, S.L.: Asymptotic Minimal Communication for Decentralized Discrete-Event Control. In: WODES 2008, pp. 486–491 (2008)

30. Rudie, K., Lafortune, S., Lin, F.: Minimal Communication in a Distributed Discrete-Event System. *IEEE Trans. Autom. Control* 48(6), 957–975 (2003)
31. Rudie, K., Wonham, W.M.: Think Globally, Act Locally: Decentralized Supervisory Control. *IEEE Trans. Autom. Control* 37(11), 1692–1708 (1992)
32. Thistle, J.G.: Undecidability in Decentralized Supervision. *Syst. Control Lett.* 54, 503–509 (2005)
33. Tripakis, S.: Decentralized Control of Discrete Event Systems with Bounded or Unbounded Delay Communication. *IEEE Trans. Autom. Control* 49(9), 1489–1501 (2004)
34. Wang, W., Lafortune, S., Lin, F.: Minimization of Communication of Event Occurrences in Acyclic Discrete-Event Systems. *IEEE Trans. Autom. Control* 53(9), 2197–2202 (2008)
35. Wang, Y., Lafortune, S., Kelly, T., Kudlur, M., Mahlke, S.: The Theory of Deadlock Avoidance via Discrete Control. In: *POPL 2009*, pp. 252–263 (2009)
36. Reisig, W.: *Elements of Distributed Algorithms*. Springer, Berlin (1998)
37. Xu, X., Kumar, R.: Distributed State Estimation in Discrete Event Systems. In: *ACC 2009*, pp. 4735–4740 (2009)
38. Yoo, T.S., Lafortune, S.: A General Architecture for Decentralized Supervisory Control of Discrete-Event Systems. *Discrete Event Dyn. Syst.* 12(3), 335–377 (2002)
39. Yamalidou, K., Moody, J., Lemmon, M., Antsaklis, P.: Feedback Control on Petri Nets Based on Place Invariants. *Automatica* 32(1), 15–28 (1996)
40. Zielonka, W.: Notes on Finite Asynchronous Automata. *RAIRO Informatique Théorique et Applications* 21, 99–135 (1987)

## Appendix

Our Asynchronous Message Passing Automata (AMPA) differ from the communicating automata originally introduced by Brand and Zafiropulo [4] in that communications are not FIFO. Given a finite set of locations  $\mathcal{L}$ , a *B-bounded AMPA* is a collection  $\{A_\ell \mid \ell \in \mathcal{L}\}$  of deterministic finite automata (DFA) together with a finite set of messages  $P$ , where  $\lambda : P \rightarrow \mathcal{L}$  specifies the address for each message. Each  $A_\ell = (Q_\ell, \Sigma_\ell \uplus \Sigma_\ell^! \uplus \Sigma_\ell^?, \delta_\ell, q_{0,\ell})$  is a DFA with a partial transition map  $\delta_\ell$ , and initial state  $q_{0,\ell}$ , where  $\Sigma_\ell^! = \{\ell!p \mid p \in P \wedge \lambda(p) \neq \ell\}$  and  $\Sigma_\ell^? = \{\ell?p \mid p \in P \wedge \lambda(p) = \ell\}$  are the sets of sending and receiving actions, respectively. The actions in  $\Sigma_\ell$  are observable. The communication actions in  $\Sigma_\ell^! \cup \Sigma_\ell^?$  are unobservable.

The dynamics of a *B-bounded AMPA* are defined by a transition system constructed inductively from an initial configuration  $\langle \overline{q_0}, \overline{m_0} \rangle$  as follows:

- $\overline{q_0}$  is an  $\mathcal{L}$ -indexed vector with entries  $q_{0,\ell}$  for all  $\ell \in \mathcal{L}$ ,
- $\overline{m_0}$  is an  $P$ -indexed vector with null entries for all  $p \in P$ ,
- From any configuration  $\langle \overline{q}, \overline{m} \rangle$ , where  $\overline{q}$  is an  $\mathcal{L}$ -indexed vector with entries  $q_\ell \in Q_\ell$  for all  $\ell \in \mathcal{L}$  and  $\overline{m}$  is a  $P$ -indexed vector of integers with entries  $m_p \geq 0$  for all  $p \in P$ , there is a transition  $\langle \overline{q}, \overline{m} \rangle \xrightarrow{\sigma} \langle \overline{q'}, \overline{m'} \rangle$  in the following three cases:
  - $\underline{q'_\ell} = \delta_\ell(q_\ell, \sigma)$  for some  $\ell \in \mathcal{L}$  and  $\sigma \in \Sigma_\ell$ ,  $q'_k = q_k$  for all  $k \neq \ell$  and  $m' = \overline{m}$ ;

- $q'_\ell = \delta_\ell(q_\ell, \sigma)$  for some  $\ell \in \mathcal{L}$  and  $\sigma = \ell!p \in \Sigma_\ell^!$ ,  $q'_k = q_k$  for all  $k \neq \ell$ ,  $m'_p = m_p + 1 \leq B$ , and  $m'_r = m_r$  for all  $r \neq p$ ;
- $q'_\ell = \delta_\ell(q_\ell, \sigma)$  for some  $\ell \in \mathcal{L}$  and  $\sigma = \ell?p \in \Sigma_\ell^?$ ,  $q'_k = q_k$  for all  $k \neq \ell$ ,  $m'_p = m_p - 1 \geq 0$ , and  $m'_r = m_r$  for all  $r \neq p$ ;

Branching bisimulation was defined by van Glabbeek and Weijland [14] for processes with a single unobservable action  $\tau$ . The following is an adaptation of the original definition to processes defined by automata with several unobservable actions. Let  $\Sigma = \Sigma_o \cup \Sigma_{uo}$  be a set of labels, where  $\Sigma_o$  and  $\Sigma_{uo}$  are the subsets of observable and unobservable labels, respectively. Let  $A = (Q, \Sigma, \delta, q_0)$  and  $A' = (Q', \Sigma, \delta, q'_0)$  be two automata over  $\Sigma$ .  $A$  and  $A'$  are *branching bisimilar* if there exists a symmetric relation  $R \in Q \times Q' \cup Q' \times Q$  such that  $(q_0, q'_0) \in R$  and for all  $(r, s) \in R$ :

- if  $\delta(r, \sigma) = r'$  and  $\sigma \in \Sigma_{uo}$ , then  $(r', s) \in R$ ;
- if  $\delta(r, \sigma) = r'$  and  $\sigma \in \Sigma_o$ , then there exists a sequence  $\sigma'_1 \dots \sigma'_k \in \Sigma_{uo}^*$  (where  $k = 0$  means an empty sequence) such that if one lets  $\delta(s, \sigma'_1 \dots \sigma'_j) = s'_j$  for  $j \leq k$ , and  $\delta(s'_k, \sigma) = s'$ , then  $s'$  and all states  $s'_j$  are effectively defined, and moreover  $(r', s') \in R$ .

# Extending PNML Scope: A Framework to Combine Petri Nets Types

Lom-Messan Hillah<sup>1</sup>, Fabrice Kordon<sup>2</sup>, Charles Lakos<sup>3</sup>, and Laure Petrucci<sup>4</sup>

<sup>1</sup> LIP6, CNRS UMR 7606 and Université Paris Ouest Nanterre La Défense  
200, avenue de la République, F-92001 Nanterre Cedex, France  
Lom-Messan.Hillah@lip6.fr

<sup>2</sup> LIP6 - CNRS UMR 7606, Université P. & M. Curie  
4 Place Jussieu, F-75252 Paris cedex 05, France  
Fabrice.Kordon@lip6.fr

<sup>3</sup> University of Adelaide, Adelaide, SA 5005, Australia  
Charles.Lakos@adelaide.edu.au

<sup>4</sup> LIPN, CNRS UMR 7030, Université Paris XIII  
99, avenue Jean-Baptiste Clément, F-93430 Villetaneuse, France  
Laure.Petrucci@lipn.univ-paris13.fr

**Abstract.** The Petri net standard ISO/IEC 15909 comprises 3 parts. The first one defines the most used net types, the second an interchange format for these – both are published. The third part deals with Petri net extensions, in particular structuring mechanisms and the introduction of additional, more elaborate net types within the standard.

This paper presents a contribution to elaborate an extension framework for the third part of the standard. This strategy aims at composing enabling rules and augmenting constraints in order to build new Petri net types. We show as a proof of concept how this can be achieved with priorities, times, inhibitor arcs in the context of an interleaving semantics. We then map this framework onto the current standard metamodels.

**Keywords:** Standardisation, PNML, Prioritised Petri Nets, Time Nets.

## 1 Introduction

**Context.** The International Standard on Petri nets, ISO/IEC 15909, comprises three parts. The first one (ISO/IEC 15909-1) deals with basic definitions of several Petri net types: Place/Transition, Symmetric, and High-level nets<sup>1</sup>. It was published in December 2004 [13]. The second part, ISO/IEC 15909-2, defines the interchange format for Petri net models: the Petri Net Markup Language [15] (PNML, an XML-based representation). This part of the standard was published on February 2011 [14]. It can now be used by tool developers in the Petri Nets community with, for example, the companion tool to the standard, PNML Framework [10].

---

<sup>1</sup> In this paper, the term “high-level net” is used in the sense of the standard and corresponds to coloured Petri nets as in Jensen’s work [16].

The standardisation effort is now focussed on the third part. ISO/IEC 15909-3 aims at defining extensions on the whole family of Petri nets. Extensions are, for instance, the support of modularity, time, priorities or probabilities. Enrichments consider less significant semantic changes such as inhibitor arcs, capacity places, etc. This raises flexibility and compatibility issues in the standard.

While parts 1 and 2 of the ISO/IEC 15909 standard address simple and common Petri nets types, part 3 is concerned with extensions. The work on these issues started with a one-year study group drawing conclusions with respect to the scope to be addressed. Then, according to the study group conclusions, the standardisation project was launched in May 2011, for delivery within 5 years.

**Contribution.** The choices to be made in part 3 of the standard must of course ensure compatibility with the previous parts. We propose to achieve this goal by using the notion of *orthogonality*. It allows us to build a *framework* to describe the behavioural semantics of nets in a compositional way. This is achieved by revisiting the firing rule of several well known Petri net types based on the enabling rule. The objective is to compose existing enabling rules with augmenting constraints in order to elaborate these Petri net types consistently. We apply it to P/T nets, Prioritised nets and Time nets.

Associated with this framework is a MDE-based one that allows us to compose pieces of metamodels corresponding to *enabling functions* and *augmenting constraints* (see section 3.2). This is an important link between the syntax-based way of handling semantic in MDE techniques and the formal definition of a behavioural semantics.

Based on this framework, we map the selected Petri nets types onto our MDE-based framework as a proof of concept, as planned for in future evolutions of the standard.

**Content.** Section 2 recalls some well-known Petri nets types. Then, section 3 presents the framework to define the behavioural semantics of nets and applies it to the net types already presented. Section 4 details the MDE-based framework associated with the standard. Section 5 maps the notions of the behavioral semantics defined in section 3 into this MDE-based framework to build metamodels of net types suitable for generating PNML descriptions, followed by a discussion in section 6.

## 2 Some Petri Nets Definitions

This section introduces the notations for different types of Petri Nets.

### 2.1 Definition of Place/Transition Nets

This section first introduces Place/Transitions nets.

#### **Definition 1 (Place/Transition Net).**

A Place/Transition Net (*P/T net*) is defined by a tuple  $N = \langle P, T, Pre, Post, M_0 \rangle$ , where:

- $P$  is a finite set (the set of places of  $N$ ),
- $T$  is a finite set (the set of transitions of  $N$ ), disjoint from  $P$ ,

- $Pre, Post \in \mathbb{N}^{|P| \times |T|}$  are matrices (the backward and forward incidence matrices),
- $M_0$ , a vector in  $\mathbb{N}^{|P|}$  defining the initial number of tokens in places.

We now introduce  $M(p)$ ,  $\bullet t$ , and  $t\bullet$  that are respectively:

- the current marking of place  $p$ ,
- the subset of places which constitute the precondition of a transition  $t \in T$ ,
- the subset of places which constitute the postcondition of a transition  $t \in T$ .

From these notations, we can define the enabling and firing rules for P/T nets as follows.

**Definition 2 (P/T Net enabling rule).**

A transition  $t \in T$  is enabled in marking  $M$ , denoted by  $M[t]$ , iff:  $\forall p \in \bullet t, M(p) \geq Pre(p, t)$ .

**Definition 3 (P/T Net firing rule).**

If a transition  $t \in T$  is enabled in marking  $M$ , it can fire leading to marking  $M'$ , denoted by  $M[t]M'$ , where:  $\forall p \in P, M'(p) = M(p) - Pre(p, t) + Post(p, t)$ .

An *inhibitor arc* is a special kind of arc that reverses the logic of an input place. Instead of testing the presence of a minimum number of tokens in the related place, it tests the lack of tokens.

**Definition 4 (Petri Nets with Inhibitor Arcs).**

A Petri net with inhibitor arcs is a Petri Net  $N$  together with a matrix  $I \in \mathbb{N}^{|P| \times |T|}$  of inhibitor arcs.

**Definition 5 (P/T Nets with inhibitor arcs enabling rule).**

A transition  $t \in T$  is enabled in marking  $M$ , denoted by  $M[t]$ , iff:  $\forall p \in \bullet t, (M(p) \geq Pre(p, t)) \wedge (M(p) \leq I(p, t))$ .

Then, the firing rule is identical to the one for P/T nets, provided the transition is enabled.

## 2.2 Definition of Prioritised Petri Nets

This section introduces the definition of prioritised Petri nets.

**Definition 6 (Statically Prioritised Petri net).**

A Statically Prioritised Petri net is a tuple  $SPPN = \langle P, T, Pre, Post, M_0, \rho \rangle$ , where:

- $\langle P, T, Pre, Post, M_0 \rangle$  is a P/T net.
- $\rho$  is the static priority function mapping a transition into  $\mathbb{R}^+$ .

We can also consider the case where the priority of transitions is *dynamic*, i.e. it depends on the current marking [1]. This definition was introduced in [18]. Note that the only difference with statically prioritised Petri nets concerns the priority function  $\rho$ .

**Definition 7 (Prioritised Petri net).**

A Prioritised Petri net is a tuple  $PPN = \langle P, T, Pre, Post, M_0, \rho \rangle$ , where:

- $\langle P, T, Pre, Post, M_0 \rangle$  is a P/T net.
- $\rho$  is the priority function mapping a marking and a transition into  $\mathbb{R}^+$ .

The behaviour of a prioritised Petri net is now detailed, markings being those of the underlying Petri net. Note that the firing rule is the same as for non-prioritised Petri nets, the priority scheme influencing only the enabling condition.

**Definition 8 (Prioritised enabling rule).**

A transition  $t \in T$  is priority enabled in marking  $M$ , denoted by  $M[t]^\rho$ , iff:

- it is enabled, i.e.  $M[t]$ , and
- no transition of higher priority is enabled, i.e.  $\forall t' : M[t'] \Rightarrow \rho(M, t) \geq \rho(M, t')$ .

The definition of the priority function  $\rho$  is extended to sets and sequences of transitions (and even markings  $M$ ):

- $\forall X \subseteq T : \rho(M, X) = \max \{ \rho(M, t) \mid t \in X \wedge M[t] \}$
- $\forall \sigma \in T^* : \rho(M, \sigma) = \min \{ \rho(M', t') \mid M'[t']^\rho \text{ occurs in } M[\sigma]^\rho \}$ .

For static prioritised nets where  $\rho(M, t)$  is a constant function associated with  $t$  and for dynamic prioritised nets, for at least one transition,  $\rho(M, t)$  depends on the current marking. For now, we will consider only dynamically prioritised nets since static ones are encompassed by these.

If the priority function is constantly zero over all markings and all transitions, then the behaviour of a Prioritised Petri Net is isomorphic to that of the underlying P/T Net.

Note that we choose to define priority as a positive real-valued function over markings and transitions — the higher the value, the greater the priority. We could equally define priority in terms of a *rank function* which maps markings and transitions to positive real values, but where the smaller value has the higher priority. This would be appropriate, for example, if the rank were an indication of earliest firing time.

### 2.3 Definition of Petri Nets with Time

*Time Petri nets* (TPN) are Petri nets where timing constraints are associated with the nodes or arcs. Timing constraints are given as time intervals. This section briefly presents the definition of TPNs and their semantics [3], then introduces the model we will be focusing on, which is *Time Petri Nets* [4], where the timing constraints are associated with transitions.

**Definition 9 (Generic Time Petri net).**

A Generic Time Petri net is a tuple  $\langle P, T, Pre, Post, S_0, I \rangle$  such that:

- $\langle P, T, Pre, Post, S_0 \rangle$  is a (marked) P/T net ( $S_0$  denotes its initial state, i.e. marking + clocks);
- $I : X \rightarrow I(\mathbb{R}_+)$  is a mapping from  $X \in \{P, T, P \times T \cup T \times P\}$  to the set  $I(\mathbb{R}_+)$  of intervals. These intervals have real bounds or are right-open to infinity.

**Semantics.** The semantics of TPNs is based on the notion of *clocks*. One or more clocks can be associated with a time interval and the value of all clocks progress synchronously as time elapses. The firability of enabled transitions depends on having the value of the related clocks in their associated intervals. A clock may be reset upon meeting a condition on the marking of the net, usually after the firing of a transition.

The semantics of TPNs is defined in terms of:

- *Reset policy*: the value of a clock is reset upon firing some transition. It is the only way to decrease its value. But it is also meaningful not to reset a clock.
- *Strong firing policy*: in a *strong* semantics, when the upper bound of the interval associated with a clock is reached, transitions must fire instantaneously, until the clock is reset. The clock can go beyond the upper bound of the interval, if there is no possible instantaneous sequence of firings, in which case dead tokens are usually generated. This generally models a bad behaviour, since tokens become too old to satisfy the timing constraints.
- *Weak firing policy*: in this case, the clock leaving the interval prevents the associated firings from taking place. Dead tokens may also be generated, but this time they are considered part of the normal behaviour of the net.
- *Monoserver setting*: each interval only has one associated clock, which usually denotes a single task processing.
- *Multiserver setting*: each interval has more than one associated clock, which usually denotes the handling of several similar tasks. In this setting, each clock evolves independently of the others.

**Time Petri Nets.** We consider in this paper the Time Petri Net model [4], where time intervals are associated with transitions. The semantics is strong and monoserver. Time Petri Nets are appropriate for modelling real-time systems. More formally:

**Definition 10 (Time Petri net).** A Time Petri Net is a tuple  $\langle P, T, Pre, Post, S_0, \alpha, \beta \rangle$  where:

- $\langle P, T, Pre, Post, S_0 \rangle$  is a (marked) P/T net.
- $\alpha : T \mapsto \mathbb{Q}_+$  and  $\beta : T \mapsto \mathbb{Q}_+ \cup \{\infty\}$  are functions satisfying  $\forall t \in T, \alpha(t) \leq \beta(t)$  called respectively earliest ( $\alpha$ ) and latest ( $\beta$ ) transition firing times.

Functions  $\alpha$  and  $\beta$  are the instantiation of  $I$  in Generic Time Petri nets (see definition 9) for Time Petri nets.

Given a marking  $M$ , we write  $En(M) = \{t \in T \mid M[t]\}$  for the set of transitions enabled in  $M$ . A clock is implicitly associated with each transition and a state of the system is a pair  $(M, \nu)$ , where  $M$  is a marking and  $\nu \in \mathbb{R}_+^{En(M)}$  is a mapping associating a clock value with each transition enabled in  $M$ .

We now define the enabling rule and firing rule of Time Petri nets.

**Definition 11 (Time Petri net enabling rule).**

A transition  $t \in T$  is time enabled in state  $(M, \nu)$ , denoted by  $(M, \nu)[t]$ , if:

- it is enabled, i.e.  $M[t]$ , and
- $\nu(t) \in [\alpha(t), \beta(t)]$ .



**Definition 12 (Time Petri net firing rule).**

From a state  $(M, v)$ , two types of transitions are possible:

- if transition  $t \in T$  is time enabled in state  $(M, v)$ , firing  $t$  leads to state  $(M', v')$ , denoted by  $(M, v)[t](M', v')$ , where:
  - $\forall p \in P, M'(p) = M(p) - Pre(p, t) + Post(p, t)$ , as usual,
  - $\forall t' \in En(M'), v'(t') = 0$  if  $t'$  is newly enabled (explained below), and  $v(t')$  otherwise.
- if  $\forall t \in En(M), v(t) + d \leq \beta(t)$ , time elapsing by delay  $d \in \mathbb{R}_+$  leads to state  $(M, v')$ , where  $v'(t) = v(t) + d$  for all  $t \in En(M)$ .

Various definitions have been proposed for newly enabling of a transition [2][26]. A common one, called intermediate semantics, states that transition  $t'$  is newly enabled by the firing of  $t$  if:

- $t'$  belongs to  $En(M')$  and
- either  $t' = t$  or  $t'$  is not enabled in  $M - Pre(., t)$  (where  $Pre(., t)$  denotes the vector  $(Pre(p, t))_{p \in P}$ ).

### 3 An Engineering Approach to Extension and Composition

In considering extensions to the common Petri net types in parts 1 and 2 of the ISO/IEC 15909 standard, we wish to capture the extensions so that they are as flexible as possible, and hence applicable to multiple Petri net types. This is an engineering challenge like any software design — its success will be measured by a number of non-functional properties like extensibility, maintainability, usability and reusability.

In aiming for this goal, we wish to define extensions as “pieces of semantics” that are *orthogonal*. The term *orthogonal* has been applied in the literature to language design in a variety of ways depending on the context. We first review some of the literature on this notion before indicating how we propose to apply it in the standard.

#### 3.1 The Notion of Orthogonality

**Orthogonality for Programming Languages.** In the context of programming language design, Pratt and Zelkowitz put it this way [24]: “*The term orthogonality refers to the attribute of being able to combine various features of a language in all possible combinations, with every combination being meaningful. [...] When the features of a language are orthogonal, then the language is easier to learn and programs are easier to write because there are fewer exceptions and special cases to remember.*”

IBM [12] defines orthogonality with respect to extensions on top of a base language as follows: “*An orthogonal extension is a feature that is added on top of a base without altering the behaviour of the existing language features. A valid program conforming to a base level will continue to compile and run correctly with such extensions. The program will still be valid, and its behaviour will remain unchanged in the presence of the orthogonal extensions. Such an extension is therefore consistent with the corresponding*

base standard level. Invalid programs may behave differently at execution time and in the diagnostics issued by the compiler.

On the other hand, a non-orthogonal extension is one that can change the semantics of existing constructs or can introduce syntax conflicting with the base. A valid program conforming to the base is not guaranteed to compile and run correctly with the non-orthogonal extensions.”

Palsberg and Schwartzbach [23] argue that for object-oriented languages, class substitution is preferred to generic classes because it better complements inheritance as a mechanism for generating new classes:

- Just like with inheritance, class substitution can be used (repeatedly) to build new (sub-)classes (p. 147)
- Class substitution is orthogonal to inheritance (p. 147), which is made precise as follows (p. 152):
  - if a class D can be obtained from C by inheritance, then D cannot be obtained from C by class substitution; and
  - if a class D can be obtained from C by class substitution, then D cannot be obtained from C by inheritance.

The common thread in the above references is that *orthogonality* of language features embodies both *independence* and *consistent composition*. As examples we might note the following:

- “The length of time data is kept in storage in a computer system is known as its persistence. Orthogonal persistence is the quality of a programming system that allows a programmer to treat data similarly without regard to the length of time the data is kept in storage.” [27]
- C has two kinds of built-in data structures, arrays and records (structs). It is not orthogonal for records to be able to be returned from functions, but arrays cannot. [8]
- In C,  $a + b$  usually means that they are added, unless  $a$  is a pointer [in which case] the value of  $b$  may be changed before the addition takes place. [8]

Similar concerns are raised by Szyperski in his study of component software. He identifies an important paradigm of *independent extensibility* [29]. The essential property is that independently developed extensions can be combined (p. 84). He notes that traditional class frameworks are specialised at application construction time and thereafter disappear as no longer separable parts of the generated application. He argues for independently extensible systems to specify clearly *what* can be extended — each one of these is then referred to as a *dimension of (independent) extensibility*. These dimensions may not be orthogonal, e.g. extensions to support object serialisation will overlap extensions to support persistence.

**Orthogonality for Concurrent Systems.** The term *orthogonality* has also been applied in other contexts, and these uses are especially pertinent to our concerns with a formalism that embodies concurrency.

In the context of the Unix shell, Raymond understands *orthogonality* to mean side-effect free [25]. In the context of extensions to the Unix C-shell, Pahl argues that

“language extension is presented as a refinement process. [...] The property we want to preserve during the refinement process is behaviour, also called safety refinement elsewhere. [...] Behaviour preservation is in particular important since it guarantees orthogonality of the new feature with the basic language.” [22]. In the context of state charts, orthogonality is presented as a form of *conceptual concurrency* which is captured as *AND-decomposition* [9].

### 3.2 Application to Petri Net Extensions

It is our intention to apply the above experience from language design to the formulation of extensions to the base Petri net formalisms, so that we arrive at a set of *orthogonal* extensions. As implied by the above discussion, this is an engineering or aesthetic goal rather than a theoretical one, and its success will be measured by a number of non-functional properties, including the number of extensions that can be accommodated before the metamodel architecture needs to be refactored.

Firstly, we recall that where orthogonality is defined with respect to extensions on top of a base language it was stated that: “An orthogonal extension is a feature that is added on top of a base without altering the behaviour of the existing language features.” In this regard, if we were considering a step semantics, we would follow the example of Christensen and Hansen who observed that for inhibitor or threshold arcs, any upper bound on a place marking ought to take into account the tokens added by the step [5]. This would be required for a step semantics if the diamond rule is to hold for concurrently enabled transitions.

Secondly, we note that orthogonality embodies both independence and consistent composition of the language elements. In our subsequent discussion, this applies to the addition of inhibitor arcs and prioritised transitions. These extensions involve disjoint attributes and therefore do not interfere with each other and can be applied in any order.

Thirdly, we note that in the context of concurrent systems and specifically the Unix shell, orthogonality has been understood as requiring extensions to be side-effect free. For this reason, we do not currently contemplate extensions like that of Reference nets as implemented in *Renew* (the Reference Nets Workshop [17]), where transitions can be annotated with arbitrary (Java) code segments which then prompts warnings in the user guide of the pitfalls of side effects.

Fourthly, in line with Pahl’s approach to the Unix C-shell, we propose to capture Petri net extensions in terms of behaviour-preserving refinement. In line with earlier work [30,19], the extension of a base net  $N$  to an extended version  $N'$  is defined as a morphism:  $\phi : N' \rightarrow N$ . A morphism respects structure and behaviour and thus the components of one object are mapped by the morphism to their counterparts in the other. It is more usual to consider the morphism as mapping the extended form to the base form. Thus, for example, the (possibly) extended or embellished set of transitions is mapped to the simpler set, rather than vice versa. Where the extension introduces new attributes and constructs then  $\phi$  will essentially be a restriction mapping which ignores the additional components. Where the extension modifies existing attributes and constructs then  $\phi$  will

essentially be a projection mapping. Note that we do not here consider the forms of refinement appropriate to node refinement, where a node is refined by a subnet.

For example, we might take P/T nets as our base Petri nets. If we extended these with inhibitor arcs, then, in mapping from the extended to the base form  $\phi$  would ignore those arcs. On the other hand, if we extended the base Petri nets so that all tokens included a time attribute, then  $\phi$  would project out that additional attribute.

In characterising Petri net extensions, we focus on the firing rule introduced above and specifically the boolean enabling function:  $E : \mathcal{N} \times T \rightarrow \mathbb{B}$ , where  $\mathcal{N}$  is the set of Petri nets. In fact, it is more convenient to work with an extended version in terms of steps,  $Y$ , (i.e. sets of transitions), giving  $E : \mathcal{N} \times \mathbb{P}(T) \rightarrow \mathbb{B}$ <sup>2</sup>. We will also need to refer to the firing rule, which we characterise as a mapping from states to states:  $[ \ ] : S \rightarrow S$ . Note that we use the more general term *state* in preference to *marking* because we envisage that some extensions may introduce additional state components, such as a global clock.

Our primary requirement for orthogonal extensions is that extensions maintain behavioural consistency with the base formalism:

1.  $\phi(E'(N', Y')) \Rightarrow \phi(E')(\phi(N'), \phi(Y'))$   
In words: if the enabling condition holds in the extended net, then the corresponding (abstracted) enabling condition holds in the base net.
2.  $S'_1[Y']S'_2 \Rightarrow \phi(S'_1)[\phi(Y')]\phi(S'_2)$   
In words: if the step  $Y'$  causes a change of state from  $S'_1$  to  $S'_2$  in the extended system, then the corresponding step  $\phi(Y')$  effects the corresponding change of state from  $\phi(S'_1)$  to  $\phi(S'_2)$  in the base system. Note that if  $\phi(Y')$  is null (because the step is part of the additional components and thus ignored by  $\phi$ ), then this should have no effect on the base system state, i.e.  $\phi(S'_1) = \phi(S'_2)$ .

For simplicity, we prefer to work with a more constrained version of the first condition:  $E'(N', Y') = \phi(E')(\phi(N'), \phi(Y')) \wedge E''(N', Y')$ , where  $E''$  supplies an *augmenting constraint* in addition to the enabling rule in the base system<sup>3</sup>. Of course, there is no guarantee that this approach will always be applicable, but where it is, the commutativity and associativity of the conjunction operation will facilitate the orthogonality of extensions which are disjoint (as noted above for inhibitor arcs and prioritised nets).

The essential element of the extension mechanism contemplated above is that it constitutes a form of refinement that maintains behavioural consistency, in line with Pahl<sup>4</sup>. He argues that “*Behaviour preservation is in particular important since it guarantees orthogonality of the new feature with the basic language*”.

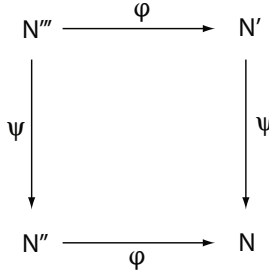
We extend this to independence of multiple extensions by requiring that they can be applied in any order and the result is the same. This corresponds to requiring that

<sup>2</sup> While restricting our attention to an interleaving semantics, the use of steps is desirable, especially when the extension introduces additional kinds of transitions or actions (such as the elapse of time). Then the morphism will be a restriction which ignores those additional transitions and the behaviour corresponding to these transitions in the base system will be null. Thus, our steps will typically be singleton or empty sets.

<sup>3</sup> This has been motivated by the use of conjunction for refining class invariants and pre- and post-conditions in Eiffel [20].

<sup>4</sup> Quoted in section 3.1

for two morphisms,  $\phi$  and  $\psi$ , the square of Fig. 1 commutes. As an example of this, we might consider the combination of both priorities and time to simple P/T nets. The interesting question is how the elapse of time would work in such a system — should time be allowed to elapse when no priority-enabled transition becomes disabled, or when no enabled transition becomes disabled. The former choice would contradict the requirement that we should be able to add the refinements in either order.



**Fig. 1.** State space for simplified Petri net for device message generation

Against the general background of orthogonality considered in section 3.1, this approach also has the following properties:

- In line with Raymond, our extensions are side-effect free, in the sense that the only way for the extended system to affect the state of the base system is for the action of the extended system to map to an appropriate action in the base system. Thus, if a different kind of action is introduced, e.g. a procedure call, then it cannot affect the underlying marking.
- In line with IBM’s definition of orthogonality, an orthogonal extension is a feature added on top of a base without altering the behaviour of the existing language features. This is especially clear if we require extensions to have an *identity* element, e.g. a prioritised net where all priorities are the same, or a timed net where the timing conditions never constrain the firing of the transition.
- The requirement by Pratt and Zelkowitz that an orthogonal language feature can be added to all other (relevant) existing constructs is *not* addressed by our proposal above — it is a matter for the language designer (or in this case, the designer of the Petri net extension). As in the case of the Unix shell, an orthogonal extension is side-effect free.

**Petri Nets Firing Rule Revisited.** In the context of Petri net extensions, it is our intention to ensure that extensions are side-effect free and that they will be formalised as behaviour-preserving refinements. To do so, let us revisit the definition of a firing rule in the context of the interleaving semantics [6]:

1. Computation of  $T' = \cup\{t_i\}$ , the set of enabled transitions. In other words  $T'$  is the subset of  $T$  for which  $E(N \in \mathcal{N}, t_i \in T_N) = true$ .  $E$  is the enabling function

$E : \mathcal{N} \times T \rightarrow \mathbb{B}$ , which has two parameters — a net within a net type  $N \in \mathcal{N} = \langle P, T, Pre, Post, S \rangle$  and the transition  $t \in T_N$  to which it applies.  $E$  is defined as follows:

$$E(N \in \mathcal{N}, t \in T_N) : \begin{cases} \text{True} & \text{when } t \text{ is firable} \\ \text{False} & \text{otherwise} \end{cases}$$

Note that  $S$  of  $N \in \mathcal{N}$  denotes the current state of net  $N$ . For a P/T net or a Prioritised Net, it is simply the current marking  $M$ , while for Time Petri Nets, it corresponds to  $(M, v)$  as defined in section 2.3

Thus firability of a given transition  $t \in T_N$  can be checked. This is also the case in later definitions of enabling rules.

2. Selection of one  $t \in T'$  to be fired, or some action like the elapse of time as in the case of Time Petri Nets;
3. Update of the state of  $N$ .

Note that Time Petri Nets allow models to evolve by firing a transition or by having time elapse. Step 2 of our firing rule caters for such alternative actions by insisting that one action be chosen at each step. Definition 12 ensures that the advance of time does not disable any already enabled transitions. In this way, we eliminate the possibility of side effects when there are two concurrent ways for the model to evolve. Thus, at this stage, the absence of side effects between enabling conditions appears to be a sufficient requirement to fit within our framework.

In the following section, we revisit in a compositional way some firing rules of well known types of Petri nets and associated features (inhibitor arcs, time and priorities management). Then, we compose them to build more elaborate Petri net types.

### 3.3 Revisiting Basic Enabling Functions and Augmenting Constraints

We now present the enabling functions for P/T nets and their augmenting constraints for inhibitor arcs, time (in the sense of [4]) and priorities (in the sense of [18]) in the framework we have developed above.

**Enabling Rule for P/T Nets.** Let us define  $E_{pt}(N, t)$  that returns true when the marking of input places is sufficient:

$$E_{pt}(N \in \mathcal{N}, t \in T_N) : \begin{cases} \text{True} & \text{iff } \forall p \in \bullet t, M(p) \geq Pre(p, t) \\ \text{False} & \text{otherwise} \end{cases} \quad (1)$$

**Enabling Rule for Priorities.** Let us define the augmenting constraint  $E_p(N, t)$  that returns true when  $prio(t)$  has of the lowest value over the net:

$$E_p(N \in \mathcal{N}, t \in T_N) : \begin{cases} \text{True} & \text{iff } \forall t', \rho(M, t) \leq \rho(M, t') \\ \text{False} & \text{otherwise} \end{cases} \quad (2)$$

The enabling condition for P/T nets with priorities is thus:

$$E_{np}(N, t) = E_{pt}(N, t) \wedge E_p(N, t) \quad (3)$$

**Enabling Rule for Time Conditions.** Let us define the augmenting constraint  $E_{tt}(N, t)$  that returns true when the local  $v(t)$  associated with  $t$  is in the range  $[\alpha, \beta]$  (constants associated with  $t$ ).

$$E_{tt}(N \in \mathcal{N}, t \in T_N) : \begin{cases} \text{True} & \text{iff } v(t) \geq \alpha(t) \wedge v(t) \leq \beta(t) \\ \text{False} & \text{otherwise} \end{cases} \quad (4)$$

The enabling condition for P/T nets with timing constraints is thus:

$$E_{nt}(N, t) = E_{pt}(N, t) \wedge E_{tt}(N, t) \quad (5)$$

Note that equation 5 is consistent with definition 11 (enabling function) assuming that the generic firing rule still allows the action corresponding to the elapse of time.

**Augmenting Constraint for Inhibitor Arcs.** Let us define the augmenting constraint  $E_i(N \in \mathcal{N}, t \in T_N)$  that returns true when there are less tokens than the value specified on the inhibitor arc ( $Pre_i(p, t) \neq 0$ ).

$$E_i(N \in \mathcal{N}, t \in T_N) : \begin{cases} \text{True} & \text{iff } \forall p \in \bullet t \text{ s.t. } Pre_i(p, t) > 0 : Pre_i(p, t) > M(p) \\ \text{False} & \text{otherwise} \end{cases} \quad (6)$$

**Composing Inhibitor Arcs with Defined Net Types.** We can now combine the definition of the inhibitor arc augmenting constraint with the net types we already defined, to build P/T nets with inhibitor arcs (equation 7), Prioritised nets with inhibitor arcs (equation 8), Time nets with inhibitor arcs (equation 9), Time nets with inhibitor arcs and priorities (equation 10). More combinations can be elaborated.

$$E_{pti}(N, t) = E_{pt}(N, t) \wedge E_i(N, t) \quad (7)$$

$$E_{npti}(N, t) = E_{np}(N, t) \wedge E_i(N, t) = E_{pt}(N, t) \wedge E_p(N, t) \wedge E_i(N, t) \quad (8)$$

$$E_{nti}(N, t) = E_{nt}(N, t) \wedge E_i(N, t) = E_{pt}(N, t) \wedge E_{tt}(N, t) \wedge E_i(N, t) \quad (9)$$

$$E_{npti}(N, t) = E_{nti}(N, t) \wedge E_p(N, t) = E_{pt}(N, t) \wedge E_{tt}(N, t) \wedge E_i(N, t) \wedge E_p(N, t) \quad (10)$$

**Discussion.** Now we have established a framework that encompasses the formal definitions of several types of Petri nets. Only interleaving semantics is considered so far. Moreover, if P/T nets constitute our base Petri nets, the framework could work with Symmetric Nets or High-Level Nets as well since the enabling functions defined in this section refer to the notion of state for which only the structure of the marking is impacted by colours. We do not detail this more due to space limitation.

In the next section, we present the MDE-based framework to compose Petri net types in a similar way to the formal framework presented here. Then section 5 shows how prioritised and time nets can be elaborated in PNML thanks to the MDE-based framework.

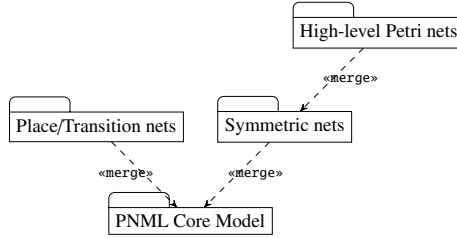


Fig. 2. Metamodels architecture currently defined in ISO/IEC-15909 Part 2

## 4 A MDE Framework to Extend and Compose Petri Net Types

As highlighted in the previous section, orthogonality of augmenting constraints as well as the semantic compatibility of firing rules are crucial for the formal definitions to work properly. At the syntactic level, upward compatibility is important as well. We consider upward compatibility as the ability to extract a base net type from its extended version.

We show in this section how the formal concepts developed earlier could be projected onto the metamodel architecture of the standard, thus yielding a model-based framework to extend and compose Petri Net types.

### 4.1 Current Metamodels Architecture

Figure 2 shows an overview of the metamodels architecture currently defined in Part 2 of the standard [11]. This architecture features three main Petri net types: P/T, Symmetric and High-level Petri nets. They rely on the common foundation offered by the PNML Core Model. It provides the structural definition of all Petri nets, which consists of nodes and arcs and an abstract definition of their labels. There is no restriction on labels since the PNML Core Model is not a concrete Petri net type.

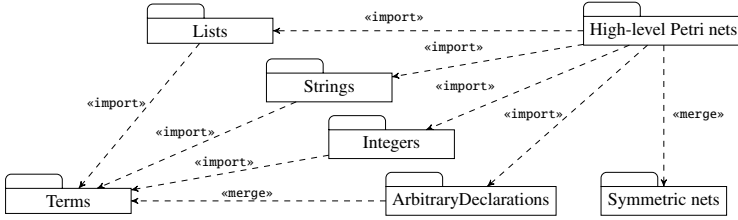
Such a modular architecture favours reuse between net types. Reuse takes two forms in the architectural pattern of the standard: `import` and `merge` package relationships, as defined in the UML 2 standard [21].

`Import` is meant to use an element from another namespace (package) without the need to fully qualify it. For example, when package A includes: `import B.b`, then in A we can directly refer to `b` without saying `B.b`. But `b` still belongs to the namespace B. In the ISO/IEC 15909-2 standard, Symmetric nets import sorts packages such as Finite Enumerations, Cyclic Enumerations, Booleans, etc.

`Merge` is meant to combine similar elements from the merged namespace to the merging one. For example, let us assume that `A.a`, `B.a` and `B.b` are defined. If B is merged into A (B being the target of the relationship), it will result in a new package `A'`:

- all elements of B now explicitly belong to `A'` (e.g., `A'.b`);
- `A.a` and `B.a` are merged into a single `A'.a` which combines the characteristics of both;
- actually, since A is the merging package (or the receiving package), A becomes `A'` (in the model, it is still named A).





**Fig. 3.** Modular construction of High-level Petri Nets based on Symmetric Nets

**Merge** is useful for incremental definitions (extensions) of the same concept for different purposes.

In the standard, this form of reuse is implemented for instance by defining P/T nets upon the Core Model and High-level nets upon Symmetric nets, as depicted in Figure 2. Therefore, Symmetric net elements and annotations are also valid in High-Level Petri nets (but not considered as Symmetric nets namespace elements anymore).

This extensible architecture is compatible with further new net type definitions, as well as with orthogonal extensions shared by different net types. These two extension schemes are put into practice for defining Symmetric nets and High-level nets as discussed in the next section. Prioritised Petri nets, presented in sections 2.2 and 5.1 are also defined using the same extension schemes.

## 4.2 Standard Nets Types Modular Definition

With the two forms of reuse, namely *import* and *merge*, Symmetric Nets are defined in the standard upon the PNML Core Model and High-level Petri Nets are defined upon the Symmetric Nets.

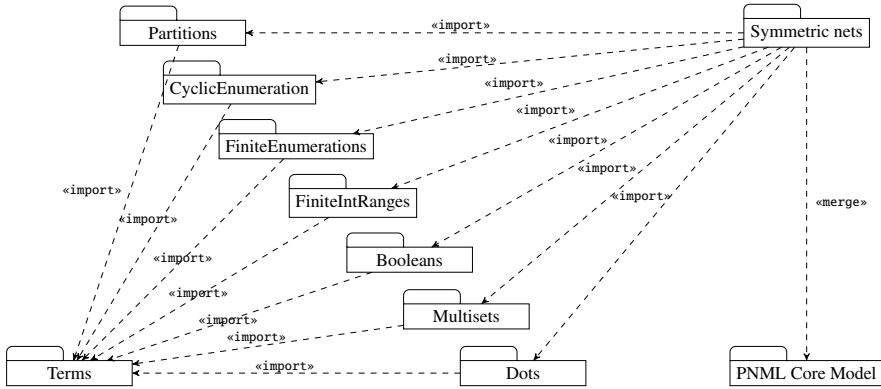
Figure 3 shows how High-level Petri Nets are defined. The *merge* relationship enables the reuse of the common foundation provided by Symmetric Nets. The provided concepts are now fully part of the High-level Petri Nets namespace. Then, with the *import* relationship, sorts specific to High-level Petri Nets (Lists, Strings, Integers and Arbitrary Declarations) are integrated to build this new type.

The definition of Symmetric Nets follows the same modular approach (see Figure 4), where the package of Symmetric Nets merges the PNML Core Model and imports the allowed sorts, the carrier sets of which are finite. The **Terms** package provides the abstract syntax to build algebraic expressions denoting the net declarations, place markings, arc annotations and transition guards.

## 4.3 Extended Metamodels Architecture Framework

From the common layout of the standards metamodels, we can extract an architectural pattern which could be used to extend them in order to build new net types.

Figure 5(a) describes such a pattern. The new net type **XX Extension YY Petri Net** is built upon an existing Petri net type, which is represented by **YY Petri net** and an extension (e.g. priority, time), which is represented by **XX Extension**. The new net



**Fig. 4.** Modular construction of Symmetric Nets based on PNML Core Model

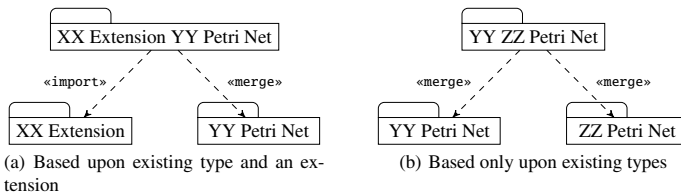
type package imports the extension package, while it merges the existing net one. This is consistent with the semantics of reuse in the standard architectural pattern, through the `import` and `merge` relationships.

Note that combining independent extensions boils down to performing a conjunct of several firing rules, which is in essence commutative. Thus, in that case, applying `XX` and `YY` extensions can be done in either order. On another hand, if one extension is further extended, then these two extensions are no longer independent, and cannot be applied in a different order.

Figure 5(b) describes the case where the new type is built upon existing net types only, which have already embedded their own extensions. The observed pattern is then composed only of the merged ones. This means all building blocks needed to build the new net type package come from the ones being merged. Therefore, no additional constructs are necessary. If any specific extension to the new net type is required, then the pattern of Figure 5(a) is applied.

## 5 The MDE-Framework Applied to Nets with Priorities and Time

We now use the model-based framework previously defined to build new net types through extension and composition of metamodels. First, PT-Nets with static and



**Fig. 5.** Modular construction of a new net type

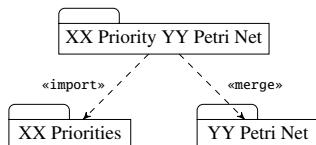


Fig. 6. Modular construction of prioritised Petri Nets metamodels

dynamic priorities are built, then PT-Nets with time. Afterwards, we present more elaborate compositions which are the projections of some of the later equations from section 3.3.

### 5.1 Metamodels for PT-Nets with Priorities

A prioritised Petri net basically associates a priority description with an existing standardised Petri net, thus building a new Petri net type. The metamodel in Figure 6 illustrates this modular definition approach, in line with the pattern of Figure 5(a). It shows a blueprint for instantiating a concrete prioritised Petri net type, by merging a concrete Petri net type and importing a concrete priority package. The *XX Priorities* package is the virtual representation of a concrete priority package and the *YY Petri Net* is the virtual representation of a concrete Petri net type.

For example, Figure 7 shows a prioritised PT-Net using static priorities only. It is built upon a standardised PT-Net which it merges, and a *Priority Core* package, which it imports. The *Priority Core* package provides the building blocks to define *Static Priorities*, as depicted by Figure 8.

The purpose of the *Priority Core* package is to provide:

- the root metaclass for priorities, represented by the *Priority* metaclass;
- a priority level, which is an evaluated value represented as a property of the *Priority* metaclass;
- the ordering policy among the priority values of the prioritised Petri net. This ordering policy is represented by the *PrioOrderingPolicy* metaclass.

The purpose of priority levels is to provide an ordered scalar enumeration of values such that either the higher the value, the higher the priority, or the lower the value, the higher

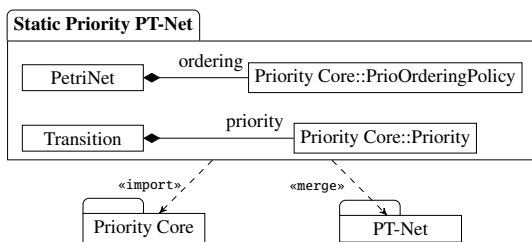


Fig. 7. Prioritised PT-Net metamodel showing how the priority description is attached

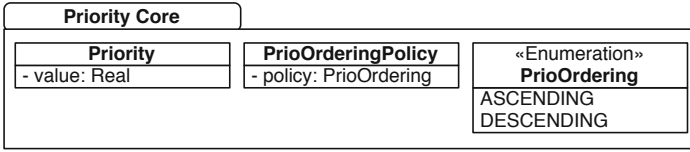


Fig. 8. Core package of priorities

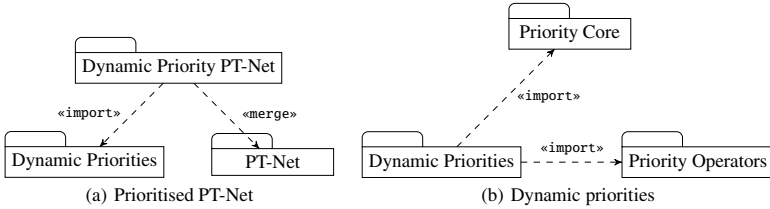


Fig. 9. Prioritised PT-Net metamodel using dynamic priorities

the priority. With the **Priority Core** package, and thanks to the **Priority** meta-class, static priorities can thus be attached to transitions, as in the **Static Priority PT-Net** shown in Figure 7.

Using the same approach, Figure 9(a) shows a prioritised PT-Net which uses dynamic priorities. Dynamic priorities are built upon **Priority Core**.

This modular construction follows the extension schemes adopted so far in the PNML standard, explained earlier in this section. For instance, High-Level Petri nets build upon Symmetric nets that they merge, and new specific sorts (such as List, String and arbitrary user-defined sorts) that they import. The use of the **merge** and **import** relationships is therefore consistent.

This approach is consistent with the idea that a new Petri net type subsumes the underlying one it builds upon, but the algebraic expressions it reuses are generally orthogonal to net types. Next, we introduce the metamodel for priorities.

**Priority Metamodel.** Prioritised Petri nets augment other net models (e.g. P/T or Symmetric nets) by associating a priority description with the transitions. Such priority schemes are of two kinds:

- *static priorities*, where the priorities are given by constant values which are solely determined by the associated transition<sup>5</sup>;
- *dynamic priorities*, where the priorities are functions depending both on the transition and the current net marking.

Figure 9(b) shows the modular architecture of priorities metamodels. The **Priority Core** package (detailed in Figure 8) provides the building blocks to define both **Static**

<sup>5</sup> For high-level nets such as Coloured nets, the priorities are given by constant values which are solely determined by the associated binding element.

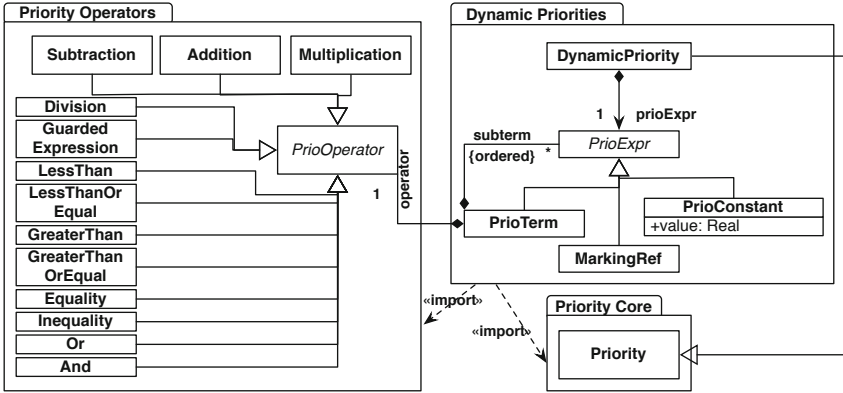


Fig. 10. Dynamic priorities and priorities operators packages

Priorities and Dynamic Priorities. However, dynamic priorities are further defined using Priority Operators. Dynamic priorities can encompass static ones by using a constant function (for the sake of consistency in the use of priority operators).

Figure 10 shows how the Dynamic Priorities metamodel is built. A DynamicPriority is a Priority Core::Priority. It contains a priority expression (PrioExpr). A concrete priority expression is either a PrioTerm which represents a term, a PrioConstant which holds a constant value or MarkingRef which will hold a reference to the marking of a place.

Note that the actual reference to the metaclass representing markings is missing. It must be added as an attribute (named ref) to MarkingRef once the concrete prioritised Petri net type is created. Its type will then be a reference to the actual underlying Petri net type Place metaclass, which refers to the marking.

A PrioTerm is composed of an operator (PrioOperator) and ordered subterms. This definition enables priority expressions to be encoded in abstract syntax trees (AST). For example, the conditional priority expression: *if*  $M(P2) > 3$  *then*  $3 \times M(P2)$  *else*  $2 \times M(P1)$ , is encoded by the AST of Figure 11 assuming that:

- P1 and P2 are places;
- $M(P1)$  and  $M(P2)$  are respectively markings of P1 and P2.

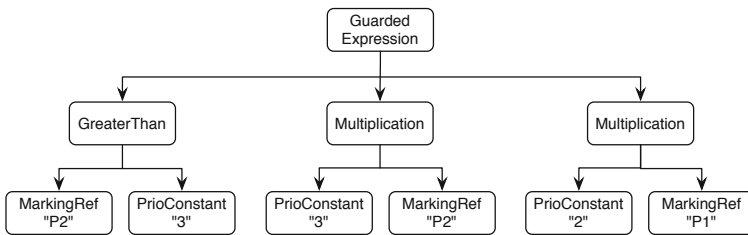


Fig. 11. AST of the conditional expression: *if*  $M(P2) > 3$  *then*  $3 * M(P2)$  *else*  $2 * M(P1)$

The priority operators are gathered within the `Priority Operators` package to allow for more flexibility in extending this priority framework. New operators can thus be added easily to this package.

Note that all these operators can also be found in ISO/IEC 15909-2, but are scattered among different sorts packages, being directly tied to the most relevant sort. For the next revision of the standard, we suggest that they be gathered in separate and dedicated packages (e.g. arithmetic operators, relational operators, etc.). This refactoring will allow for more reusability across different Petri net type algebra definitions.

## 5.2 Metamodels for PT-Nets with Time

Building a metamodel for TPNs starts with defining the metamodel for time features. We apply the same modular definition approach introduced in earlier sections. Regarding the particularly rich extension domain of TPNs, we explored several design choices. To ease the extensibility of this family of Petri nets, we sought to maximise the modularity of the definitions of the different concepts. The metamodel for time features presented in Figure 12 shows two packages, each containing a set of related features.

Package `Time4PetriNets` provides the building blocks to include time intervals and associated clocks, respectively represented by `TimeInterval` and `Clock` metaclasses. Package `Semantics4TPN` provides the different representation of the semantics for TPNs, as presented in section 2.3.

Since TPNs have a rich extension domain, `Semantics4TPN` is intended to be a flexible and easily extendable package for the different semantics. We thus have metaclasses for representing firing policies, reset policies and the server settings in TPNs. They are defined in another package which is imported by `Semantics4TPN`. That package is not shown here since it is too detailed for the granularity level of this example.

Using the metamodel for time features, we now define the metamodel for Time Petri nets, where time intervals are attached to transitions. Figure 13 shows such a metamodel, where the semantics policies are attached to the net itself, represented by the `Petrinet` metaclass. The new Petri net type package merges that for P/T nets, so as to be able to build upon existing constructs from P/T nets, within a new namespace. Note that, since a Petri net evolves using a single semantics only, it is attached to the whole model. The corresponding metaclasses are therefore attached to the net node in figure 13.

In a TPN the semantics is monoserver, strong and clocks are reset upon firing of newly enabled transitions. This can be specified by OCL constraints which will be

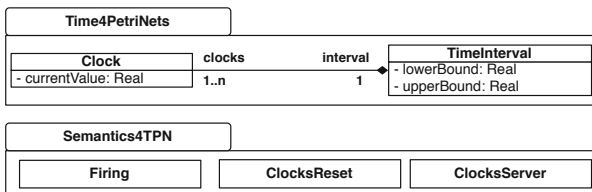


Fig. 12. Time features for TPNs

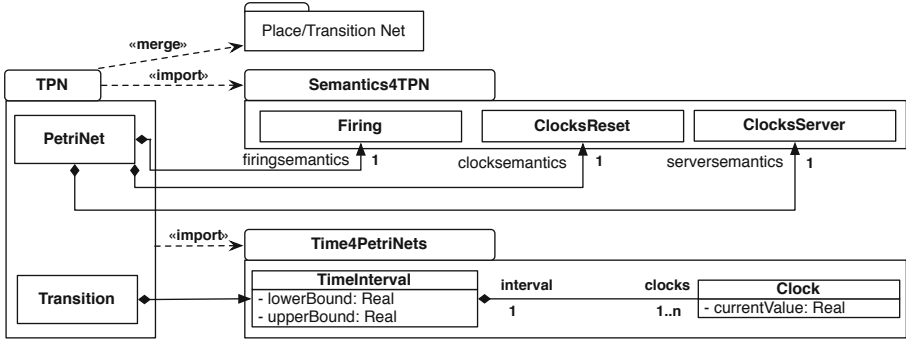


Fig. 13. Time Petri nets

globally attached to the `Semantics4TPN` package. These constraints are not presented so as to avoid cluttering the example.

These extensions of P/T nets by time features to build Time Petri nets can be removed, thus easily falling back to P/T nets, in line with the direction stated in section 3.2. With such a modular definition approach, it is easy to define the metamodel for P-time Petri nets<sup>6</sup> (P-TPN) and A-time Petri nets<sup>7</sup> (A-TPN). For instance, in the new metamodel of P-TPN, the `Place` metaclass is associated with `TimeInterval` and OCL constraints are updated so that the multiserver semantics is taken into account.

Next, orthogonality of combined features is fully implemented, through the definition of Time Prioritised Petri nets.

### 5.3 Metamodels for PT-Nets with Time and Priorities

We now consider building Dynamic Priority Time Petri nets (DPTPN), where two orthogonal extensions to P/T nets are combined. The features provided by these extensions are dynamic priorities and time. To do so, two main building blocks, already defined in earlier sections are needed:

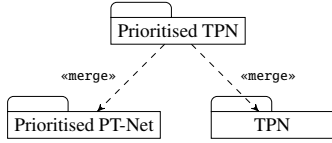
- Dynamic Priority Petri nets, whose metamodel is shown in Figure 9(a), and
- Time Petri nets, whose metamodel is shown in Figure 13.

Figure 14 depicts the metamodel of DPTPNs, where the new Petri net type package merges the Dynamic Priority PT-Net and the TPN ones. No additional construct is needed in the new package. Every concept comes from the building blocks, i.e. the merged packages.

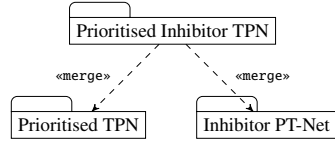
Indeed, in the new package, the `Transition` metaclasses from the building blocks are merged, yielding a resulting metaclass which holds a composition relationship with `DynamicPriorities::DynamicPriority` (see Figure 10) and another composition

<sup>6</sup> Based on [3], time intervals are associated with places, semantics is multiserver and strong.

<sup>7</sup> Based on [3], time intervals are associated with arcs, semantics is multiserver and weak.



**Fig. 14.** Prioritised TPN builds upon existing orthogonal net types



**Fig. 15.** Prioritised Inhibitor TPN builds upon existing orthogonal net types

relationship with `TPN::TimeInterval` (see Figure 13). The same applies to the resulting `PetriNet` metaclass formed by merging those from `TPN` and `Dynamic Priority PT-Net`.

Orthogonality is fully implemented in extending P/T nets through the composition of these two extensions. Whenever `TPN` is removed, the whole metamodel will fall back to `Dynamic Priority PT-Net`. Whenever `Dynamic Priority PT-Net` is removed, it will fall back to `TPN`. Whenever both extensions are removed, the metamodel will fall back to the one of P/T nets.

Next, we go one step further, by composing this new net type with special arcs.

### 5.4 Metamodel for PT-Nets with Time, Priorities and Special Arcs

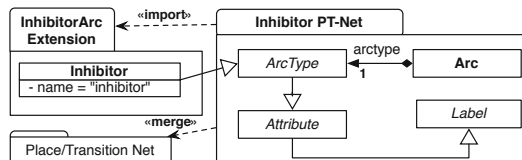
Finally, we want to build `DPTPNs` with special arcs, such as inhibitor arcs, as defined in equation 10. But first, let us define a metamodel for inhibitor PT-Nets.

Figure 16 shows the modular construction of inhibitor PT-Net, which reflects equation 7. The `Inhibitor PT-Net` package is actually sufficiently generic to be reused, by just renaming it, to build another net type with another kind of arc (e.g. reset). It will just require importing the new extension package, as a substitute for or in addition to the `InhibitorArcExtension` one.

Every building block required to construct a metamodel for `DPTPNs` with inhibitor arcs is now established. Figure 15 shows the straightforward modular construction of this new net type.

## 6 Technological Issues and Discussion

This section discusses technological issues related to the implementation of this model within the Eclipse Modeling Framework [28] (EMF).



**Fig. 16.** Modular construction of inhibitor PT-Net



**Current Implementation.** The approach advocated here was implemented in PNML Framework [10], which stands as one of the standard's companion tools. PNML Framework allows for handling Petri net types in a "Petri net way", in order to avoid any XML explicit manipulation. PNML Framework design and development follow model-driven engineering (MDE) principles and rely on the Eclipse Modeling Framework.

The implementation was successful, its main steps being:

1. designing the new metamodels,
2. annotating them with PNML specific information (tags),
3. assembling them and,
4. pushing a button to generate an API able to manipulate the new Petri net types.

The specific code generated by our templates to export and import models into/from a PNML file is created using the annotations decorating the metamodels in step 2. The PNML Framework plugin for Prioritised Petri nets, generated as a proof of concept for the presented extension approach, is available at <http://pnml.lip6.fr/extensions.html>. At this stage, it is provided as an Eclipse project with the PNML-ready source code the reader can browse and use. In the future, further updates with new net types will be provided on that web page. With the provision of our tool, it will be possible for other tool developers to define their own Petri net types, on top of existing ones.

**Technological Limitations.** We encountered some technological limitations during step 3 of the process because EMF does not yet offer a very convincing merge operation between models. The EMF Compare plug-in [7] seems to be a promising project towards this goal, but it is not yet mature enough for our use of merge. The merge had to be performed class by class, which is tedious for large metamodels.

OMG acknowledges that the UML package merge is too complex for tools to implement. Even though the Eclipse UML2 plugin currently implements this operation, it generates some inconsistencies.

Thus, to overcome this problem, we came to use the more robust `import` operation between EMF models. The procedure is the following:

1. When the composition pattern is based on merging a single base type into the new one:
  - start from the base Petri net type,
  - rename it as the new type,
  - imports the extensions.
2. When more than one base Petri net type is involved:
  - start from the one that was previously extended the most,
  - loop to step 1.

**Assessment within the Standardisation Process.** The model driven development approach for the standard metamodels caters for extensibility maintainability, usability and reusability. However, at this stage, quantitative evaluation requires experimentations by the community. This is planned within the standardisation process, especially in part 3.

Moreover, the standardisation team aims at evaluating the opportunity to provide a more detailed description of the semantical aspects of Petri nets via the definition of the enabling functions and firing rules.

## 7 Conclusion

In this paper, we have explored extensions suitable for part 3 of the Petri Net Standard. We have proposed a framework which justifies describing these extensions as orthogonal. We have demonstrated how such extensions can be implemented in PNML Framework, an MDE-based framework which is a companion tool to the standard. This can be the stepping stone for a more general extension mechanism to integrate new Petri net types within the standard.

The experiments presented in this paper rely on Place/Transition nets as a basis for extension. We could equally well have as well chosen Symmetric Nets, or High-level Petri nets. However, the presentation would have been longer and more clumsy with no additional technical value.

Beside the immediate outcome for the Petri net community, we consider this as an interesting contribution for handling formal notations by means of Model Driven Engineering techniques. So far, metamodel management is achieved through syntactical aspects only. Our framework better captures the behavioural semantics of Petri nets by connecting the enabling rule to the attributes of the Petri net objects.

Future work aims at building a composition framework encompassing a library of existing net types and extensions, along with rules that express their semantic compatibility. This would provide safe guidelines for the construction of new net types, taking advantage of reuse, and fostering sound contributions to the standard.

**Acknowledgments.** We thank Béatrice Bérard for her fruitful help with regards to the time Petri nets aspects. We would also like to thank the anonymous reviewers for their comments that enriched the paper.

## References

1. Bause, F.: Analysis of Petri Nets with a Dynamic Priority Method. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 215–234. Springer, Heidelberg (1997)
2. Bérard, B., Cassez, F., Haddad, S., Lime, D., Roux, O.H.: Comparison of Different Semantics for Time Petri Nets. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 293–307. Springer, Heidelberg (2005), <http://move.lip6.fr/Beatrice.Berard/PDF/bchlr-atva05.pdf>
3. Bérard, B., Lime, D., Roux, O.: A Note on Petri Nets with Time. Integrated in report on WG19 plenary meeting in Paris, ISO/IEC/JTC1/SC7/WG19 (2011)
4. Berthomieu, B., Diaz, M.: Modeling and verification of time dependent systems using time Petri nets. *IEEE Trans. on Soft. Eng.* 17(3), 259–273 (1991)

5. Christensen, S., Hansen, N.D.: Coloured Petri Nets Extended with Place Capacities, Test Arcs and Inhibitor Arcs. In: Ajmone Marsan, M. (ed.) ICATPN 1993. LNCS, vol. 691, pp. 186–205. Springer, Heidelberg (1993)
6. Czaja, I., van Glabbeek, R.J., Goltz, U.: Interleaving Semantics and Action Refinement with Atomic Choice. In: Rozenberg, G. (ed.) APN 1992. LNCS, vol. 609, pp. 89–107. Springer, Heidelberg (1992)
7. Eclipse Foundation: The Eclipse Compare project home page (2011), <http://www.eclipse.org/emf/compare/>
8. Green, R.: Java Glossary: Orthogonal (1996-2011), <http://mindprod.com/jgloss/orthogonal.html>
9. Harel, D.: Lecture on Executable Visual Languages for System Development (2011), <http://www.wisdom.weizmann.ac.il/~michalk/VisLang2011/>
10. Hillah, L.M., Kordon, F., Petrucci, L., Trèves, N.: PNML Framework: An Extendable Reference Implementation of the Petri Net Markup Language. In: Lilius, J., Penczek, W. (eds.) PETRI NETS 2010. LNCS, vol. 6128, pp. 318–327. Springer, Heidelberg (2010)
11. Hillah, L., Kindler, E., Kordon, F., Petrucci, L., Trèves, N.: A primer on the Petri Net Markup Language and ISO/IEC 15909-2. Petri Net Newsletter 76, 9–28 (2009), Originally Presented at CPN 2009
12. IBM: The IBM Language Extensions (1991), [http://publib.boulder.ibm.com/infocenter/lnxpcmp/v7v91/index.jsp?topic=%2Fcom.ibm.vacpp71.doc%2Flanguage%2Fref%2Fclrc00ibm\\_lang\\_extensions.html](http://publib.boulder.ibm.com/infocenter/lnxpcmp/v7v91/index.jsp?topic=%2Fcom.ibm.vacpp71.doc%2Flanguage%2Fref%2Fclrc00ibm_lang_extensions.html)
13. ISO/IEC: Software and Systems Engineering - High-level Petri Nets, Part 1: Concepts, Definitions and Graphical Notation, International Standard ISO/IEC 15909 (December 2004)
14. ISO/IEC: Software and Systems Engineering - High-level Petri Nets, Part 2: Transfer Format, International Standard ISO/IEC 15909 (February 2011)
15. ISO/IEC/JTC1/SC7/WG19: The Petri Net Markup Language home page (2011), <http://www.pnml.org>
16. Jensen, K., Kristensen, L.M.: Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer (June 2009)
17. Kummer, O., Wienberg, F., Duveigneau, M., Cabac, L.: Renew - User Guide. Tech. Rep. Release 2.2, University of Hamburg (2009), <http://www.renew.de/>
18. Lakos, C., Petrucci, L.: Modular State Spaces for Prioritised Petri Nets. In: Calinescu, R., Jackson, E. (eds.) Monterey Workshop 2010. LNCS, vol. 6662, pp. 136–156. Springer, Heidelberg (2011)
19. Lakos, C.: Composing Abstractions of Coloured Petri Nets. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 323–342. Springer, Heidelberg (2000)
20. Meyer, B.: Eiffel: The Language. Prentice Hall, New York (1992)
21. OMG: Unified Modeling Language: Superstructure - Version 2.4 - ptc/2010-11-14 (January 2011), <http://www.uml.org/>
22. Pahl, C.: Modular, Behaviour Preserving Extensions of the Unix C-shell Interpreter Language. Tech. Rep. IT-TR:1997-014, Department of Information Technology, Technical University of Denmark (1997), <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.8183>
23. Palsberg, J., Schwartzbach, M.: Object-Oriented Type Systems. Wiley Professional Computing. Wiley, Chichester (1994)
24. Pratt, T., Zelkowitz, M.: Programming Languages Design and Implementation, 3rd edn. Prentice-Hall (1999)
25. Raymond, E.S.: The Art of Unix Programming (2003), <http://www.catb.org/esr/writings/taoup/html/ch04s02.html#orthogonality>

26. Reynier, P.-A., Sangnier, A.: Weak Time Petri Nets Strike Back! In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 557–571. Springer, Heidelberg (2009)
27. SearchStorage: Definition: Orthogonal (June 2000), <http://searchstorage.techtarget.com/definition/orthogonal>
28. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Eclipse Series. Addison-Wesley Professional (December 2008)
29. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley (1998)
30. Winskel, G.: Petri Nets, Algebras, Morphisms, and Compositionality. *Information and Computation* 72, 197–238 (1987)

# Modelling Local and Global Behaviour: Petri Nets and Event Coordination

Ekkart Kindler

Informatics and Mathematical Modelling  
Technical University of Denmark  
eki@imm.dtu.dk

**Abstract.** Today, it is possible to generate major parts of a software system from models. Most of the generated code, however, concerns the structural parts of the software; the code that concerns the actual functionality or behaviour of the software system is often still programmed manually. In order to address this problem, we developed the notation of *coordination diagrams*, which allows us to define the *global behaviour* of a software system on top of existing class diagrams. One of the major objectives of coordination diagrams was to make it easy to integrate them and the code generated from them with existing structural models, with existing code, and with other behavioural models. Basically, coordination diagrams define how the *local behaviour* of the individual parts of the software is coordinated with each other. The main concepts of coordination diagrams and their semantics are stabilising now: We call it the *Event Coordination Notation (ECNO)*.

ECNO's coordination diagrams define the global behaviour of a system only: they define how the local behaviour is coordinated and jointly executed in so-called *interactions*. In principle, ECNO is independent from a specific notation for modelling the local behaviour. For our experiments with ECNO, however, we implemented a simple modelling notation for the local behaviour, which is based on Petri nets: *ECNO nets*. Together, ECNO coordination diagrams and ECNO nets allow us to completely model a software system, and generate executable code for it.

In this paper, we discuss the general idea of ECNO and of ECNO nets. ECNO nets are implemented as a Petri net type for the ePNK tool, together with a code generator that produces code that can be executed by the ECNO execution engine.

**Keywords:** Model-based Software Engineering, Local and global behaviour modelling, Event coordination, Petri nets, Code generation.

## 1 Introduction

Software models and the automatic generation of code from these models are becoming more and more popular in modern software development – as suggested by the success of one of the major approaches, the Model Driven Architecture (MDA) [1]. In many cases, however, modelling and code generation concern the

structural parts (as for example defined by class diagrams) or the standard parts of the software only; as soon as actual functionality or specific behaviour of the software is concerned, these parts of the software are often still programmed manually. As pointed out in our previous work [2], the reason that the code for behaviour is often still programmed manually is not so much that there are no modelling notations for behaviour or that it is difficult to generate code from behaviour models. The actual reason is that it is tedious to integrate the behaviour models or the code generated from them with the code generated from the structural models or with pre-existing code of parts of the software.

Based on some earlier ideas [2-4], *events* can be used to identify points in which different parts of the software could engage or participate; then, a *coordination notation* can be used for defining which partners can or must participate in an event in a given situation; a combination of a set of events and the participating partners that meet the requirements of the coordination model in a given situation, is called an *interaction*. This way, the overall behaviour of a system is a result of the coordination and synchronisation of events combined with the local behaviour of the different parts. The main concepts of the coordination notation are fixed now; the resolution of some subtle issues requires more experience with the practical use of the notation. In order to gain this experience, we have implemented a prototype, which consists of a modelling part, which we call *Event Coordination Notation (ECNO)*, and a framework and engine for its execution. From using this prototype, we hope to learn more about which constructs do help to adequately model and coordinate behaviour and to collect efficiency and performance results for larger systems and for more complex coordinations. This way, we intend to fine-tune ECNO's constructs and notation and to strike a balance between an adequate and smooth notation on a high level of abstraction with sufficient expressive power and universality on the one hand, and efficient and fast execution on the other hand.

ECNO focuses on the coordination of the behaviour of different parts of the system, i. e. on *global behaviour*. The *local behaviour* would still be programmed manually based on an API, which is part of the ECNO framework (see [5] for more details). On the one hand, the possibility of programming the local behaviour in a traditional way, makes it possible to integrate ECNO with classical software development approaches. We consider this possibility a major feature of ECNO – in particular easing the gradual transition from programmed software to fully modelled software. On the other hand, programming the local behaviour is not in the spirit of model-based software engineering (MBSE). Therefore, we started a sideline that is concerned with modelling the local behaviour of parts of the software from which the code for the local behaviour could be generated fully automatically. We use an extended version of Place/Transition-systems (P/T-systems) [6, 7] for that purpose, which we call *ECNO nets*. The reason for using Petri nets is mostly our own background in Petri nets and our generic tool, the ePNK [8], which allows us to easily define new types of Petri nets and which is integrated to the Eclipse platform, which provides all the necessary infrastructure

and technology for code generation from models and for software development in general [9].

We discuss the main concepts of ECNO's coordination diagrams in Sect. 2 and the concepts of ECNO nets in Sect. 3. The presentation is driven by a running example. The resulting implementation, the execution engine, and ECNO's current tool support are briefly discussed in Sect. 4. In the running example, we focus on the main concepts of ECNO; some more advanced concepts are discussed in Sect. 5. The novelty of ECNO lies mainly in the new combination of well-known and well-established concepts, which are discussed in Sect. 6.

## 2 The Event Coordination Notation

In this section, we discuss the main ideas and the main concepts of ECNO and its coordination diagrams. We start with our running example in Sect. 2.1 and summarise the main concepts in Sect. 2.2.

### 2.1 Example

We explain the concepts of ECNO by the example of a coffee (and tea) vending machine. Figure 1 shows a class diagram [1] with some extensions concerning *events* and their *coordination*. Therefore, we call it a *coordination diagram* [2].

Before explaining the extensions that concern the coordination, let us have a brief look at it as a class diagram. The diagram shows the different types of *elements* [3] that are part of the system and their possible relations: A coin can be close to the slot, which is represented by the reference from Coin to Slot, or a coin can be in the slot, which is represented by the reference from Slot to Coin. There is a Safe to which a coin is passed when a coffee or tea is dispensed. There is a Panel for the user to interact with the vending machine. The panel can be connected to controllers, which is represented by the reference from Panel to Control. A controller is connected to brewers, which can be either coffee or tea brewers. At last, there is an output device for the beverage, which is connected to the brewers.

As a class diagram, Fig. 1 can have instances (in UML, this would be *object diagrams*), each of which would represent a concrete *configuration* or situation of a vending machine. Figure 2 shows the initial configuration of our vending machine: There are three coins that are ready to be inserted to the slot, and there are two coffee brewers and one tea brewer; for all other classes, there is exactly one instance. Remember that, in the instances, the references of the class

<sup>1</sup> For the experts: In the tool, this will be an Ecore diagram [9].

<sup>2</sup> Note that coordination diagrams are fundamentally different from UML's communication diagrams (see Sect. 2.2 for details).

<sup>3</sup> In order to point out that our objects are a bit more than objects in the traditional sense of object orientation, we call them *elements* throughout this paper. Accordingly, we talk about *element types* instead of *classes*.

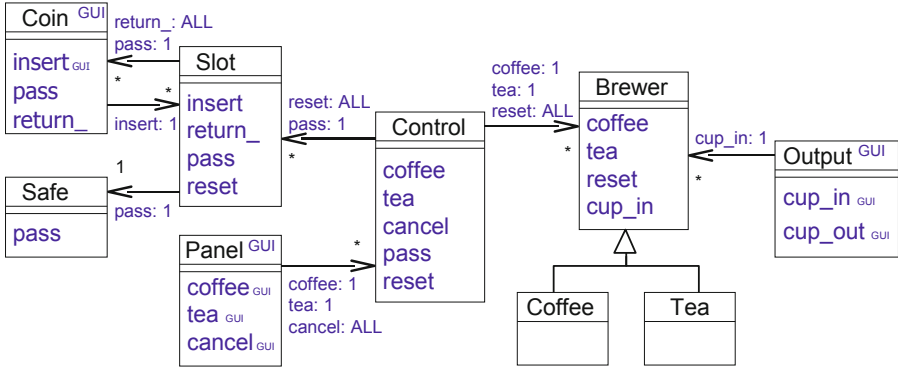


Fig. 1. A class and coordination diagram

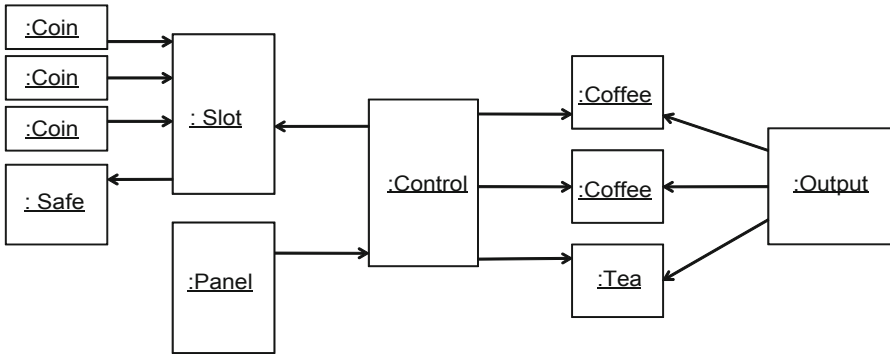


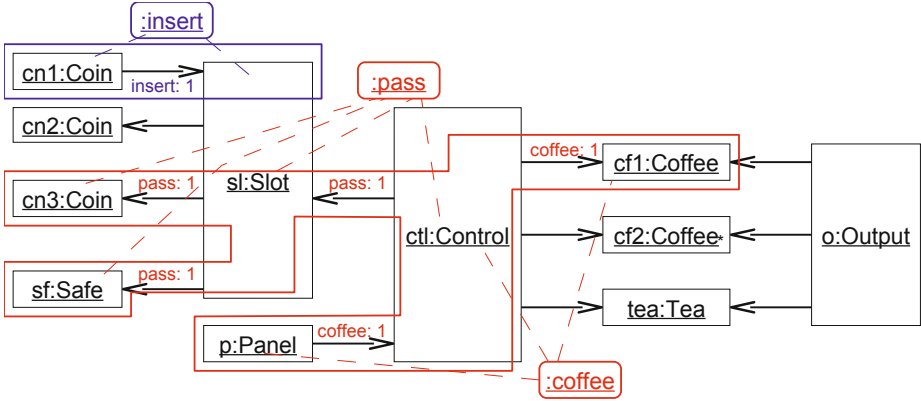
Fig. 2. A vending machine configuration

diagram are represented by *links* from one element to another (where a link’s type is the resp. reference of the class diagram).

Now, let us explain the extensions concerning the coordination of events between the different elements in Fig. 1: First of all, there are some events mentioned in the operations sections of the different element types, like `insert`, `pass`, and `return_`<sup>4</sup> for `Coin`. This defines in which *events* the respective elements could be involved (or participate in). The actual definition of these events is discussed later (see Fig. 4). More importantly, the references between the different element types are annotated with events and an additional quantifier, which can be 1 or ALL. These *coordination annotations* define the coordination of events and the participating elements: more precisely, for an element executing some event, it

<sup>4</sup> Since `return` is a keyword of our target language Java, we use an additional underscore in the event name `return_`.





**Fig. 3.** Another vending machine configuration with two interactions high-lighted

defines which other elements need to participate in the execution of this event. We call a combination of all the required elements and events an *interaction*. Figure 3 shows another configuration of the vending machine after two coins (cn2 and cn3) have been inserted to the slot. On top of this configuration, two examples of interactions that are possible in this situation are indicated (note that more interactions are possible, which are not shown here).

For explaining the meaning of coordination annotations, let us assume that some element is involved in the execution of some event and that, in the coordination diagram, the type of this element has a reference that is annotated with that event. Then, some elements at the other end of the respective links also need to participate in the interaction. In Fig. 3, for example, panel *p* is involved in the event *coffee*, which is indicated by the dashed line to the event *coffee*. For panel *p*, the coordination annotation *coffee:1* at the reference to control implies that the control also needs to participate in the interaction with that event; in turn, the reference from control to brewer annotated with *coffee:1* requires that also one brewer must participate: In the interaction that is outlined in Fig. 3, the choice is coffee brewer *cf1*; but coffee brewer *cf2* would be okay too – giving rise to another interaction which is not outlined in Fig. 3. For reasons that will become clear later, the control is required to participate in a *pass* event together with a *coffee* event. This required *pass* event is again indicated by a dashed line in Fig. 3. Because of the coordination reference from Control to Slot, the slot is required to participate in the *pass* event too. The slot, in turn, has two references annotated with *pass:1*, one to the class *Coin* and one to the class *Safe*, both of which must be met. Therefore, one coin and one safe need to participate in the interaction – this way, the coin will be passed from the slot to the safe, which will be discussed in more detail later in Sect. 3.1. Altogether, this gives us an interaction with six elements and two events as outlined in Fig. 3. Note that another possible interaction in that situation is outlined at the top of Fig. 3.

```

insert(Coin coin, Slot slot);  reset();    coffee();  cup_in();
pass(Coin coin, Slot slot);   cancel();  tea();    cup_out();
return_(Slot slot);

```

**Fig. 4.** Event type declarations

As discussed above, a coordination annotation refers to an event and has a quantifier, which can be 1 or ALL. In the example above, we have discussed the quantifier 1 already: this quantification means that one partner at the other end of the links corresponding to that reference must participate.

If the event in the coordination annotation is quantified by ALL, all the elements at the other end of these links need to participate. In the situation shown in Fig. 3, there are two coins inserted to the slot, which is represented by the two links from the slot to the coins `cn2` and `cn3`. If the slot participates in a `return_` event, the annotation `return_:ALL` at the reference from Slot to Coin means, that both coins must participate in the execution of the event `return_`, which, as we will see later, actually returns all coins. Note that this interaction is not outlined in Fig. 3 in order not to clutter the diagram.

Up to now, events have, basically, been used as names, which were used in coordination annotations to identify other partners that need to participate in an interaction. In addition to that, events can also be used to exchange information between the partners of an interaction. To this end, events can have *parameters*. The declarations of the events of our vending machine along with their parameters are shown in Fig. 4 in order to distinguish them from the concrete instances in interactions, we actually call them *event types*. At a first glance, the declaration of an event type looks like a method declaration. In contrast to methods however, event types or events do not have behaviour of their own. Events are used only for synchronising participants in an interaction and to share information between them. Moreover, events are shared between different elements and do not belong to a particular element. This is why events are declared outside a specific element and are types in their own right. In particular, events do not have a caller or callee. Therefore, event parameters can be contributed in many different ways, and by different elements. It is not defined in advance who will provide and who will use the parameters and in which direction the values will be propagated. ECNO's semantics and the ECNO execution engine, however, guarantee that all elements participating in an interaction have the same parameters for the same event (instance) – if two partners contribute inconsistent values to the same event, the interaction is not possible. We will see some more details of how the local behaviour of an element contributes parameters to an event and how it uses them in Sect. 3.

A minor extension of coordination diagrams on top of class diagrams are the GUI annotations. These annotations indicate which elements and events are relevant to the end user: As the name GUI indicates, this is relevant for the GUI part of the execution engine for generating buttons and user dialogs.

## 2.2 Discussion and Summary of Concepts

Altogether, ECNO's *coordination diagrams* extend class diagrams by the explicit definition of *events*, resp. *event types*, and coordination annotations, which define in which way different elements need to participate in an interaction.

In a given configuration, an *interaction* can be represented as a set of elements, where each element is associated with some instances of events of some type. Note that different elements can be associated with the same event instance and that an element can be associated with more than one event (see Fig. 3). An interaction is possible or *valid* in the given configuration, if for each element each coordination annotation of its *element type* is *valid*:

- For an element, a coordination annotation  $e:1$  is valid, if either the element is not associated with an event instance of event type  $e$  at all, or if there exists a link for the resp. reference to another element that is associated with the same event.
- For an element, a coordination annotation  $e:ALL$  is valid, if either the element is not associated with an event instance of event type  $e$  or if, for all links for the resp. reference, all elements at the other end of these links are part of the same interaction, and all these elements are associated with that same event.

At a first glance, coordination diagrams look similar to UML's communication diagrams (also called collaboration diagrams in earlier versions of UML). But, there are some fundamental differences between ECNO's coordination diagrams and UML's communication diagrams: First of all, UML's communication diagrams are defined on top of an object diagram; ECNO's coordination diagrams are defined on top of class diagrams. Moreover, communication diagrams define the order in which messages are sent between the objects in a particular setting, whereas coordination annotations can be considered as independent requirements on partners that must participate in an interaction. Methodologically, a communication diagram is a kind of example scenario. By contrast, ECNO's coordination diagrams formulate rules that define which elements and events need to participate in a valid interaction. In any given configuration, the coordination diagram together with the local behaviour of the elements define all currently possible interactions. When an interaction is, actually, executed no order is specified among the different elements.

The basic mechanism for defining these coordination requirements is annotating references of the class diagram with an *event type* and a *quantifier*. Each of these annotations is bilateral. In combination, however, they can require that many different elements participate in an *interaction* (cf. Fig. 3): First, there might be different references for the same event, which require different other elements to participate. Second, the other elements that are required to participate might have references with annotations, which require further elements to participate; in this way, establishing a chain or network of required elements until all requirements are met. Third, an event annotation with quantifier **ALL** requires that all the elements at the other end of the respective links participate.

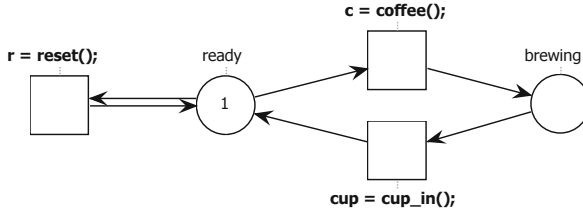


Fig. 5. Local behaviour of the coffee brewer

Fourth, the local behaviour of an element can require the synchronisation of two or more different events that all need to be part of the interaction. As we will see later, the local behaviour of the control in Fig. 8 enforces the combination of events `coffee` and `pass`. The required other event, may in turn impose additional requirements on participating partners. This way, coordination diagrams define the *global behaviour* of a system by coordination annotations based on the *local behaviour* of its elements<sup>5</sup>.

A coordination diagram does not say anything about the possible local behaviour of the elements. In essence, the local behaviour answers the following questions: when can an element participate in an event, what is the local effect when the element participates in such an interaction, and which events need to be executed together. The ECNO framework provides an API for programming the local behaviour for every element<sup>6</sup>. In this paper, we discuss an extended version of Petri nets for modelling such local behaviour.

### 3 Modelling Local Behaviour with ECNO Nets

The *local behaviour* of an element defines when and under which conditions the element can participate in an event or in a combination of events, and it defines what happens locally for the element when it participates in an interaction.

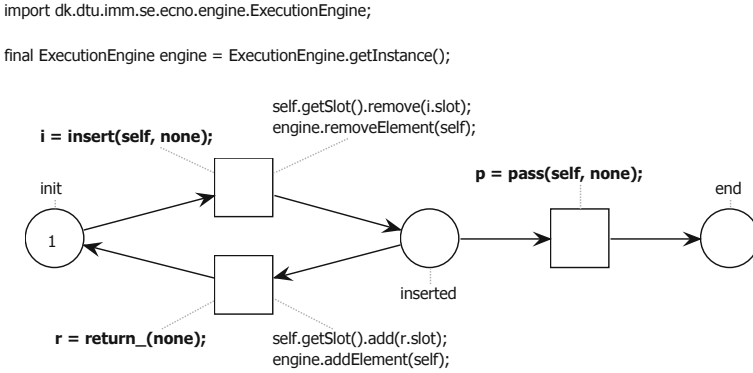
#### 3.1 Examples

Not surprisingly, such local behaviour can be defined by a slightly extended version of P/T-systems, which we call *ECNO nets*. We discuss the main concepts of ECNO nets by the help of some elements from our running example first. The concepts will then be clarified and explained in general in Sect. 3.2.

We start with a simple ECNO net, which models the local behaviour of the coffee brewer. It is shown in Fig. 5. Except for the annotations associated with the transitions, this is a conventional P/T-system. A transition annotation relates a transition to an event<sup>6</sup>; this annotation is called *event binding* and is graphically

<sup>5</sup> Similar in spirit, but technically very different, Harel and Marelly called the global behaviour inter-object behaviour and the local behaviour intra-object behaviour [10].

<sup>6</sup> We will see later that a transition can actually be bound to more than one event – this way enforcing the joint execution (the synchronisation) of two different events.



**Fig. 6.** Local behaviour of the coin

represented in bold-faced font. After the event `coffee`, which represents the user pressing the coffee button, the coffee is brewed, which will be dispensed, when a cup is inserted (event `cup_in`). The `reset` event is possible only when the coffee machine is in the `init` state (no coffee is being made). In this example, the notation for event bindings might appear overly verbose, and it is not obvious why the event needs to be assigned to some “variable”, as in `c = coffee()` in this case. The reason for denoting an event binding as an assignment will become clear in the next example, when we need to refer to the event’s parameters in conditions or actions.

The local behaviour of a coin, which is shown in Fig. 6, is more interesting<sup>7</sup>. First of all, the event bindings and the involved events have parameters. Let us consider the transition that is bound to the `insert` event first: as we have seen in Fig. 4, the event `insert` has two parameters: the coin and the slot. The event binding refers to these two parameters in the order of their declaration: The first one, `self`, assigns the coin itself as the first parameter (coin) to this event, where `self` is a keyword allowing the local behaviour of an element to refer to the element itself. The second parameter is `none`, which is another keyword indicating that, in this instance, the coin does not assign a parameter to the event `insert` (this parameter is provided by another partner of the interaction). The other annotation of this transition is the *action*, which will be executed when all partners of an interaction are found and the interaction is executed. This annotation is shown in normal font. For the transition bound to the `insert` event, the action does two things: First, it deletes the link to the slot (since it is inserted now); to this end, the action refers to the element again by `self` and uses the API generated from the class diagram by the Eclipse Modeling Framework (EMF) [9] to remove its link to the slot; the slot that is removed is by

<sup>7</sup> Note that in a real model of a vending machine, a coin would probably not have a behaviour of its own; it is just a piece of metal. For making the example more interesting, we associate some behaviour with the coin.

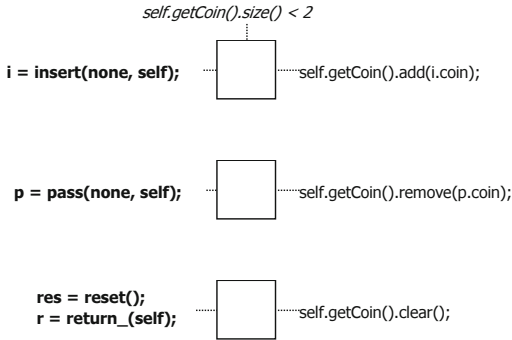


Fig. 7. Local behaviour: slot

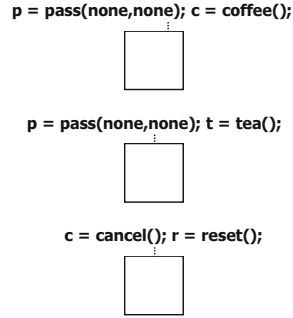


Fig. 8. Local behaviour: control

`i.slot`, where `i` is the variable to which the `insert` event was assigned, and `slot` is the respective parameter, which, in this case, is assigned to the event by the element `slot` (see Fig. 7). In order to refer to the `insert` event and its parameters, the notation for an event binding gives a name to every event. Second, the action removes the coin from the ECNO engine so as not to be visible at the GUI anymore, where the engine is accessed via the attribute engine, which is discussed later.

Once the coin is inserted, the ECNO net allows two things to happen: either the coin can be passed to the safe by the transition that is bound to event `pass`, or the coin is returned by the transition bound to event `return_`. In the case of a `pass` event, the coin assigns itself (`self`) as the coin parameter; and there is no action. In the case of a `return_` event, no parameter is assigned to the event (`none`), but the action will add a link from the coin to the slot again, where the slot is coming from the parameter `r.slot` of the event `return_`. Moreover, the coin registers itself with the engine, so as to be visible in the GUI again.

In the local behaviour of the coin, we see two other extensions of ECNO nets. At the top, there is an import statement, which, in this case, is needed to access the ECNO execution engine. The second extension is a declaration and initialisation of the attribute engine, which is used in the actions of the ECNO net. These declarations follow the syntax of the Java language. In this case, the additional Java keyword `final` actually defines a constant.

Figure 7 shows the local behaviour of the slot. This is a rather degenerated ECNO net. As a P/T-system, all transitions would be enabled all the time since their presets are empty. Due to the event bindings, however, the local behaviour becomes a bit more interesting. We start with explaining the bottom transition: This transition, actually, has two events bound to it: `reset` and `return_`. This implies that `reset` and `return_` must be executed together; this way, the slot defines that all coins must be returned during a reset. The slot assigns itself as a parameter to the `return_` event, which is used by the action of the coin to set the link to this slot again. Moreover, the slot deletes all the links to the coins it contains (i. e. it returns the coins) using the API generated by EMF.

The transition bound to the `pass` event is even simpler. When this event happens, the link to the coin that is passed (accessible via the parameter `p.coin`) is removed (since the coin is moved from the slot to the safe).

At last, let us discuss the top transition of Fig. 7. It is associated with an `insert` event, where the slot assigns itself to the event's slot parameter (since the slot should contain the coin after the insert). In the action, it sets a link to that coin. In this transition, we see another modelling concept of ECNO nets: the *condition* or guard, which is shown above the transition in italics. This condition guarantees that an `insert` event can happen only when there are less than two coins in the slot, where the condition refers to the list of coins which the slot has links to by `getCoin()`. In general a condition can refer to the events' parameters and anything that the element (`self`) can access via the information available in the object locally or accessible via its API (generated by EMF).

The ECNO net models for the other elements are similar. The last one that we discuss here is the one for the control. This net is shown in Fig. 8. The first transition guarantees that an event `coffee` goes together with an event `pass`. By this, it is indirectly guaranteed that there is a coin inserted in the slot when the corresponding interaction is valid; and other parts of the interaction (the action of the slot) will pass this coin to the safe – making sure that the coffee is paid when the coffee button is pressed. Actually, the coffee button on the panel is enabled only when there is a coin inserted in the slot and when at least one of the coffee brewers is ready. The second transition does the same for event `tea`. The last event synchronises the `cancel` event (which is triggered by pressing the cancel button on the panel) with the `reset` event: this way, it is indirectly guaranteed that all coins that are currently inserted in the slot are returned when cancel is pressed (see the ECNO net for the slot in Fig. 7).

### 3.2 Concepts

The main concept for modelling the local behaviour of an element is to define when the element can participate in an event or a combination of events and what happens when the event is executed. In ECNO nets, the event bindings for transitions define when an event or a combination of some events are possible.

In the event binding, an element can provide values for some parameters for the respective events. In our example, these parameters were expressions with values of the element. But, these expressions can be more general. For example, if we have some event type `event(Integer x, Integer y)`, it would be possible that an event binding looks as follows: `e = event(none, e.x + 1)`. The meaning of this is that the element takes the first parameter of the event, increments it, and assigns that value to the second parameter. If there were more events in the binding, we could also use a parameter of one event and assign it as a parameter to another event. There is no restriction in which way this could be done. When there are cyclic dependencies in the parameter assignments of the partners of an interaction, however, it will not be possible to assign all parameters. In that case, the interaction is not valid and will not be executable. The execution engine will detect that and properly deal with these situations when they occur at runtime.

In other situations, two or more different elements could try to assign a value to the same parameter of the same event. Here, we distinguish two cases: if all the values provided for the same parameter are the same (`equal` in Java terms), this is considered to be legal; if the values, however, are not the same, the interaction is considered to be invalid.

On the one hand, this mechanism of assigning and using parameters provides much expressive power and flexibility. Within the same interaction, data can be passed in opposite directions. That is why events are fundamentally different from function calls or methods invocations – there are no callers or callees, and the contribution and use of parameters is completely symmetric. On the other hand, such power requires great care when using this mechanism in order not to preclude desired interactions by unintended cyclic dependencies. This, however, is a question of methodology and analysis functions for checking that cycles of that nature would not occur (which however are yet to be developed). The ECNO execution engine copes with these situations: it will find all possible legal interactions – though computing all the parameters might be quite computation intensive in complex situations. Making this more efficient is ongoing research.

In ECNO nets, transitions have two more extensions: *conditions* and *actions*. Conditions are expressions that may refer to local attributes of the element (and whatever the element can access from there), and they may refer to the parameters of the events that are bound to the respective transition. Conceptually, the condition is evaluated when all parameters of the events bound to this transition are available. If, in a given combination of events, the condition of an element evaluates to false, the interaction is invalid. Only if the conditions of all participating elements evaluate to true, the interaction is valid and, therefore, executable. The execution of a valid interaction amounts to executing all the actions of all participating elements (in an arbitrary order). An action may refer to and access and change the local attributes of the element and all the elements it can access by its local attributes or the events' parameters. An event parameter itself, however, cannot be changed in order to avoid non-local side effects when executing all the actions of an interaction. This, basically, follows the principle of parameter passing in the Java language.

Altogether, ECNO coordination diagrams for modelling the global behaviour together with ECNO nets for modelling the local behaviour allow us to fully model the behaviour of a system, in our case the vending machine. The generated code together with the ECNO execution engine implement that behaviour, which will be discussed in the next section.

## 4 ECNO: Engine and Tool

In the previous sections, we have discussed ECNO's concepts for modelling global and local behaviour, and how they define valid interactions. Now, we discuss some details of ECNO's tool support, its execution engine and the code generated from ECNO models. Here, we discuss version 0.2.0 of the ECNO prototype, which is deployed as an extension of Eclipse 3.7 (Indigo) based on the



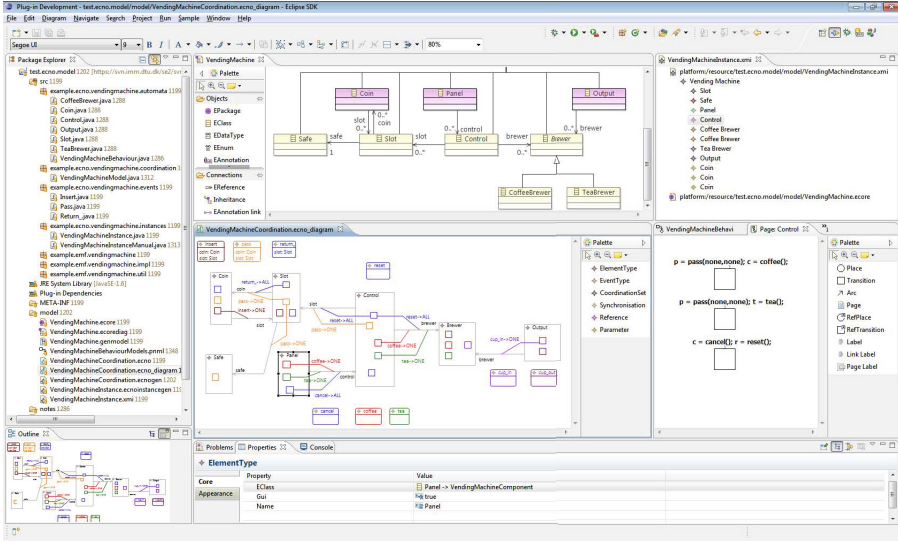


Fig. 9. ECNO: Tool support

ePNK [8]. The prototype and the instructions on how to install it can be found on the ECNO Homepage at <http://www2.imm.dtu.dk/~eki/projects/ECNO/>. The vending machine example, as discussed in this paper, can be obtained as an Eclipse project from <http://www2.imm.dtu.dk/~eki/projects/ECNO/version-0.2.0/download/ECNO-topnoc-example-0.2.0.zip>.

### 4.1 ECNO Tool: Vending Machine Example

Figure 9 shows Eclipse with the project of our vending machine example and some of its models opened. In the Eclipse package explorer on the left-hand side, the folder model contains all the models for the vending machine example. From these models, all the code for running the example can be generated fully automatically. The generated Java packages are contained in the folder src. To the right of the package explorer, some of the models are open in an editor.

Let us briefly explain the role of the different models, and how they affect the generated code. `VendingMachine.ecore` is the Ecore model of the vending machine; the respective diagram `VendingMachine.ecorediag` is open in the top-row on the left. From these and the `VendingMachine.genmodel`, the code for the structural part can be generated completely automatically by the standard EMF mechanisms. This code is contained in the packages `example.emf.vendingmachine.*`, which are not open here and are not discussed since this is standard EMF code.

`VendingMachineCoordination.ecno` contains the ECNO coordination diagram, where the graphical representation of this model is contained in `VendingMachineCoordination.ecno_diagram`. This diagram is open in the bottom row on the

left in a very simple GMF editor, which uses a slightly different graphical syntax; the information is the one from Fig. 11 and Fig. 4. Note that this diagram includes the definition of the event types. The Java class `VendingMachineModel` in package `example.ecno.vendingmachine.coordination` is generated from this model, which represents the coordination information at run-time. In addition, the classes for events with parameters<sup>8</sup> are generated from this diagram, which can be found in package `example.ecno.vendingmachine.events`.

`VendingMachineBehaviourModels.pnml` contains all the ECNO nets for the local behaviour of elements with non-trivial behaviour. The ECNO net for the Control is shown in the bottom row to the right. From these nets, the Java classes in package `example.ecno.vendingmachine.automata` are generated. There is one class for each net and one extra class that is used for making all these behaviour classes known to the ECNO engine: `VendingMachineBehaviour`.

Actually, the Java code for the coordination diagrams and the ECNO nets is generated in a single go. This needs some additional information that defines the packages to which the different parts of the code should go, and the names of the generated configuration classes. This information is similar to the EMF genfile; in our example, it is `VendingMachineCoordination.ecnogen`. From this file, the actual code generation is initiated.

The initial configuration of the vending machine is defined in `VendingMachineInstance.xmi`. The editor in the top row to the right shows the configuration from Fig. 2 in a standard EMF tree editor – it is a so-called dynamic instance of the Ecore model. Together with some additional configuration information from `VendingMachineInstance.ecnoinstancegen`, the code for this instance is generated: `VendingMachineInstance` in package `example.ecno.vendingmachine.instances`. This Java class can be started as a Java application – as usual in Eclipse.

## 4.2 Running the Vending Machine Example

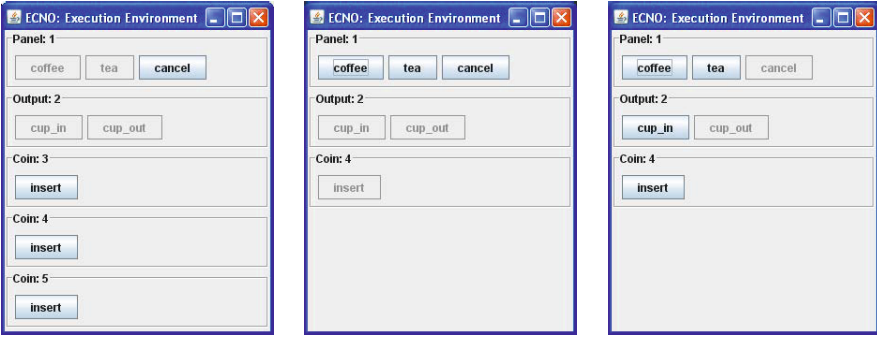
When the Java class `VendingMachineInstance` is started as a Java application, a GUI will start, which looks like the ones in Fig. 10. The left one shows the GUI of the vending machine immediately after start up; the middle one shows the GUI after pressing the “insert” buttons on two coins. Note that the last coin cannot be inserted anymore, due to the condition that ensures that a slot can not take more than two coins (see Fig. 7); therefore, the “insert” button is not enabled in this situation. The GUI to the right shows the situation after pressing the “coffee” button. Note that since two coins were inserted before and since there are two independent coffee brewers, you could order the next coffee right away (indicated by the enabled “coffee” button).

## 4.3 Code for ECNO Nets

As discussed in Sect. 4.1, there are many different Java classes generated from the models – some of them generated even by other technologies like EMF. And

---

<sup>8</sup> Events without parameters are, basically, used as names only; therefore, we do not need a class representing instances of these events.



**Fig. 10.** Running machine: initial, after inserting two coins, after ordering a coffee

there is some more code behind the scenes, which is provided by the ECNO engine and its run-time environment, which integrates the different parts – in particular the code for the class diagram with the code for the ECNO coordination diagrams and ECNO nets. For lack of space, we cannot discuss this code in detail here. But, we give an overview of the code generated for ECNO nets: a class extending `AbstractPetriNetBehaviour` from the ECNO run-time, which provides some infrastructure for mapping Petri net behaviour to ECNO concepts.

Listing 1 shows some snippets from the code that was generated from the ECNO net for the local behaviour of the coin from Fig. 6, with major omissions, which are indicated by ellipses. Anyway, it should give an idea of how the generated code looks like. The first line, shows the import declaration as defined in the net, and line 5 shows the attribute declaration. Line 4 is an automatically generated attribute, which “implements” ECNO’s keyword `self`; this attribute is set in the constructor of this class, which is not shown in the listing.

The methods `enabled` and `fireImpl` implement the enabledness of a transition and the change of the marking when it fires, where the marking is represented as an array; since this is trivial, we do not show the details here. The methods `doesAssignParam` and `getParamAssignment` are responsible for assigning the values to the parameters of a possible event. `doesAssignParam` says whether, for some transition, event and parameter (each represented by a number), this transition would assign a value to the parameter (i. e. whether it is not `none` in the ECNO net). If so, the actual value is provided by the method `getParamAssignment`, which will need the context of the concrete choice in order to access event parameters (since this is similar to the action which is discussed next, we do not discuss this here). `evaluateCondition` checks the condition of the choice.

The method `executeAction` executes the actual action once the interaction with this choice is executed. The transition number of the involved choice is obtained from the choice. For each case (here we discuss the case 0, which represents the transition for the `insert` event), the code of the transition action is executed. In order to access the event parameters, however, the events need to be made available. In our example, the `insert` event (an instance of the

**Listing 1.** Snippets from the Java class generated from the ECNO net Coin

```

1 import dk.dtu.imm.se.ecno.engine.ExecutionEngine;

public class Coin extends AbstractPetriNetBehaviour {
    final private example.emf.vendingmachine.Coin self;
5    final ExecutionEngine engine = ExecutionEngine.getInstance();
    ...
    public boolean enabled(int t) ...

    public void fireImpl(int t) ...

10    public boolean doesAssignParam(int t, int e, int p) ...

    public Object getParamAssignment(PetriNetChoice c, int e, int p) ...

15    public boolean evaluateCondition(PetriNetChoice c) ...

    public void executeAction(PetriNetChoice choice) {
        int transition = choice.getTransition();
        switch (transition) {
20        case 0: {
            Insert i = (Insert) choice.getEventValues("insert");
            { self.getSlot().remove(i.slot); engine.removeElement(self); }
            fire(transition);
            return;
        }
    }
}

```

automatically generated class `Insert` from the events package) is obtained from the choice. And then the code from the transitions action (line 22) is executed and the transition is fired.

Altogether, the code generation is quite straight-forward: the code snippets from the ECNO nets are literally copied to the right place in the generated class. There is just some wrapping and preparing code needed to make this work – like providing the attribute `self` and obtaining the relevant events in the action (and similar code, in the methods for condition evaluation and parameter assignment).

#### 4.4 Execution Engine and Controllers

ECNO coordination diagrams and ECNO nets define which interactions are valid or possible in any given configuration. They do not define which of these interactions must or will be executed. And the ECNO execution engine does not make this choice either. These choices will be made by *controllers*: a controller registers with the ECNO engine for some element and some event type. Once registered, the ECNO engine will keep these controllers informed about the possible interactions in which the element could participate for the given event type. When the controller has obtained a valid interaction, it can schedule it for execution. The ECNO engine will make sure that interactions are executed

atomically and without interference of possibly conflicting interactions. As soon as the enabledness of some interactions changes – either due to the execution of other interactions or by changes of the structure by other parts of the software – the respective controllers will be informed. The details on the implementation of these mechanisms are discussed in a separate paper [11]. In this paper, it is also discussed how ECNO can be integrated with other technologies and pre-existing code.

In our example, some standard GUI controllers are automatically set up, when the engine is started. For every element with an element type that has a GUI tag and every event type with a GUI tag, a GUI controller will be attached to it. The ECNO engine will notify these controllers when interactions become enabled or disabled; then, the respective button of the GUI will update accordingly. The actual choice of executing the interaction is then made by the end user by clicking on an enabled button of the GUI.

## 5 Discussion and Extensions

In the previous sections, we have discussed the main concepts of the Event Coordination Notation and its concepts for coordinating global and local behaviour. In order to convey the spirit of ECNO, some of its less important concepts were not discussed yet. In this section, we discuss these additional concepts – all of which are implemented in the prototype of the ECNO engine (version 0.2.0).

### 5.1 Coordination Sets

As discussed in Sect. 2, we assumed that, for identifying all the partners that an element needs when executing some event of type  $e$ , we must consider all the references of the element's type that have a coordination annotation for  $e$ . For executing the `pass` event in the slot, for example, two references annotated with `pass`, one to element type `Coin` and the other to `Safe`, required that a coin and the safe needed to participate in this interaction – this way, passing the coin from the slot to the safe. In most of our examples, this is the desired behaviour. In some cases, however, we do not want to consider all the references that are annotated by the event type. Sometimes, only one or a particular subset of references would be needed. To this end, ECNO provides the concept of coordination sets.

A *coordination set* is a set of references that start at the same element type and have the same event type attached to it<sup>9</sup>. For the same event type, an element type can have several coordination sets. If there is more than one coordination set for an event type, only for one of these coordination sets all its references must be considered together when the element is participating in the event. References that are not in this coordination set are not considered. Putting two references of the same element type in two different coordination sets would make these

---

<sup>9</sup> Actually, it is an open issue, whether it would make sense to have references with different event types in the same coordination set.

references an alternative. We did not finally decide yet on a graphical syntax for a coordination set – it will probably be a label attached to the element that has a pointer to all the references that belong to the same coordination set (similar to the simple GMF editor for coordination diagrams in Fig. 9).

Anyway, coordination sets add clutter to the graphical representation of coordination diagrams. In order to avoid clutter in coordination diagrams in standard situations, where all references with the same event type are in the same coordination set, we introduce an abbreviation, which coincides with the meaning of coordination diagrams that we used in Fig. 1: For each element type and each event type, there is a distinguished *standard coordination set*. Any coordination reference that is not attached to an explicitly defined coordination set is assumed to be part of this standard coordination set. Since the diagram from Fig. 1 does not have explicit coordination sets, all references with the same event type attached belong to the standard coordination set – which is why they are joined, as discussed in the example.

## 5.2 Exclusive and Collective Parameters

As discussed before, the parameters of events allow different partners of an interaction to share values. Typically, one partner would contribute the value for an event parameter and many partners would use it. But, it would also be possible that several partners contribute a value – provided that it is the same value. Therefore, we call these parameters *exclusive*.

In some cases, however, we would like to allow different partners to contribute different values to the same parameter of an event. We call these parameters *collective parameters*. In the declaration of an event type, it would be indicated for each parameter whether it is collective or not. Then, there could be any number (including zero) of values contributed to such a parameter, but each partner can contribute at most one value. When such a parameter is accessed in a condition or an action, it returns a list of all the values that were contributed.

One example of the use of a collective parameter could be that all participating elements assign themselves as a value to this parameter. This way, a partner could set links to all the partners of the interaction in its action.

## 5.3 Inheritance on Events

Another concept is *inheritance* on events: a *derived event type* could have some more parameters than the event type it is derived from. In our example, the brewer could be associated with a general event *drink*, which could be specialised to events *coffee* and *tea* in the coffee and tea brewers. This way, the vending machine can be modelled in a slightly more elegant way. Version 0.2.0 of the ECNO prototype supports inheritance of events already and comes with a vending machine example exploiting event inheritance. Note that there are some subtle issues, like exclusion of multiple inheritance, which we cannot discuss here.

## 5.4 Determinism

In any given situation of the system, there is a precisely defined set of *valid interactions* that are possible. The ECNO engine itself does not make any decision which one to execute; it only notifies the registered controllers about the possible interactions (and keeps them posted, when the situation changes). It is up to the controllers to choose the interaction that should be executed.

In this sense, ECNO is deterministic. Choices between different valid interactions are made by the controllers – in our example, by the end user pressing buttons on the GUI. The only source of non-determinism within ECNO is the order of the execution of all the actions of the elements of the interaction. The idea of ECNO would be that every action makes only local changes in the element itself based on the parameters of the events and the local attributes. Under this assumption, also the result of the execution of an interaction would be deterministic. Since ECNO nets allow arbitrary Java code in the actions, the ECNO framework cannot guarantee that all changes in the actions are local, however. In the future, the ECNO framework could be equipped with a policy that enforces the locality of actions and a mechanism warning the modeller in case of potential non-determinism. What is reasonable and necessary, however, would require more examples and more realistic ones. This is why, for now, the modeller can do whatever he sees fit.

## 6 Related Work

The ideas for ECNO have developed over some years; they started out in the field of Business Process Modelling, where we used events and their synchronisation for identifying and formalising the basic concepts of business process models and their execution: AMFIBIA [3, 12]. It turned out that these ideas are much more general and do not only apply in the area of business processes. This generalisation resulted in MoDowA [4]. MoDowA, however, was tightly coupled to aspects, and event coordination was possible only for very specific types of relations. Therefore, the quest for distilling the basic coordination primitive was still on. In [2], we pointed out some first ideas for such an event coordination notation, which we call ECNO now.

Actually, none of the concepts of ECNO are particularly new or original. For example, Petri nets [6, 7] have been made exactly for the purpose of modelling behaviour. And many different mechanisms have been proposed for coordinating different Petri nets. For example, the box calculus and M-nets [13, 14] or an extension of CPNs [15] use synchronisation of transitions for coordinating different Petri nets. For most Petri net approaches, the synchronisation structure, however, is static and often between two partners only. Renew [16, 17] proposes a more general concept of executable Petri nets where the transitions can have several synchronous channels for synchronising different Petri nets – which are called up- and down-links. These references are more dynamic, following the structure of the underlying nets-within-nets paradigm [18]. This way, more than two partners could be synchronised and parameters could be passed

in both directions. In ECNO, the communication structure is not derived from the net structure. Instead, ECNO’s coordination diagrams exploit the dynamic structure of instances of a class diagram.

The idea of events and the way they are synchronised dates back even further. In the earlier work, they would rather be called actions, which should, however, not be confused with ECNO’s concept of actions. ECNO’s synchronisation mechanism resembles process algebras like CSP [19], CSS [20], the Chemical Abstract Machine [21] or the  $\pi$ -calculus [22]. But, ECNO is a bit more explicit with whom to synchronise and with how many partners; the partners with whom an element needs to synchronise can change dynamically dependent on the changing links between elements. In ECNO, there can be arbitrarily long chains or networks of required participants of an interaction. One approach that allows to define such interactions (via connectors) is BIP [23]; but ECNO embeds a bit smoother with class diagrams and allows for and is tuned to dynamically changing structures. And ECNO works together with classical programming. Other parts of the software can change the configuration as they please; ECNO’s notification and controller mechanisms will properly update the possible interactions. Moreover, ECNO’s actions can call methods of other parts of the software, and via an API, other parts of the software can initiate and hook into interactions.

The ideas of ECNO are an extension of our MoDowA approach (Modelling Domains with Aspects) [4], which has some similarities with the Theme approach [24]. In MoDowA, the interactions were restricted and implicitly defined for two special kinds of relations. In ECNO, this was generalised: we introduced a separate concept on top of references for making interactions explicit [5]; some of these ideas came up during the work on a masters thesis [25]. Technically, ECNO is independent from aspect oriented modelling. Still, it was inspired by aspect orientation and is close in spirit to aspect oriented modelling (see [26, 27] for an overview) or subject orientation [28]. Moreover, ECNO could serve as an underlying technology for implementing aspect oriented models: an aspects of an object could be represented by a reference from the object to the aspect, where the coordination annotations would define in which way the aspect joins into events executed by the object. In a way, events are *join points* and the interactions are *point cuts* as, for example, in AspectJ [29] in aspect oriented programming. There are two main differences though: in aspect oriented programming, the join points are defined in the final program; in that sense, the join points are programming artefacts and not domain concepts. By contrast, our events are concepts of the domain! The other difference is the more symmetric participation in an interaction. The participants in an interaction are not only invoked by another element; they can actively contribute parameters, and even prevent an interaction from happening (if no other partners are available). A more subtle difference is that interactions in our approach are attached to objects (actually, we called them elements) and not to lines in a program, and interactions can only be defined along existing links between the elements. This is a restriction – but a deliberate one: it provides more structure and avoids clutter and unexpected interactions between completely unrelated elements.



One of the main objectives of ECNO and its coordination diagrams is that the coordination of events should relate to the structural models (e.g. a class diagram). In this respect, ECNO is similar in spirit to Ptolemy II [30]. In a way, our coordination diagrams could be considered a specific configurable *model of computation* of Ptolemy. Our rationale is the smooth extension of existing notations and technologies and easing the integration of behaviour models with structural ones and with pre-existing software.

Whereas other approaches like Executable UML [31] try to obtain executable models by restricting UML, ECNO proposes an extension on top of class diagrams. What is more, the concept of interactions allows us to get rid of the thread oriented way of “thinking concurrency” – as pointed out in [32]. Interactions are concurrent in nature. In this paper, we did not focus on issues of behaviour inheritance; but, by attaching and synchronising different ECNO nets to the same element, different notions of behaviour inheritance as for example discussed in [33] could be implemented in ECNO.

To sum up, all the individual concepts of ECNO existed before – we did not invent them. The main contribution of ECNO is the combination of these concepts. This way, making them more usable in practical software development in general – and in model-based software engineering in particular. The prototype implementation of the ECNO execution engine shows that this combination of concepts can actually be made work.

## 7 Conclusion

In this paper, we have discussed the main concepts of the *Event Coordination Notation (ECNO)*. This notation allows defining the global behaviour of a system by coordinating local behaviour: the global coordination is defined on top of structural diagrams. In addition, we discussed a specific modelling notation for the local behaviour which we called ECNO nets. From these models, the code for the complete system including its behaviour can be generated. The example shows that the coordination mechanisms of ECNO for defining global behaviour together with the mechanisms for defining local behaviour are powerful enough to completely define a system and generate code for it (where the structural parts could even be generated by another technology – by EMF, in our example).

The most interesting research on ECNO, however, is yet to come: Methodology, adequateness of the modelling notation, scalability, and performance still need more investigation; the constructs need to be adjusted, so as to strike a balance between these different objectives. The prototype implementation discussed in this paper, lays the foundation for these investigations.

**Acknowledgements.** The ideas of ECNO gradually evolved over many years and many people have directly or indirectly contributed to it. A list of these people can be found at <http://www2.imm.dtu.dk/~eki/projects/ECNO/>. David Schmelter and Steve Hostettler have helped improving the presentation of this paper with their comments and suggestions on earlier versions of this paper. I would like to thank all of them.

## References

1. OMG: MDA guide (2003), <http://www.omg.org/cgi-bin/doc?omg/03-06-01>
2. Kindler, E.: Model-based software engineering: The challenges of modelling behaviour. In: Aksit, M., Kindler, E., Roubtsova, E., McNeile, A. (eds.) Proceedings of the Second Workshop on Behavioural Modelling - Foundations and Application, BM-FA 2010, pp. 51–66 (2010)
3. Axenath, B., Kindler, E., Rubin, V.: AMFIBIA: A meta-model for the integration of business process modelling aspects. *International Journal on Business Process Integration and Management* 2(2), 120–131 (2007)
4. Kindler, E., Schmelter, D.: Aspect-oriented modelling from a different angle: Modelling domains with aspects. In: 12th International Workshop on Aspect-Oriented Modeling (2008)
5. Kindler, E.: Integrating behaviour in software models: An event coordination notation – concepts and prototype. In: Proceedings of the Third Workshop on Behavioural Modelling - Foundations and Application, BM 2011 (2011)
6. Reisig, W.: Petri Nets. EATCS Monographs on Theoretical Computer Science, vol. 4. Springer (1985)
7. Murata, T.: Petri nets: Properties, analysis and applications. *Proceedings of the IEEE* 77, 541–580 (1989)
8. Kindler, E.: The ePNK: An Extensible Petri Net Tool for PNML. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 318–327. Springer, Heidelberg (2011)
9. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework, 2nd edn. The Eclipse Series. Addison-Wesley (2006)
10. Harel, D., Marely, R.: Come let's play: Scenario-based programming using LSCs and the Play-engine. Springer (2003)
11. Kindler, E.: The Event Coordination Notation: Execution engine and programming framework. In: Fourth Workshop on Behavioural Modelling – Foundations and Application, BM-FA 2012, ECMFA 2012 Joint Proceedings of Co-located Events, pp. 143–157 (2012)
12. Axenath, B., Kindler, E., Rubin, V.: An open and formalism independent meta-model for business processes. In: Kindler, E., Nüttgens, M. (eds.) Workshop on Business Process Reference Models 2005, BPRM 2005, Satellite Event of the Third International Conference on Business Process Management, pp. 45–59 (2005)
13. Best, E., Devillers, R., Hall, F.: The Box Calculus: A New Causal Algebra with Multi-Label Communication. In: Rozenberg, G. (ed.) APN 1992. LNCS, vol. 609, pp. 21–69. Springer, Heidelberg (1992)
14. Best, E., Fraczak, W., Hopkins, R.P., Klaudel, H., Pelz, E.: M-nets: An algebra of high-level Petri nets, with an application to the semantics of concurrent programming languages. *Acta Inf.* 35(10), 813–857 (1998)
15. Christensen, S., Hansen, N.D.: Coloured Petri Nets Extended with Channels for Synchronous Communication. In: Valette, R. (ed.) ICATPN 1994. LNCS, vol. 815, pp. 159–178. Springer, Heidelberg (1994)
16. Kummer, O.: A Petri net view on synchronous channels. *Petri Net Newsletter* 56, 7–11 (1999)
17. Kummer, O., Wienberg, F., Duvigneau, M., Schumacher, J., Köhler, M., Moldt, D., Rölke, H., Valk, R.: An Extensible Editor and Simulation Engine for Petri Nets: RENEW. In: Cortadella, J., Reisig, W. (eds.) ICATPN 2004. LNCS, vol. 3099, pp. 484–493. Springer, Heidelberg (2004)

18. Valk, R.: Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In: Desel, J., Silva, M. (eds.) ICATPN 1998. LNCS, vol. 1420, pp. 1–25. Springer, Heidelberg (1998)
19. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
20. Milner, R.: Communication and Concurrency. International Series in Computer Science. Prentice Hall (1989)
21. Berry, G., Boudol, G.: The chemical abstract machine. In: POPL 1990, pp. 81–94 (1990)
22. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes (Parts I & II). *Information and Computation* 100(1), 1–40, 41–77 (1992)
23. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Fourth IEEE International Conference on Software Engineering and Formal Methods, pp. 3–12. IEEE Computer Society (2006)
24. Clarke, S., Baniassad, E.: Aspect-oriented analysis and design: The Theme approach. Addison-Wesley (2005)
25. Nowak, L.: Aspect-oriented modelling of behaviour – implementation of an execution engine based on MoDowA. Master’s thesis, DTU Informatics (2009)
26. Brichau, J., Haupt, M.: Survey of aspect-oriented languages and execution models. Technical Report AOSD-Europe-VUB-01, AOSD-Europe (2005)
27. Chitchyan, R., Rashid, A., Sawyer, P., Garcia, A., Alarcon, M.P., Bakker, J., Tekinerdogan, B., Clarke, S., Jackson, A.: Survey of aspect-oriented analysis and design approaches. Technical Report AOSD-Europe-ULANC-9, AOSD-Europe (2005)
28. Harrison, W., Ossher, H.: Subject-oriented programming (a critique of pure objects). In: OOPSLA 1993, pp. 411–428. ACM (1993)
29. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Lindskov Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
30. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neundorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE* 91(1), 127–144 (2003)
31. Mellor, S.J., Balcer, M.J.: Executable UML: A Foundation for Model-driven Architecture. Addison-Wesley (2002)
32. Lee, E.A.: The problem with threads. *IEEE Computer* 39(5), 33–42 (2006)
33. McNeile, A.T., Simons, N.: State machines as mixins. *Journal of Object Technology* 2(6), 85–101 (2003)

# Model Checking Using Generalized Testing Automata

Ala-Eddine Ben Salem<sup>1,2</sup>, Alexandre Duret-Lutz<sup>1</sup>, and Fabrice Kordon<sup>2</sup>

<sup>1</sup> LRDE, EPITA, Le Kremlin-Bicêtre, France  
{ala,adl}@lrde.epita.fr

<sup>2</sup> LIP6, CNRS UMR 7606, Université P. & M. Curie – Paris 6, France  
Fabrice.Kordon@lip6.fr

**Abstract.** Geldenhuys and Hansen showed that a kind of  $\omega$ -automata known as *Testing Automata* (TA) can, in the case of stuttering-insensitive properties, outperform the Büchi automata traditionally used in the automata-theoretic approach to model checking [10].

In previous work [23], we compared TA against *Transition-based Generalized Büchi Automata* (TGBA), and concluded that TA were more interesting when counterexamples were expected, otherwise TGBA were more efficient.

In this work we introduce a new kind of automata, dubbed *Transition-based Generalized Testing Automata* (TGTA), that combine ideas from TA and TGBA. Implementation and experimentation of TGTA show that they outperform other approaches in most of the cases.

**Keywords:** testing automata, model checking, emptiness check.

## 1 Introduction

**Context.** The automata-theoretic approach to model checking linear-time properties [28] splits the verification process into four operations:

1. Computation of the state-space for the model  $M$ . This state-space can be seen as an  $\omega$ -automaton  $A_M$  whose language,  $\mathcal{L}(A_M)$ , represents all possible infinite executions of  $M$ .
2. Translation of the temporal property  $\varphi$  into an  $\omega$ -automaton  $A_{\neg\varphi}$  whose language,  $\mathcal{L}(A_{\neg\varphi})$ , is the set of all infinite executions that would invalidate  $\varphi$ .
3. Synchronization of these automata. This constructs a product automaton  $A_M \otimes A_{\neg\varphi}$  whose language,  $\mathcal{L}(A_M) \cap \mathcal{L}(A_{\neg\varphi})$ , is the set of executions of  $M$  invalidating  $\varphi$ .
4. Emptiness check of this product. This operation tells whether  $A_M \otimes A_{\neg\varphi}$  accepts an infinite word, and can return such a word (a counterexample) if it does. The model  $M$  verifies  $\varphi$  iff  $\mathcal{L}(A_M \otimes A_{\neg\varphi}) = \emptyset$ .

**Problem.** Different kinds of  $\omega$ -automata have been used with the above approach. In the most common case, a property expressed as an LTL (linear-time temporal logic) formula is converted into a Büchi automaton with state-based acceptance, and a Kripke structure is used to represent the state-space of the model.

In Spot [17], our model checking library, we prefer to represent properties using *generalized* (i.e., multiple) Büchi acceptance conditions *on transitions* rather than on states [7]. Any algorithm that translates LTL into a Büchi automaton has to deal with

generalized Büchi acceptance conditions at some point, and the process of *degeneralizing* the Büchi automaton often increases its size. Several emptiness-check algorithms can deal with generalized Büchi acceptance conditions, making such an a degeneralization unnecessary and even costly [5]. Moving the acceptance conditions from the states to the transitions also reduces the size of the property automaton [4, 13].

Unfortunately, having a smaller property automaton  $A_{\neg\phi}$  does not always imply a smaller product with the model ( $A_M \otimes A_{\neg\phi}$ ), and the size of this product really affects model checking efficiency. Thus, instead of targeting smaller property automata, some people have attempted to build automata that are *more deterministic* [25]. However even this does not guarantee the product to be smaller.

Hansen et al. [14] introduced a new kind of  $\omega$ -automaton called *Testing Automaton* (TA). These automata are less expressive than Büchi automata since are tailored to represent *stuttering-insensitive* properties (such as any LTL property that does not use the X operator). They are often a lot larger than their equivalent Büchi automaton, but surprisingly, their high degree of determinism often lead to a smaller product [10]. As a back-side, TA have two different modes of acceptance (Büchi-acceptance or livelock-acceptance), and their emptiness check may require two passes, mitigating the benefits of a having a smaller product.

**Objectives.** The study of Geldenhuys and Hansen [10] shows TA are statistically more efficient than Büchi automata. In a previous work [23], we have extended their comparison to TGBA, and shown that TA are indeed better when the formula to be verified is violated (i.e., a counterexample is found), but this is not the case when the property is verified since the entire state space may have to be visited twice to check for each acceptance mode of a TA.

This paper introduces a new type of  $\omega$ -automata, Transition-based Generalized Testing Automata (TGTA), that mixes features from both TA and TGBA. From TA, it reuses the labeling of transitions with changesets, and simplifications based on stuttering. From TGBA, it inherits the use of transition-based acceptance conditions. TGTA combine the advantages of TA and TGBA: it is still statistically more efficient than other  $\omega$ -automata when the property is violated but does not require a second pass when no counterexample is found, thus remaining more efficient than other  $\omega$ -automata in that situation.

We have implemented this new approach in Spot. This required little effort since TGTA reuse the emptiness check algorithm of TGBA. We are thus able to compare TGTA with the “traditional” algorithms we used on TA, BA and TGBA. These experiments show that TGTA compete well on our examples.

**Contents.** Section 2 provides a brief summary of the three  $\omega$ -automaton (BA, TGBA and TA) and pointers to their associated operations for model checking before Section 3 presents TGTA. Section 4 reports our experiments before a discussion in Section 5.

## 2 Presentation of Three Existing Approaches

Let  $AP$  designate the set of *atomic proposition* of the model. We use  $AP$  to build a linear-time property. Any state of the model is labeled by a valuation of these atomic propositions. We denote by  $\Sigma = 2^{AP}$  the set of these valuations, which we interpret either as a set or as Boolean conjunctions. For instance if  $AP = \{a, b\}$ , then  $\Sigma = 2^{AP} =$

$\{\{a, b\}, \{a\}, \{b\}, \emptyset\}$  but we equivalently interpret it as  $\Sigma = \{ab, a\bar{b}, \bar{a}b, \bar{a}\bar{b}\}$ . All the executions of the model can be represented by a Kripke structure  $\mathcal{K}$ . An execution of the model is simply an infinite sequence of such valuations, i.e., an element from  $\Sigma^\omega$ .

**Definition 1.** A Kripke structure over the alphabet  $\Sigma = 2^{AP}$  is a tuple  $\mathcal{K} = \langle S, I, R, L \rangle$  where:

- $S$  is a finite set of states,
- $I \subseteq S$  is the set of initial states,
- $R \subseteq S \times S$  is the transition relation,
- $L : S \rightarrow \Sigma$  is a state-labeling function.

An execution  $w = k_0k_1k_2 \dots \in \Sigma^\omega$  is accepted by  $\mathcal{K}$  if there exists an infinite sequence  $s_0, s_1, \dots \in S^\omega$  such that  $s_0 \in I$  and  $\forall i \in \mathbb{N}, L(s_i) = k_i \wedge (s_i, s_{i+1}) \in R$ . The language accepted by  $\mathcal{K}$  is the set  $\mathcal{L}(\mathcal{K}) \subseteq K^\omega$  of executions it accepts.

A property can be seen as a set of sequences, i.e., a subset of  $\Sigma^\omega$ . Among these properties, we want to distinguish those that are *stuttering-insensitive*:

**Definition 2.** A property, or a language, i.e., a set of infinite sequences  $\mathcal{P} \subseteq \Sigma^\omega$ , is stuttering-insensitive iff any sequence  $k_0k_1k_2 \dots \in \mathcal{P}$  remains in  $\mathcal{P}$  after repeating any valuation  $k_i$  or omitting duplicate valuations. Formally,  $\mathcal{P}$  is stuttering-insensitive iff

$$k_0k_1k_2 \dots \in \mathcal{P} \iff k_0^{i_0}k_1^{i_1}k_2^{i_2} \dots \in \mathcal{P} \text{ for any } i_0 > 0, i_1 > 0 \dots$$

**Theorem 1.** Any  $LTL \setminus X$  formula (i.e., an LTL formula that does not use the  $X$  operator) describes a stuttering-insensitive property. Conversely any stuttering-insensitive property can be expressed as an  $LTL \setminus X$  formula [19].

The following sections presents three kinds of  $\omega$ -automata [8] that can be used to express properties in the automata-theoretic approach to model checking. *Transition-based Generalized Büchi Automata* and *Büchi Automata* can both express general properties, while *Testing Automata* are tailored to stuttering-insensitive properties.

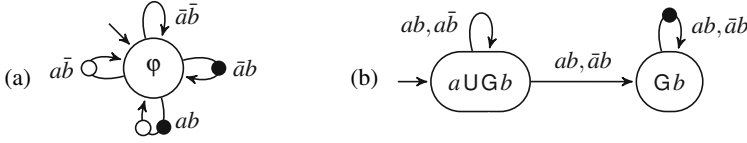
## 2.1 Transition-Based Generalized Büchi Automata

We begin by defining Transition-based Generalized Büchi Automata (TGBA), which are a generalization of the better known Büchi automata used for model checking [13]. In our context, the TGBA represents the negation of the LTL property to verify.

**Definition 3.** A TGBA over the alphabet  $\Sigma = 2^{AP}$  is a tuple  $\mathcal{G} = \langle S, I, R, F \rangle$  where:

- $S$  is a finite set of states,
- $I \subseteq S$  is the set of initial states,
- $F$  is a finite set of acceptance conditions,
- $R \subseteq S \times 2^\Sigma \times 2^F \times S$  is the transition relation, where each element  $(s_i, K_i, F_i, d_i)$  represents a transition from state  $s_i$  to state  $d_i$  labeled by the non-empty set of valuation  $K_i$ , and a set of acceptance conditions  $F_i$ .

An execution  $w = k_0k_1k_2 \dots \in \Sigma^\omega$  is accepted by  $\mathcal{G}$  if there exists an infinite path  $(s_0, K_0, F_0, s_1)(s_1, K_1, F_1, s_2)(s_2, K_2, F_2, s_3) \dots \in R^\omega$  where:



**Fig. 1.** (a) A TGBA with acceptance conditions  $F = \{\bullet, \circ\}$  recognizing the LTL property  $\varphi = GFa \wedge GFb$ . (b) A TGBA with  $F = \{\bullet\}$  recognizing the LTL property  $aUGb$ .

- $s_0 \in I$ , and  $\forall i \in \mathbb{N}, k_i \in K_i$  (the execution is recognized by the path),
- $\forall f \in F, \forall i \in \mathbb{N}, \exists j \geq i, f \in F_j$  (each acceptance condition is visited infinitely often).

The language accepted by  $\mathcal{G}$  is the set  $\mathcal{L}(\mathcal{G}) \subseteq \Sigma^\omega$  of executions it accepts.

Any LTL formula  $\varphi$  can be converted into a TGBA whose language is the set of executions that satisfy  $\varphi$ . Several algorithms exist to translate an LTL formula into a TGBA [4, 9, 13, 1].

Figure 1 shows two examples of TGBA: one deterministic TGBA derived from the LTL formula  $GFa \wedge GFb$ , and one non-deterministic TGBA derived from  $aUGb$ . The LTL formulae that label states represent the property accepted starting from this state of the automaton: they are shown for the reader's convenience but not used for model checking. As can be inferred from Fig. 1(a), an LTL formula such as  $\bigwedge_{i=1}^n GF p_i$  can be represented by a one-state deterministic TGBA with  $n$  acceptance conditions.

Any infinite path in these examples is accepted if it visits all acceptance conditions (represented by colored dots on the transitions) infinitely often.

Testing a TGBA for emptiness amounts to the search of a strongly connected component that contains at least one occurrence of each acceptance condition. This can be done in different ways [5]. We are using Couvreur's SCC-based emptiness check algorithm [4] because it needs to traverse the state-space only once, and its complexity does not depend on the number of acceptance conditions. This algorithm is detailed in Appendix B.

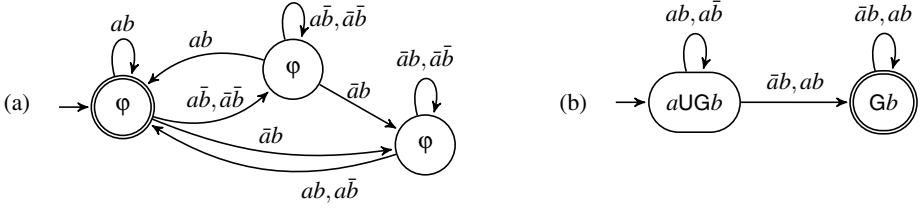
## 2.2 Büchi Automata

Compared to TGBA, the more traditional Büchi Automata (BA) have only one state-based acceptance condition.

One common way to obtain a BA from an LTL formula is to first translate the formula into some Generalized Büchi Automata with multiple acceptance conditions (it could be a TGBA [13, 9] or a state-based GBA [12]) and then to *degenerate* this automaton to obtain a single acceptance condition.

**Definition 4.** A BA over the alphabet  $\Sigma = 2^{AP}$  is a tuple  $\mathcal{B} = \langle S, I, R, F \rangle$  where:

- $S$  is a finite set states,
- $I \subseteq S$  is the set of initial states,
- $F \subseteq S$  is a finite set of acceptance states,
- $R \subseteq S \times 2^\Sigma \times S$  is the transition relation where each transition is labeled by a set of valuations.



**Fig. 2.** Two examples of BA, with acceptance states shown as double circles. (a) A BA for the LTL property  $\varphi = GF a \wedge GF b$  obtained by degeneralizing the TGBA for Fig. 1(a). (b) A BA for the LTL property  $aUGb$ .

An execution  $w = k_0 k_1 k_2 \dots \in \Sigma^\omega$  is accepted by  $\mathcal{B}$  if there exists an infinite path  $(s_0, K_0, s_1)(s_1, K_1, s_2)(s_2, K_2, s_3) \dots \in R^\omega$  such that:

- $s_0 \in I$ , and  $\forall i \in \mathbb{N}, k_i \in K_i$  (the execution is recognized by the path),
- $\forall i \in \mathbb{N}, \exists j \geq i, s_j \in F$  (at least one acceptance state is visited infinitely often).

The language accepted by  $\mathcal{B}$  is the set  $\mathcal{L}(\mathcal{B}) \subseteq \Sigma^\omega$  of executions it accepts.

**Theorem 2.** TGBA and BA have the same expressive power: any TGBA can be converted into a language-equivalent BA and vice-versa [4, 13].

The process of converting a TGBA into a BA is called *degeneralization*. In the worst case, a TGBA with  $s$  states and  $n$  acceptance conditions will be degeneralized into a BA with  $s \times (n + 1)$  states.

Figure 2 shows the same properties as Fig. 1 but expressed as Büchi automata. The automaton from Fig. 2(a) was built by degeneralizing the TGBA from Fig. 1(a). The worst case of the degeneralization occurred here, since the TGBA with 1 state and  $n$  acceptance conditions was degeneralized into a BA with  $n + 1$  states. It is known that no BA with less than  $n + 1$  states can accept the property  $\bigwedge_{i=1}^n GF p_i$  so this Büchi automaton is optimal [3]. The property  $aUGb$ , on the right hand side of the figure, is easier to express: the BA has the same size as the TGBA.

In the other way, a BA can be seen as a TGBA, by simply marking transitions leaving acceptance states as accepting, without adding states nor transitions. Algorithms that input TGBA can therefore be easily adapted to process BA. More importantly, BA can be checked for emptiness using the same one-pass emptiness-check algorithm.

### 2.3 Testing Automata

Testing Automata (TA) were introduced by Hansen et al. [14] to represent stuttering-insensitive properties. While a Büchi automaton observes the value of the atomic propositions  $AP$ , the basic idea of TA is to detect the *changes* in these values; if a valuation of  $AP$  does not change between two consecutive valuations of an execution, the TA can stay in the same state. To detect infinite executions that end stuck in the same TA state because they are stuttering, a new kind of acceptance states is introduced: *livelock-acceptance states*.

If  $A$  and  $B$  are two valuations,  $A \oplus B$  denotes the symmetric set difference, i.e., the set of atomic propositions that differ (e.g.,  $a\bar{b} \oplus ab = \{b\}$ ). Technically, this is implemented with an XOR operation (also denoted by the symbol  $\oplus$ ).



**Definition 5.** A TA over the alphabet  $\Sigma = 2^{AP}$  is a tuple  $\mathcal{T} = \langle S, I, U, R, F, G \rangle$ , where:

- $S$  is a finite set of states,
- $I \subseteq S$  is the set of initial states,
- $U : I \rightarrow 2^\Sigma$  is a function mapping each initial state to a set of valuations (set of possible initial configurations),
- $R \subseteq S \times \Sigma \times S$  is the transition relation where each transition  $(s, k, d)$  is labeled by a changeset:  $k \in \Sigma$  is interpreted as a (possibly empty) set of atomic propositions whose value must change between states  $s$  and  $d$ ,
- $F \subseteq S$  is a set of Büchi-acceptance states,
- $G \subseteq S$  is a set of livelock-acceptance states.

An execution  $w = k_0 k_1 k_2 \dots \in \Sigma^\omega$  is accepted by  $\mathcal{T}$  if there exists an infinite sequence  $(s_0, k_0 \oplus k_1, s_1)(s_1, k_1 \oplus k_2, s_2) \dots (s_i, k_i \oplus k_{i+1}, s_{i+1}) \dots \in (S \times \Sigma \times S)^\omega$  such that:

- $s_0 \in I$  with  $k_0 \in U(s_0)$ ,
- $\forall i \in \mathbb{N}$ , either  $(s_i, k_i \oplus k_{i+1}, s_{i+1}) \in R$  (the execution progresses in the TA), or  $k_i = k_{i+1} \wedge s_i = s_{i+1}$  (the execution is stuttering and the TA does not progress),
- Either,  $\forall i \in \mathbb{N}$ ,  $(\exists j \geq i, k_j \neq k_{j+1}) \wedge (\exists l \geq i, s_l \in F)$  (the TA is progressing in a Büchi-accepting way), or,  $\exists n \in \mathbb{N}$ ,  $(s_n \in G \wedge (\forall i \geq n, s_i = s_n \wedge k_i = k_n))$  (the sequence reaches a livelock-acceptance state and then stays on that state because the execution is stuttering).

The language accepted by  $\mathcal{T}$  is the set  $\mathcal{L}(\mathcal{T}) \subseteq \Sigma^\omega$  of executions it accepts.

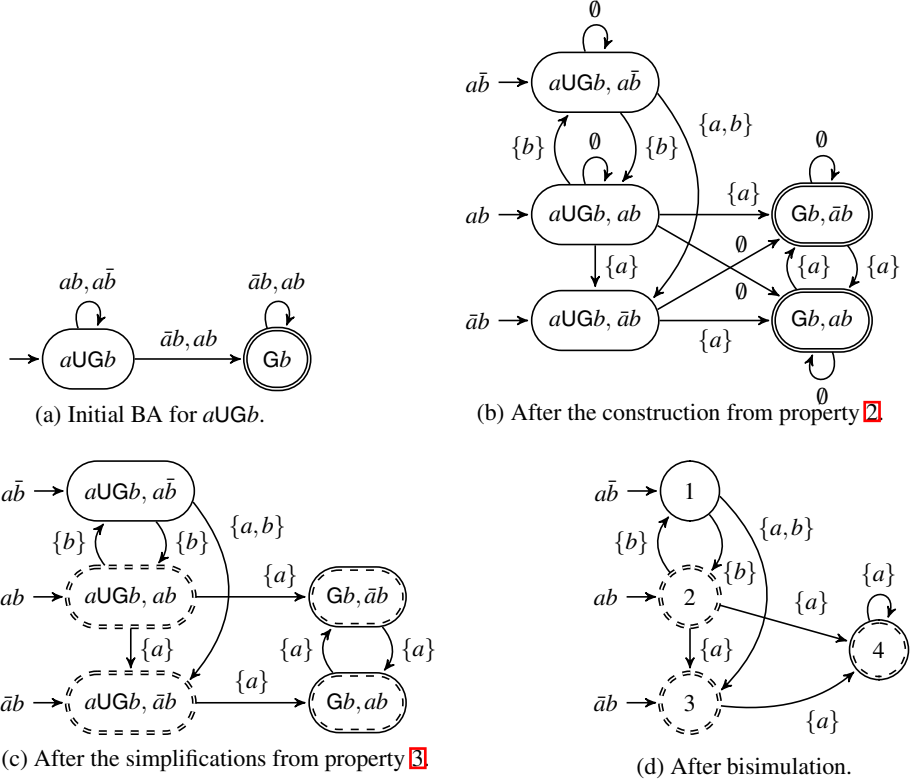
To illustrate this definition, consider Fig. 3d, representing a TA for aUGb.

- The execution  $ab; \bar{a}b; ab; \bar{a}b; ab; \bar{a}b; ab; \dots$  is Büchi accepting. A run recognizing such an execution must start in state 2, then it always changes the value of  $a$ , so it has to take transitions labeled by  $\{a\}$ . For instance it could be the run  $2 \xrightarrow{\{a\}} 4 \xrightarrow{\{a\}} 4 \xrightarrow{\{a\}} 4 \dots$  or the run  $2 \xrightarrow{\{a\}} 3 \xrightarrow{\{a\}} 4 \xrightarrow{\{a\}} 4 \dots$ . Both visit the run state  $4 \in F$  infinitely often, so they are Büchi accepting.
- The execution  $ab; \bar{a}b; \bar{a}b; \bar{a}b; \dots$  is livelock accepting. An accepting run starts in state 2, then moves to state 4, and stutters on this livelock-accepting state. Another possible accepting run goes from state 2 to state 3 and stutters in  $3 \in G$ .
- The execution  $ab; \bar{a}\bar{b}; ab; \bar{a}\bar{b}; ab; \bar{a}\bar{b}; \dots$  is not accepted. It would correspond to a run alternating between states 2 and 1, but such a run is neither Büchi accepting (does not visit any  $F$  state) nor livelock-accepting (it passes through state  $2 \in G$ , but does not stay into this state continuously).

**Property 1.** The language accepted by a testing automaton is stuttering-insensitive.

*Proof.* This follows from definition of accepted executions: a TA may not change its state when an execution stutters, so stuttering is always possible.  $\square$

**Construction of a Testing Automaton from a Büchi Automaton.** Geldenhuis and Hansen [10] have shown how to convert a BA into a TA by first converting the BA into an automaton with valuations on the states, and then converting this automaton into a TA by computing the difference between the labels of the source and destination of each transition. The next proposition implements these first two steps at once.



**Fig. 3.** Steps of the construction of a TA from a BA. States with a double enclosure belong to either  $F$  or  $G$ : states in  $F \setminus G$  have a double plain line, states in  $G \setminus F$  have a double dashed line, and states in  $F \cap G$  use a mixed dashed/plain style.

**Property 2 (Converting a BA into a TA [10]).** For any BA  $\mathcal{B} = \langle S_{\mathcal{B}}, I_{\mathcal{B}}, R_{\mathcal{B}}, F_{\mathcal{B}} \rangle$  over the alphabet  $\Sigma = 2^{A^P}$  and such that  $\mathcal{L}(\mathcal{B})$  is stuttering insensitive, let us define the TA  $\mathcal{T} = \langle S_{\mathcal{T}}, I_{\mathcal{T}}, U_{\mathcal{T}}, R_{\mathcal{T}}, F_{\mathcal{T}}, \emptyset \rangle$  with  $S_{\mathcal{T}} = S_{\mathcal{B}} \times \Sigma$ ,  $I_{\mathcal{T}} = I_{\mathcal{B}} \times \Sigma$ ,  $F_{\mathcal{T}} = F_{\mathcal{B}} \times \Sigma$  and

- $\forall (s, k) \in I_{\mathcal{T}}, U_{\mathcal{T}}((s, k)) = \{k\}$
- $\forall (s, k) \in S_{\mathcal{T}}, \forall (s', k') \in S_{\mathcal{T}},$   
 $((s, k), k \oplus k', (s', k')) \in R_{\mathcal{T}} \iff \exists K \in 2^{\Sigma}, ((s, K, s') \in R_{\mathcal{B}}) \wedge (k \in K)$

Then  $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{T})$ .

Figure 3b shows the result of applying this construction to the example Büchi automaton shown for  $aUGb$ . This testing automaton does not yet use livelock-acceptance states (the  $G$  set). The next property, again from Geldenhuys and Hansen [10], shows how filling  $G$  allows to remove all transitions labeled by  $\emptyset$ .

**Property 3 (Using  $G$  to simplify a TA [10]).** Let  $\mathcal{T} = \langle S, I, U, R, F, G \rangle$  be TA. By combining the first three of the following operations we can remove all transitions of the form  $(s, \emptyset, s')$  (i.e. stuttering-transitions) from a TA. The fourth simplification can be performed along the way.

1. If  $Q \subseteq S$  is a Strongly Connected Component (SCC) such that  $Q \cap F \neq \emptyset$  (it is Büchi accepting), and any two states  $q, q' \in Q$  can be connected using a non-empty sequence of stuttering-transitions  $(q, \emptyset, q_1) \cdot (q_1, \emptyset, q_2) \cdots (q_n, \emptyset, q') \in R^*$ , then the testing automaton  $\mathcal{T}' = \langle S, I, U, F, G \cup Q \rangle$  is such that  $\mathcal{L}(\mathcal{T}') = \mathcal{L}(\mathcal{T})$ . Such a component  $Q$  is called an **accepting Stuttering-SCC**.
2. If there exists a transition  $(s, \emptyset, s') \in R$  such that  $s' \in G$ , the automaton  $\mathcal{T}'' = \langle S, I, R \setminus \{(s, \emptyset, s')\}, F, G \cup \{s\} \rangle$  is such that  $\mathcal{L}(\mathcal{T}'') = \mathcal{L}(\mathcal{T})$ .
3. If  $\mathcal{T}$  does not contain any accepting Stuttering-SCC, and there exists a transition  $(s, \emptyset, s') \in R$  such that  $s'$  cannot reach any state from  $G$  using only transitions labeled by  $\emptyset$ , then these transitions can be removed. I.e., the automaton  $\mathcal{T}''' = \langle S, I, R \setminus \{(s, \emptyset, s')\}, F, G \rangle$  is such that  $\mathcal{L}(\mathcal{T}''') = \mathcal{L}(\mathcal{T})$ .
4. Any state from which one cannot reach a Büchi-accepting cycle nor a livelock-acceptance state can be removed without changing the automaton's language.

The resulting TA can be further simplified by merging bisimilar states (two states are bisimilar if the automaton can accept the same executions starting for either of these states). This can be achieved using any algorithm based on partition refinement [e.g., 27], taking  $\{F \cap G, F \setminus G, G \setminus F, S \setminus (F \cup G)\}$  as initial partition.

Fig. 3 shows how a BA denoting the LTL formula  $aUGb$  is transformed into a TA by applying prop. 2 prop. 3 and finally merging bisimilar states.

A TA for  $\mathbf{GF}a \wedge \mathbf{GF}b$  is too big to be shown: even after simplifications it has 11 states and 64 transitions.

An unfortunate consequence of having two different ways of accepting executions (livelock or Büchi), is that the emptiness-check algorithm required during model checking must perform two passes on the whole state space in the worst case. Geldenhuys and Hansen [10] have devised a heuristic that often renders the second pass useless when the formula is violated. Another optimization we present in Appendix D is to omit the second pass when no livelock-accepting states is encountered during the first pass.

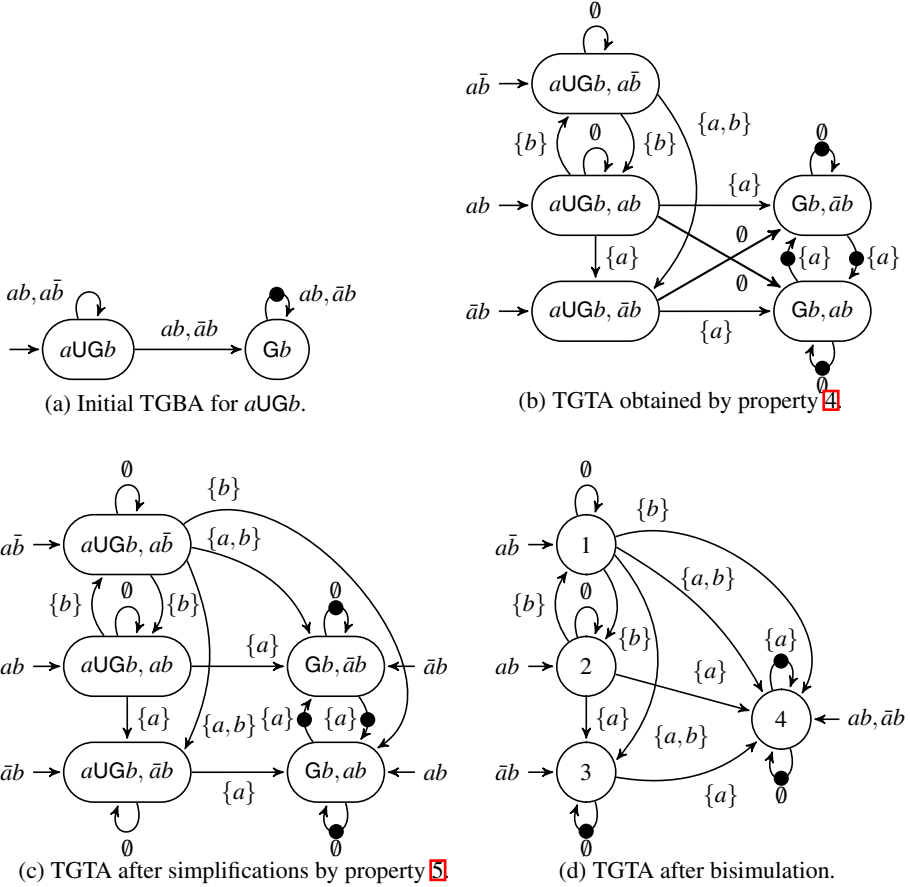
### 3 Transition-Based Generalized Testing Automata

This section introduces a new kind of automaton that combines features from both TA and TGBA. From TA, we take the idea of labeling transitions with changesets, however we remove the use of livelock-acceptance (because it may require a two-pass emptiness check), and the implicit stuttering. From TGBA, we inherit the use of transition-based generalized acceptance conditions.

The resulting Chimera, which we call *Transition-based Generalized Testing Automaton* (TGTA), accepts only stuttering-insensitive languages like TA, and inherits advantages from both TA and TGBA: it has a simple one-pass emptiness-check procedure (the same as the one for TGBA), and can benefit from reductions based on the stuttering of the properties pretty much like a TA. Livelock acceptance states, which are no longer supported, can be emulated using states with a Büchi accepting self-loop labeled by  $\emptyset$ .

**Definition 6.** A TGTA over the alphabet  $\Sigma$  is a tuple  $\mathcal{T} = \langle S, I, U, R, F \rangle$  where:

- $S$  is a finite set of states,



**Fig. 4.** TGTA obtained after various steps while translating the TGBA representing  $aUGb$ , into a TGTA with  $F = \{\bullet\}$

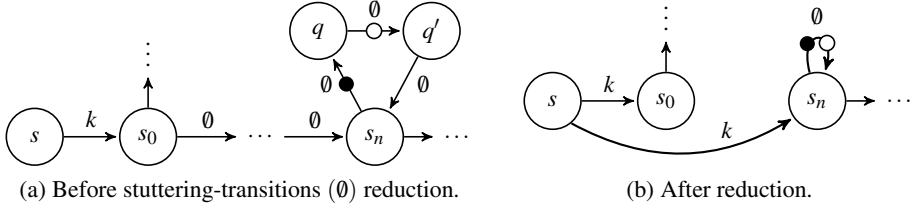
- $I \subseteq S$  is the set of initial states,
- $U : I \rightarrow 2^\Sigma$  is a function mapping each initial state to a set of symbols of  $\Sigma$
- $F$  is a finite set of acceptance conditions,
- $R \subseteq S \times \Sigma \times 2^F \times S$  is the transition relation, where each element  $(s_i, k_i, F_i, d_i)$  represents a transition from state  $s_i$  to state  $d_i$  labeled by a changeset  $k_i$ ; interpreted as a (possibly empty) set of atomic propositions whose value must change between states  $s_i$  and  $d_i$ , and the set of acceptance conditions  $F_i$ .

An execution  $w = k_0k_1k_2\dots \in \Sigma^\omega$  is accepted by  $\mathcal{T}$  if there exists an infinite path  $(s_0, k_0 \oplus k_1, F_0, s_1)(s_1, k_1 \oplus k_2, F_1, s_2)(s_2, k_2 \oplus k_3, F_2, s_3)\dots \in R^\omega$  where:

- $s_0 \in I$  with  $k_0 \in U(s_0)$  (the execution is recognized by the path),
- $\forall f \in F, \forall i \in \mathbb{N}, \exists j \geq i, f \in F_j$  (each acceptance condition is visited infinitely often).

The language accepted by  $\mathcal{T}$  is the set  $\mathcal{L}(\mathcal{T}) \subseteq \Sigma^\omega$  of executions it accepts.

Figure 4d shows a TGTA constructed for  $aUGb$  in the same way as we did for Fig. 3d. The only accepting runs are those that see  $\bullet$  infinitely often. The reader can verify that



**Fig. 5.** Using stuttering-transitions to simplify a TGTA

all the executions taken as example in section 2.3 are still accepted, but not always with the same runs (for instance  $ab; \bar{a}b; \bar{a}b; \bar{a}b; \dots$  is accepted by the run  $2, 4, 4, 4, \dots$ , but not by the run  $2, 3, 3, 3, \dots$ ). This small difference is due to the way we emulate livelock-accepting states, as we shall describe later (in Property 5).

**Construction of a TGTA from a TGBA.** We now describe how to build a TGTA starting from a TGBA. The construction is inspired by the one presented in section 2.3 to construct a TA from a BA. In future work we plan to implement a direct translation from LTL to TGTA, but the construction presented below is enough to show the benefits of using TGTAs, and makes it easier to understand how TGTAs relates from TGBA.

Our first property is the counterpart of Prop. 2: we can construct a TGTA from a TGBA by moving labels to states, and labeling each transition by the set difference between the labels of its source and destination states. While doing so, we keep the generalized acceptance conditions on the transitions. An example is shown on Fig 4b.

**Property 4 (Converting TGBA into TGTA).** For any TGBA  $\mathcal{G} = \langle S_{\mathcal{G}}, I_{\mathcal{G}}, R_{\mathcal{G}}, F \rangle$  over the alphabet  $\Sigma = 2^A$  and such that  $\mathcal{L}(\mathcal{G})$  is stuttering insensitive, let us define the TGTA  $\mathcal{T} = \langle S_{\mathcal{T}}, I_{\mathcal{T}}, U_{\mathcal{T}}, R_{\mathcal{T}}, F \rangle$  with  $S_{\mathcal{T}} = S_{\mathcal{G}} \times \Sigma$ ,  $I_{\mathcal{T}} = I_{\mathcal{G}} \times \Sigma$  and

- (i)  $\forall (s, k) \in I_{\mathcal{T}}, U_{\mathcal{T}}((s, k)) = \{k\}$
- (ii)  $\forall (s, k) \in S_{\mathcal{T}}, \forall (s', k') \in S_{\mathcal{T}},$   
 $((s, k), k \oplus k', F_1, (s', k')) \in R_{\mathcal{T}} \iff \exists K \in 2^{\Sigma}, ((s, K, F_1, s') \in R_{\mathcal{G}}) \wedge (k \in K)$

Then  $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{T})$ . (See appendix E for a proof.)

The next property is the pendent of Prop. 3 to simplify the automaton by removing stuttering-transitions. Here we cannot remove self-loop transitions labeled by  $\emptyset$ , but we can remove all others. The intuition behind this simplification is illustrated on Fig 5a:  $s_0$  is reachable from state  $s$  by a non-stuttering transition, but  $s_0$  can reach an accepting stuttering-cycle by following only stuttering transitions. In the context of TA we would have to declare  $s_0$  as being a livelock-accepting state. For TGTA, we replace the accepting stuttering-cycle by adding a self-loop labeled by all acceptance conditions on  $s_n$ , then the predecessors of  $s_0$  are connected to  $s_n$  as in Fig. 5b.

**Property 5 (Using stuttering-transitions to simplify a TGTA).** Let  $\mathcal{T} = \langle S, I, U, R, F \rangle$  be TGTA such that  $\mathcal{L}(\mathcal{T})$  is stuttering insensitive. By combining the first three of the following operations, we can remove all stuttering-transitions that are not self-loop (see Fig. 5). The fourth simplification can be performed along the way.

1. If  $Q \subseteq S$  is a SCC such that any two states  $q, q' \in Q$  can be connected using a sequence of stuttering-transitions  $(q, \emptyset, F_0, q_1)(q_1, \emptyset, F_1, q_2) \cdots (q_n, \emptyset, F_n, q') \in R^*$  with  $F_0 \cup F_1 \cup \cdots \cup F_n = F$ , then we can add an accepting stuttering self-loop  $(q, \emptyset, F, q)$  on each state  $q \in Q$ . I.e., the TGTA  $\mathcal{T}' = \langle S, I, U, R \cup \{(q, \emptyset, F, q) \mid q \in Q\}, F \rangle$  is such that  $\mathcal{L}(\mathcal{T}') = \mathcal{L}(\mathcal{T})$ . Let us call such a component  $Q$  an **accepting Stuttering-SCC**.
2. If there exists an accepting Stuttering-SCC  $Q$  and a sequence of stuttering-transitions  $(s_0, \emptyset, F_1, s_1)(s_1, \emptyset, F_2, s_2) \cdots (s_{n-1}, \emptyset, F_n, s_n) \in R^*$  such that  $s_n \in Q$  and  $s_0, s_1, \dots, s_{n-1} \notin Q$  (Fig. 5a), then:
  - For any non-stuttering transition,  $(s, k, f, s_0) \in R$  going to  $s_0$  and such that  $k \neq \emptyset$ , the TGTA  $\mathcal{T}'' = \langle S, I, U, R \cup \{(s, k, f, s_n)\}, F \rangle$  is such that  $\mathcal{L}(\mathcal{T}'') = \mathcal{L}(\mathcal{T})$ .
  - If  $s_0 \in I$ , the TGTA  $\mathcal{T}'' = \langle S, I \cup \{s_n\}, U'', R, F \rangle$  with  $\forall s \neq s_n, U''(s) = U(s)$  and  $U''(s_n) = U(s_n) \cup U(s_0)$ , is such that  $\mathcal{L}(\mathcal{T}'') = \mathcal{L}(\mathcal{T})$ .
3. Let  $\mathcal{T}^\dagger = \langle S, I^\dagger, U^\dagger, R^\dagger, F \rangle$  be the TGTA obtained after repeating the previous two operations as much as possible (i.e.,  $\mathcal{T}^\dagger$  contains all the transitions and initial states that can be added by the above two operations). Then, we can add non-accepting stuttering self-loops  $(s, \emptyset, \emptyset, s)$  to all states that did not have an accepting stuttering self-loop because  $\mathcal{T}$  describes a stuttering invariant property. Also we can remove all stuttering-transitions that are not self-loops since stuttering can be captured by self-loops after the previous two operations. More formally, the automaton  $\mathcal{T}''' = \langle S, I^\dagger, U^\dagger, R''', F \rangle$  with  $R''' = \{(s, k, f, d) \in R^\dagger \mid k \neq \emptyset \vee (s = d \wedge f = F)\} \cup \{(s, \emptyset, \emptyset, s) \mid (s, \emptyset, F, s) \notin R^\dagger\}$  is such that  $\mathcal{L}(\mathcal{T}''') = \mathcal{L}(\mathcal{T}^\dagger) = \mathcal{L}(\mathcal{T})$ .
4. Any state from which one cannot reach a Büchi-accepting cycle can be removed from the automaton without changing its language. (See appendix B for proofs.)

Here again, an additional optimization is to merge bisimilar states, this can be achieved using the same algorithm used to simplify a TA, taking  $S$  as initial partition and taking into account the acceptance conditions of the outgoing transitions. All these steps are shown on Fig. 4.

We can think of a TGTA as a TGBA whose transitions are labeled by changesets instead of atomic proposition valuations. When checking a TGBA for emptiness, we are looking for an accepting cycle that is reachable from an initial state. When checking a TGTA for emptiness, we are looking exactly for the same thing. The same emptiness check algorithm can be used, because emptiness check algorithms do not look at transition labels.

This is a nice feature of TGTA, not only because it gives us a one-pass emptiness check, but also because it eases the implementation of the TGTA approach in a TGBA-based model checker. We need only to implement the conversion of TGBA to TGTA and the product between a TGTA and a Kripke structure. We discuss our implementation in the next section.

## 4 Experimentation

This section presents our experimentation of the various types of automata within our tool Spot [17]. We first present the Spot architecture and the way the variation on the model checking algorithm was introduced. Then we present our benchmarks (formulae and models) prior to the description of our experiments.

### 4.1 Implementation on Top of Spot

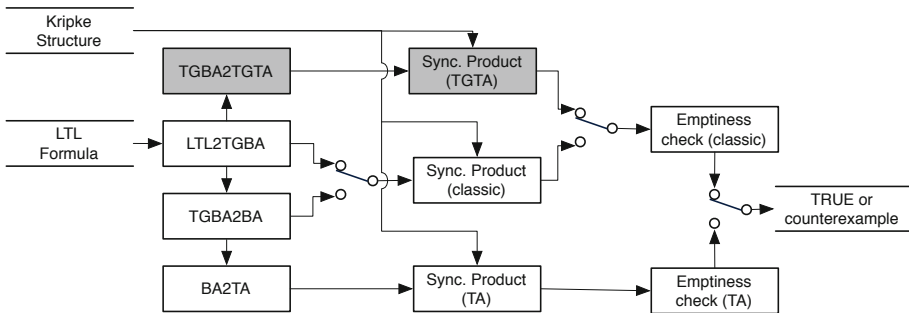
Spot is a model-checking library offering several algorithms that can be combined to build a model checker [7]. Figure 6 shows the building blocks we used to implement the three approaches.

One point that we did not discuss so far is that in the automata-theoretic approach, the automaton used to represent the property to check has to be synchronized with a Kripke structure representing the model. Depending on the kind of automaton (TGBA, BA, TA, TGTA), this *synchronized product* has to be defined differently. Only the TGBA and BA approaches can share the same product definition. The definitions of these different products follow naturally from the definition of the runs on each automata. We refer the reader to Appendix A for a definition of all these products.

The TGBA, BA, and TGTA approaches share the same emptiness check, while a dedicated algorithm is required by the TA approach. In Fig. 6 no direct translation is provided from LTL to TGTA (this is also true for BA and TA). This could be investigated later, the need being, so far, to assess their interest before optimizing the translation process.

In order to evaluate our approach on “realistic” models, we decided to couple the Spot library with the CheckPN tool [7]. CheckPN implements Spot’s Kripke structure interface in order to build the state space of a Petri net on the fly. This Kripke structure is then synchronized with an  $\omega$ -automaton (TGBA, BA, TA or TGTA) on the fly, and fed to the suitable emptiness check algorithm. The latter algorithm drives the on-the-fly construction: only the explored part of the product (and the associated states of the Kripke structure) will be constructed.

Constructing the state space on-the-fly is a double-edged optimization. Firstly, it saves memory, because the state-space is computed as it is explored and thus, does not need to be stored. Secondly, it also saves time when a property is violated because the emptiness check can stop as soon as it has found a counterexample. However, on-the-fly exploration is costlier than browsing an explicit graph: an emptiness check algorithm such as the one for TA that does two traversals of the full state-space in the worst case (e.g. when the property holds) will pay twice the price of that construction.



**Fig. 6.** The experiment’s architecture in SPOT. Three command-line switches control which one of the approaches is used to verify an LTL formula on a Kripke structure. The new components required by the TGTA approach are outlined in Gray.

In the CheckPN implementation of the Kripke structure, the Petri Net marking are compressed to save memory. The marking of a state has to be uncompressed every time we compute its successors, or when we compute the value of the atomic properties on this state. These two operations often occur together, so there is a one-entry cache that prevents the marking from being uncompressed twice in a row.

## 4.2 Benchmark Inputs

We selected some Petri net models and formulæ to compare these approaches.

**Case Studies.** The following two bigger models, were taken from actual cases studies. They come with some *dedicated* properties to check.

**PolyORB** models the core of the  $\mu$ broker component of a middleware [16] in an implementation using a Leader/Followers policy [21]. It is a Symmetric Net and, since CheckPN processes P/T nets only, it was unfolded into a P/T net. The resulting net, for a configuration involving three sources of data, three simultaneous jobs and two threads (one leader, one follower) is composed of 189 places and 461 transitions. Its state space contains 61 662 states<sup>1</sup>. The authors propose to check that once a job is issued from a source, it must be processed by a thread (no starvation). It corresponds to:

$$\Phi_1 = G(MSrc_1 \rightarrow F(DOSrc_1)) \wedge G(MSrc_2 \rightarrow F(DOSrc_2)) \wedge G(MSrc_3 \rightarrow F(DOSrc_3))$$

**MAPK** models a biochemical reaction: Mitogen-activated protein kinase cascade [15]. For a scaling value of 8 (that influences the number of tokens in the initial marking), it contains 22 places and 30 transitions. Its state space contains  $6.11 \times 10^6$  states. The authors propose to check that from the initial state, it is necessary to pass through states *RafP*, *MEKP*, *MEKPP* and *ERKP* in order to reach *ERKPP*. In LTL:

$$\Phi_2 = \neg((\neg RafP)UMEKP) \wedge \neg((\neg MEKP)UMEKPP) \wedge \\ \neg((\neg MEKPP)UERKP) \wedge \neg((\neg ERKP)UERKPP)$$

**Toy Examples.** A first class of four models were selected from the Petri net literature [2, 20]: the flexible manufacturing system (FMS), the Kanban system, the Peterson algorithm, and the slotted-ring system. All these models have a parameter  $n$ . For the Peterson algorithm, and the slotted-ring, the models are composed of  $n$  1-safe subnets. For FMS and Kanban,  $n$  only influences the number of tokens in the initial marking.

We chose values for  $n$  in order to get state space having between  $2 \times 10^5$  to  $3 \times 10^6$  nodes except for Peterson that is  $6.3 \times 10^8$  nodes. The objective is to have non trivial state spaces to be synchronized.

**Types of Formulæ.** As suggested by Geldenhuis and Hansen [10], the type of formula may affect the performances of the various algorithms. In addition to the formulæ  $\Phi_1$  and  $\Phi_2$  above, we consider two classes of formulæ:

<sup>1</sup> This is a rather small value compared to MAPK but, due to the unfolding, each state is a 189-value vector. PolyORB with three sources of data, three simultaneous jobs and three threads would generate 1 137 096 states with 255-value vectors, making the experiment much too slow.



- *RND*: randomly generated LTL formulæ (without X operator). Since random formulæ are very often trivial to verify (the emptiness check needs to explore only a handful of states), for each model we selected only random formulæ that required to explore more than 2000 states with the three approaches.
- *WFair*: properties of the form  $(\bigwedge_{i=1}^n \text{GF } p_i) \rightarrow \varphi$ , where  $\varphi$  is a randomly generated LTL formula. This represents the verification of  $\varphi$  under the weak-fairness hypothesis  $\bigwedge_{i=1}^n \text{GF } p_i$ . The automaton representing such a formula has at least  $n$  acceptance conditions which means that the BA will in the worst case be  $n + 1$  times bigger than the TGBA. For the formulæ we generated for our experiments we have  $n \approx 3.23$  on the average.

All formulæ were translated into automata using Spot, which was shown experimentally to be very good at this job [22, 6]. The time spent doing the conversion from LTL to TGBA and then to TGTA (bisimulation included) is not measured in this benchmark. This translation process is almost instantaneous ( $<0.1$ s), and even if its runtime could be improved (for instance with a direct translation from LTL to TGTA) it is clearly a non significant part of the run time of the different model checking approaches, where all the time is spent performing the emptiness check of the product (built on-the-fly) between the Kripke structure and the property automaton.

### 4.3 Results

Table 1 shows how for TGBA, TA and TGTA approaches deal with toy models and random formulæ. We omit data for BA since they are always outperformed by TGBA. For space reason, we also omit the table showing toy models against weak-fairness formulæ [23], because it shows results similar to those of table 1.

Table 2 shows the results of the two cases studies against random, weak-fairness, and dedicated formulæ issued from the studies.

These tables separate cases where formulæ are verified from cases where they are violated. In the former (left sides of the tables), no counterexample are found and the full state space had to be explored; in the latter (right sides) the on-the-fly exploration of the state space stopped as soon as the existence of a counterexample could be computed.

All values shown in all tables are averaged over 100 different formulas (except for the lines  $\Phi_1$  and  $\Phi_2$  in Table 2 where only one formula is used). For instance we checked Peterson5 against 100 random formulæ that had no counterexample, and against 100 random formulæ that had a counterexample. The average and maximum are computed separately on these two sets of formulæ.

Column-wise, these tables show the average and maximum sizes (states and transitions) of: (1) the automata  $A_{\neg\varphi_i}$  expressing the properties  $\varphi_i$ ; (2) the products  $A_{\neg\varphi_i} \otimes A_M$  of the property with the model; and (3) the subset of this product that was actually explored by the emptiness check. For verified properties, the emptiness check of TGBA and BA always explores the full product so these sizes are equal, while the emptiness check of TA always performs two passes on the full product so it shows double values. On violated properties, the emptiness check aborts as soon as it finds a counterexample, so the explored size is usually significantly smaller than the full product.

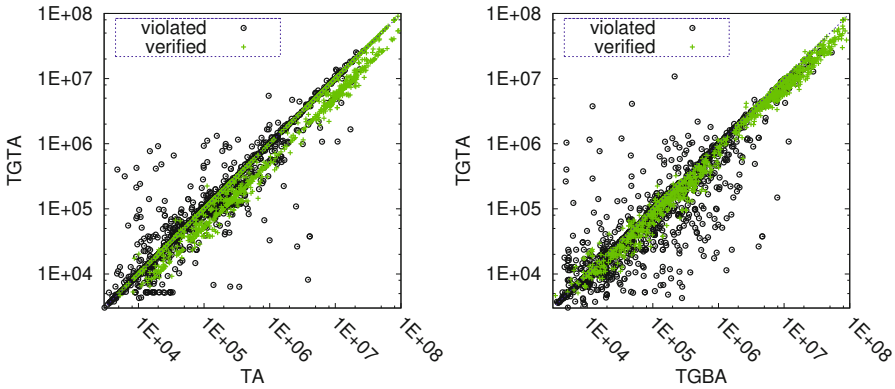
The emptiness check values show a third column labeled “T”: this is the time (in hundredth of seconds, a.k.a. centiseconds) spent doing that emptiness check,

**Table 1.** Comparison of the three approaches on toy examples with random formulae, when counterexamples do not exist (left) or when they do (right)

		Verified properties (no counterexample)						Violated properties (a counterexample exists)								
		Automa.			Emptiness check			Automa.			Full product			Emptiness check		
		st.	tr.	T	st.	tr.	T	st.	tr.	T	st.	tr.	T	st.	tr.	T
Peterson's	TGBA avg	6	74	2489651	10706255	2489651	10706255	18623	6	98	4378750	18950326	447532	1664897	3429	
	TGBA max	15	368	8648954	51231247	8648954	51231247	66305	19	452	15758813	68376074	3464205	16063131	33813	
	TA avg	21	333	2364394	9208893	3820489	14883673	29299	23	400	4530886	17667955	477568	1709098	3626	
Peterson's	TA max	86	2976	6794050	26853438	13588100	53706876	102304	96	2384	15665643	61047330	4356432	16974570	33646	
	TGTA avg	18	384	<b>2327423</b>	<b>9070953</b>	<b>2327423</b>	<b>9070953</b>	<b>17675</b>	20	473	<b>4838504</b>	<b>16958774</b>	<b>421538</b>	<b>1504616</b>	<b>3270</b>	
	TGTA max	63	2541	16802901	26886243	6802901	26886243	50992	89	3931	15665643	61102382	3171366	12373040	27590	
Rings	TGBA avg	6	62	671254	4279403	671254	4279403	986	7	96	1534755	10264597	247215	1257494	349	
	TGBA max	18	360	2788267	27749420	2788267	27749420	5348	25	347	5378284	33010296	1882843	12472859	3613	
	TA avg	18	213	<b>525472</b>	<b>3040856</b>	1020062	5908454	1686	24	366	1410442	8308311	191563	1054280	379	
Rings	TA max	61	1396	2218432	14265568	4436864	28531136	7513	79	2453	4591370	26900864	1583757	9167989	3849	
	TGTA avg	16	266	538121	3168068	<b>538121</b>	<b>3168068</b>	<b>892</b>	21	443	<b>1349758</b>	<b>8084652</b>	<b>176745</b>	<b>979121</b>	<b>323</b>	
	TGTA max	57	1874	2260160	14894816	2260160	14894816	3799	73	2257	3864570	22690780	1380951	8353228	2935	
FMSS	TGBA avg	6	68	551742	3993810	551742	3993810	814	5	69	6883843	62106759	54518	214399	55	
	TGBA max	21	462	7302624	65806572	7302624	65806572	12288	16	280	23726488	244007764	1921058	11918449	2391	
	TA avg	15	186	<b>408588</b>	<b>3161894</b>	482687	3786904	976	17	251	6686607	55303413	31632	165756	52	
FMSS	TA max	51	931	6570291	53742054	9018660	74274828	19268	53	1615	20819154	179323867	1356842	10590558	2814	
	TGTA avg	13	231	407130	3184980	<b>407130</b>	<b>3184980</b>	<b>788</b>	15	305	<b>6397882</b>	<b>53832859</b>	<b>30372</b>	<b>155899</b>	<b>46</b>	
	TGTA max	37	147	6570291	53742054	6570291	53742054	12522	52	1852	19040412	167542101	1356842	10590558	2551	
Kanbans	TGBA avg	5	48	837446	8350628	837446	8350628	1254	6	66	5974582	61030827	45690	165487	31	
	TGBA max	12	238	9500904	135774933	9500904	135774933	18934	20	268	16729695	220725750	915486	4055483	860	
	TA avg	13	133	575668	5450687	575668	5450687	1099	20	225	5659936	54667538	21839	104243	25	
Kanbans	TA max	55	1170	5970944	60234328	5970944	60234328	11567	66	1096	16091600	161286146	439088	3579035	866	
	TGTA avg	11	166	<b>560409</b>	<b>5302142</b>	<b>560409</b>	<b>5302142</b>	<b>1015</b>	17	265	<b>5153380</b>	<b>50559085</b>	<b>19503</b>	<b>97543</b>	<b>22</b>	
	TGTA max	51	1343	5970944	60234328	5970944	60234328	11013	58	1246	13940304	144001788	439056	3578982	819	

**Table 2.** Comparison of the three approaches for the case studies when counterexamples do not exist (left) or when they do (right).

		Automaton		Full product		Emptiness check		Automaton		Full product		Emptiness check	
		st.	tr.	st.	tr.	st.	tr.	st.	tr.	st.	tr.	st.	tr.
RND	TGBA avg	13	438	69141	182821	69141	182821	592	10	438	243147	50886	122212
	TGBA max	55	2852	341835	986034	341835	986034	3115	52	2136	1076730	227290	848061
	TA avg	83	4043	<b>64086</b>	<b>138821</b>	104415	226731	895	53	3474	103035	52455	117020
	TA max	411	46769	295389	653208	590778	1306416	5350	469	62602	336393	221833	478119
	TGTA avg	67	4121	65108	141407	<b>65108</b>	<b>141407</b>	<b>558</b>	43	3910	102151	<b>46907</b>	<b>104919</b>
	TGTA max	334	47567	317676	703265	317676	703265	2731	393	66517	363075	193890	503543
PolyORB 3/32	TGBA avg	4	38	65276	153015	65276	153015	509	4	36	73483	57295	130841
	TGBA max	12	128	184986	791568	184986	791568	1713	12	130	193995	135785	401864
	TA avg	55	641	79602	170480	83446	178828	630	51	593	155237	91714	201664
	TA max	164	2102	184986	397458	269934	603126	2384	195	3162	379629	197846	433800
	TGTA avg	21	264	<b>62469</b>	<b>133825</b>	<b>62469</b>	<b>133825</b>	<b>480</b>	20	243	<b>71784</b>	<b>55984</b>	<b>122560</b>
	TGTA max	52	681	184986	397458	184986	397458	1618	52	906	184986	116737	253999
$\Phi_1$	TGBA -	7	576	345241	760491	<b>345241</b>	<b>760491</b>	<b>1921</b>					
	TA -	80	14590	<b>342613</b>	<b>742815</b>	685226	1485630	3858					
	TGTA -	79	17153	345277	753798	345277	<b>753798</b>	1925					
RND	TGBA avg	5	60	1095515	11940964	1095515	11940964	2211	5	62	15364530	44621	147170
	TGBA max	14	256	11408161	147987445	11408161	147987445	26963	14	278	57275770	1657316	7858321
	TA avg	14	173	812843	10119485	812843	10119485	2228	15	192	15531840	22071	53569
	TA max	48	969	9793932	125681958	9793932	125681958	27197	55	916	44955300	318571	832689
	TGTA avg	12	226	<b>812456</b>	<b>10116066</b>	<b>812456</b>	<b>10116066</b>	<b>2173</b>	13	236	<b>14360794</b>	<b>21080</b>	<b>51231</b>
	TGTA max	47	1304	9793932	125681958	9793932	125681958	26208	44	1141	43481872	318570	832687
MAPK 8	TGBA avg	3	20	2649730	30385763	2649730	30385763	5602	4	33	9711621	43651	<b>195459</b>
	TGBA max	8	84	16962176	268267694	16962176	268267694	45042	12	136	33023267	2248882	13302369
	TA avg	31	296	1779991	21870089	1779991	21870089	5033	44	471	20285910	33160	201268
	TA max	103	1270	13254682	172100144	13254682	172100144	43217	131	1786	44841684	1130354	12156893
	TGTA avg	13	131	<b>1684025</b>	<b>20831377</b>	<b>1684025</b>	<b>20831377</b>	<b>4660</b>	18	208	<b>8596802</b>	<b>110988663</b>	<b>31661</b>
	TGTA max	43	591	13254682	172100144	13254682	172100144	37989	60	871	21684220	1129783	12155921
$\Phi_2$	TGBA -	6	165	46494	302350	46494	302350	53					
	TA -	9	293	33376	289235	33376	289235	55					
	TGTA -	8	452	<b>33376</b>	<b>289235</b>	<b>33376</b>	<b>289235</b>	<b>53</b>					



**Fig. 7.** Performance (transitions explored by the emptiness check) of TGTA against TA and TGBA

including the on-the-fly computation of the subset of the product that is explored. The time spent constructing the property automata from the formulæ is not shown (it is negligible compared to that of the emptiness check).

Figure 7 compares the number of visited transitions when running the emptiness check; plotting TGTA against TA and TGBA. This gives an idea of their relative performance. Each point corresponds to one of the 4100 evaluated formulas (2050 violated with counterexample as black circles, and 2050 verified having no counterexample as green crosses). Each point below the diagonal is in favor of TGTA while others are in favor of the other approach. Axes are displayed using a logarithmic scale. No comparison is presented with BA since they are less efficient than TGBA [23].

All these tests were run on a 64bit Linux system running on an Intel Core 2 Quad Processor Q9400 at 2.66GHz, with 4GB of RAM.

## 5 Discussion

Although the state space of cases studies can be very different from random state spaces [18], a first look at our results confirms two facts already observed in previous studies [10]: (1) although the TA constructed from properties are usually a lot larger than TGBA (and even larger than BA [23]), the average size of the full product is smaller thanks to the more deterministic nature of the TA. (2) For violated properties, the TA approach explores less states and transitions on the average than TGBA or BA.

We complete this picture by showing run times, by separating verified properties from violated properties, and by also evaluating the TGBA approach.

It should be noted that our implementation has been improved since our previous experiments [23] where the cost of computing labels in the Kripke structure was higher than it is now (we use a cache). This change mainly benefit to testing automata, because they query two labels by transition of the Kripke structure (to compute an xor between source label and destination label) while other approaches query only one label.

For weak-fairness formulæ, we show only the results for cases studies because for toy examples we obtain similar results as random formulæ.

**On verified properties** the results are very straightforward to interpret when looking at the number of transitions explored by the emptiness check. TA outperform TGBA except for both Random and weak-fairness properties against Peterson, Ring and PolyORB. These are typical cases where the TA emptiness check has to perform two passes: this can be observed in the tables [1](#) and [2](#) when the number of transitions visited by the emptiness check is on the average twice the number of transitions of the product.

In these three cases, the TGTA approach, with its single-pass emptiness check, is a clear improvement over TA. On the left scatter plots of Fig. [7](#), these cases where the TGTA approach is twice faster than TA's, appear as a linear cloud of green crosses below the diagonal (because the axes are displayed using a logarithmic scale).

In the other where TA need only one pass on the average (e.g. Kanban, MAPK), TGTA and TA have similar performance, with a slight advantage for TGTA because the products are smaller.

As a consequence the TGTA approach outperforms TGBA and TA in all cases on verified properties.

**On violated properties**, it is harder to interpret these tables because the emptiness check will return as soon as it finds a counterexample. Changing the order in which non-deterministic transitions of the property automaton are iterated is enough to change the number of states and transitions to be explored before a counterexample is found: in the best case the transition order will lead the emptiness check straight to an accepting cycle; in the worst case, the algorithm will explore the whole product until it finally finds an accepting cycle. Although the emptiness check algorithms for the three approaches share the same routines to explore the automaton, they are all applied to different kinds of property automata, and thus provide different transition orders.

We believe that the TA and TGTA, since they are more deterministic [\[10\]](#), are less sensitive to this ordering. Also, in all of our experiments the TA approach has always found the counterexample in the first pass of the emptiness check algorithm. This supports Geldenhuys and Hansen's claim that the second pass was seldom needed for violated properties (less than 0.005% of the cases in their experiments [\[10\]](#)). Finally, in the tables [1](#) and [2](#) we observe that the TGTA approach explores the smallest products on the average.

## 6 Conclusion

This paper is the sequel of a preliminary work [\[23\]](#) experimenting LTL model checking of stuttering-insensitive properties with various techniques: Büchi automata (BA), Transition-based Generalized Büchi Automata and Testing Automata [\[10\]](#). At this time, conclusions were that TA outperformed BA and sometimes TGBA for unverified properties (i.e., when a counterexample was found). However, this was not the case when no counterexample was computed since the entire state space may had to be visited twice to check for each acceptance mode of a TA (Büchi acceptance or livelock-acceptance).

This paper extends the above work by proposing a new type of  $\omega$ -automaton: Transition-based Generalized Testing Automata (TGTA). It inherits from TA the labeling of transitions by changesets and, from TGBA, the use of transition-based acceptance conditions. The idea is to combine advantages observed on both TA and TGBA.

TGTA have been implemented in Spot easily, because only two new algorithms are required: the conversion of a TGBA into a TGTA, and a new definition of a product between a TGTA and a Kripke structure.

We have run benchmarks to assess their interest. Experiments reported that, in most cases, TGTA outperform TA and TGBA when no counterexample is found in the system and are comparable when the property is violated.

We conclude that there is nothing to lose by using TGTA to verify stuttering-insensitive properties, since they are always at least as good as TA and TGBA.

**Future Work.** We plan additional work to enable symbolic model checking with TGTA, thus allowing us to tackle much larger state spaces than in explicit model checking. Another idea would be to provide a direct conversion of LTL to TGTA, without the intermediate TGBA step. We believe a tableau construction such as the one of Couvreur [4] could be easily adapted to produce TGTA.

## References

- [1] Babiak, T., Křetínský, M., Řehák, V., Strejček, J.: LTL to Büchi Automata Translation: Fast and More Deterministic. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 95–109. Springer, Heidelberg (2012)
- [2] Ciardo, G., Lüttgen, G., Siminiceanu, R.: Efficient Symbolic State-Space Construction for Asynchronous Systems. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 103–122. Springer, Heidelberg (2000)
- [3] Cichoń, J., Czubak, A., Jasiński, A.: Minimal Büchi automata for certain classes of LTL formulas. In: Proceedings of the Fourth International Conference on Dependability of Computer Systems, DEPCOS 2009, pp. 17–24. IEEE Computer Society (2009)
- [4] Couvreur, J.-M.: On-the-Fly Verification of Linear Temporal Logic. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 253–271. Springer, Heidelberg (1999)
- [5] Couvreur, J.-M., Duret-Lutz, A., Poitrenaud, D.: On-the-Fly Emptiness Checks for Generalized Büchi Automata. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 169–184. Springer, Heidelberg (2005)
- [6] Duret-Lutz, A.: LTL translation improvements in Spot. In: Proceedings of the 5th International Workshop on Verification and Evaluation of Computer and Communication Systems, VECoS 2011. Electronic Workshops in Computing. British Computer Society, Tunis (2011), <http://ewic.bcs.org/category/15853>
- [7] Duret-Lutz, A., Poitrenaud, D.: SPOT: an extensible model checking library using transition-based generalized Büchi automata. In: Proceedings of the 12th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2004, pp. 76–83. IEEE Computer Society Press, Volendam (2004)
- [8] Farwer, B.:  $\omega$ -Automata. In: Grädel, E., Thomas, W., Wilke, T. (eds.) Automata, Logics, and Infinite Games. LNCS, vol. 2500, pp. 3–21. Springer, Heidelberg (2002)
- [9] Gastin, P., Oddoux, D.: Fast LTL to Büchi Automata Translation. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 53–65. Springer, Heidelberg (2001)
- [10] Geldenhuys, J., Hansen, H.: Larger Automata and Less Work for LTL Model Checking. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 53–70. Springer, Heidelberg (2006)
- [11] Geldenhuys, J., Valmari, A.: Tarjan’s Algorithm Makes On-the-Fly LTL Verification More Efficient. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 205–219. Springer, Heidelberg (2004)

- [12] Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Proceedings of the 15th Workshop on Protocol Specification Testing and Verification, PSTV 1995, pp. 3–18. Chapman & Hall, Warsaw (1995), <http://citeseer.nj.nec.com/gerth95simple.html>
- [13] Giannakopoulou, D., Lerda, F.: From States to Transitions: Improving Translation of LTL Formulae to Büchi Automata. In: Peled, D.A., Vardi, M.Y. (eds.) FORTE 2002. LNCS, vol. 2529, pp. 308–326. Springer, Heidelberg (2002)
- [14] Hansen, H., Penczek, W., Valmari, A.: Stuttering-insensitive automata for on-the-fly detection of livelock properties. In: Cleaveland, R., Garavel, H. (eds.) Proceedings of the 7th International ERCIM Workshop in Formal Methods for Industrial Critical Systems, FMICS 2002. Electronic Notes in Theoretical Computer Science, vol. 66(2). Elsevier, Málaga (2002)
- [15] Heiner, M., Gilbert, D., Donaldson, R.: Petri Nets for Systems and Synthetic Biology. In: Bernardo, M., Degano, P., Zavattaro, G. (eds.) SFM 2008. LNCS, vol. 5016, pp. 215–264. Springer, Heidelberg (2008)
- [16] Hugues, J., Thierry-Mieg, Y., Kordon, F., Pautet, L., Barrir, S., Vergnaud, T.: On the formal verification of middleware behavioral properties. In: Proceedings of the 9th International Workshop on Formal Methods for Industrial Critical Systems, FMICS 2004. Electronic Notes in Theoretical Computer Science, vol. 133, pp. 139–157. Elsevier Science Publishers (September 2004)
- [17] MoVe/LRDE: The Spot home page (2012), <http://spot.lip6.fr>
- [18] Pelánek, R.: Properties of state spaces and their applications. International Journal on Software Tools for Technology Transfer (STTT) 10(5), 443–454 (2008)
- [19] Peled, D., Wilke, T.: Stutter-invariant temporal properties are expressible without the next-time operator. Information Processing Letters 63(5), 243–246 (1995)
- [20] Peterson, G.L.: Myths about the mutual exclusion problem. Inf. Process. Lett. 12(3), 115–116 (1981)
- [21] Pyarali, I., Spivak, M., Cytron, R., Schmidt, D.C.: Evaluating and optimizing thread pool strategies for RT-CORBA. In: Proceeding of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems, LCTES 2000, pp. 214–222. ACM (2000)
- [22] Rozier, K.Y., Vardi, M.Y.: LTL Satisfiability Checking. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 149–167. Springer, Heidelberg (2007)
- [23] Salem, A.E.B., Duret-Lutz, A., Kordon, F.: Generalized Büchi automata versus testing automata for model checking. In: Proceedings of the 2nd Workshop on Scalable and Usable Model Checking for Petri Nets and Other Models of Concurrency, SUMo 2011, vol. 726, pp. 65–79. CEUR, Newcastle (2011)
- [24] Schwoon, S., Esparza, J.: A Note on On-the-Fly Verification Algorithms. In: Halbwachs, N., Zuck, L. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 174–190. Springer, Heidelberg (2005)
- [25] Sebastiani, R., Tonetta, S.: “More Deterministic” vs. “Smaller” Büchi Automata for Efficient LTL Model Checking. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 126–140. Springer, Heidelberg (2003)
- [26] Tauriainen, H.: Automata and Linear Temporal Logic: Translation with Transition-based Acceptance. Ph.D. thesis, Helsinki University of Technology, Espoo, Finland (September 2006)
- [27] Valmari, A.: Bisimilarity Minimization in  $O(m \log n)$  Time. In: Franceschinis, G., Wolf, K. (eds.) PETRI NETS 2009. LNCS, vol. 5606, pp. 123–142. Springer, Heidelberg (2009), [http://dx.doi.org/10.1007/978-3-642-02424-5\\_9](http://dx.doi.org/10.1007/978-3-642-02424-5_9)
- [28] Vardi, M.Y.: An Automata-Theoretic Approach to Linear Temporal Logic. In: Moller, F., Birtwistle, G. (eds.) Logics for Concurrency. LNCS, vol. 1043, pp. 238–266. Springer, Heidelberg (1996)

## A Product Definitions

### A.1 Product of TGBA (or BA) with a Kripke Structure

The product of a TGBA with a Kripke structure is a TGBA whose language is the intersection of both languages.

**Definition 7.** For a Kripke structure  $\mathcal{K} = \langle S_{\mathcal{K}}, I_{\mathcal{K}}, R_{\mathcal{K}}, L_{\mathcal{K}} \rangle$  and a TGBA  $\mathcal{G} = \langle S_{\mathcal{G}}, I_{\mathcal{G}}, R_{\mathcal{G}}, F_{\mathcal{G}} \rangle$  the product  $\mathcal{K} \otimes \mathcal{G}$  is the TGBA  $\langle S, I, R, F \rangle$  where

- $S = S_{\mathcal{K}} \times S_{\mathcal{G}}$ ,
- $I = I_{\mathcal{K}} \times I_{\mathcal{G}}$ ,
- $R = \{((s, s'), L_{\mathcal{K}}(s), f, (d, d')) \mid (s, d) \in R_{\mathcal{K}}, (s', k, f, d') \in R_{\mathcal{G}}, L_{\mathcal{K}}(s) \in k\}$
- $F = F_{\mathcal{G}}$ .

**Property 6.** We have  $\mathcal{L}(\mathcal{K} \otimes \mathcal{G}) = \mathcal{L}(\mathcal{K}) \cap \mathcal{L}(\mathcal{G})$  by construction.

Since a BA can be seen as a TGBA with a unique acceptance set, and all state-based acceptance conditions pushed to the outgoing transitions, the same construction can be used to make a product between a Kripke structure and a BA.

### A.2 Product of a TA with a Kripke Structure

For TGBA (or BA) the synchronized product with a Kripke structure can be defined as another TGBA (or BA). In the case of testing automata, the product of a Kripke and a TA is not a TA: while an execution in a TA is allowed to stutter on any state, the execution in a product must always progress.

**Definition 8.** For a Kripke structure  $\mathcal{K} = \langle S_{\mathcal{K}}, I_{\mathcal{K}}, R_{\mathcal{K}}, L_{\mathcal{K}} \rangle$  and a TA  $\mathcal{T} = \langle S_{\mathcal{T}}, I_{\mathcal{T}}, U_{\mathcal{T}}, R_{\mathcal{T}}, F_{\mathcal{T}}, G_{\mathcal{T}} \rangle$ , the product  $\mathcal{K} \otimes \mathcal{T}$  is an automaton  $\langle S, I, U, R, F, G \rangle$  where

- $S = S_{\mathcal{K}} \times S_{\mathcal{T}}$ ,
- $I = \{(s, s') \in I_{\mathcal{K}} \times I_{\mathcal{T}} \mid L_{\mathcal{K}}(s) \in U_{\mathcal{T}}(s')\}$ ,
- $\forall (s, s') \in I, U((s, s')) = \{L_{\mathcal{K}}(s)\}$ ,
- $R = \{((s, s'), k, (d, d')) \mid (s, d) \in R_{\mathcal{K}}, (s', k, d') \in R_{\mathcal{T}}, k = L_{\mathcal{K}}(s) \oplus L_{\mathcal{K}}(d)\}$   
 $\cup \{((s, s'), \emptyset, (d, d')) \mid (s, d) \in R_{\mathcal{K}}, s' = d', L_{\mathcal{K}}(s) = L_{\mathcal{K}}(d)\}$
- $F = S_{\mathcal{K}} \times F_{\mathcal{T}}$ , and  $G = S_{\mathcal{K}} \times G_{\mathcal{T}}$ .

An execution  $w = k_0 k_1 k_2 \dots \in K^{\omega}$  is accepted by  $\mathcal{K} \otimes \mathcal{T}$  if there exists an infinite sequence  $(s_0, k_0 \oplus k_1, s_1)(s_1, k_1 \oplus k_2, s_2) \dots (s_i, k_i \oplus k_{i+1}, s_{i+1}) \dots \in (S \times K \times S)^{\omega}$  such that:

- $s_0 \in I$  with  $k_0 \in U(s_0)$ ,
- $\forall i \in \mathbb{N}, (s_i, k_i \oplus k_{i+1}, s_{i+1}) \in R$  (we are always progressing in the product)
- Either,  $\forall i \in \mathbb{N}, (\exists j \geq i, k_j \neq k_{j+1}) \wedge (\exists l \geq i, s_l \in F)$  (the automaton is progressing in a Büchi-accepting way), or,  $\exists n \in \mathbb{N}, \forall i \geq n, (k_i = k_n) \wedge (s_i \in G)$  (a suffix of the execution stutters in  $G$ ).

**Property 7.** We have  $\mathcal{L}(\mathcal{K} \otimes \mathcal{T}) = \mathcal{L}(\mathcal{K}) \cap \mathcal{L}(\mathcal{T})$  by construction.



### A.3 Product of a TGTA with a Kripke Structure

The product of a TGTA with a Kripke structure is a TGTA.

Comparing this definition with the previous two products shows the double inheritance of TGTA. This product is similar to the product between a TA and a Kripke structure, except it does not deal with livelock acceptance states and implicit stuttering. It is also similar to the product of a TGBA with a Kripke structure, except for the use of changesets on transitions, and the initial labels ( $U$ ).

**Definition 9.** For a Kripke structure  $\mathcal{K} = \langle S_{\mathcal{K}}, I_{\mathcal{K}}, R_{\mathcal{K}}, L_{\mathcal{K}} \rangle$  and a TGTA  $\mathcal{T} = \langle S_{\mathcal{T}}, I_{\mathcal{T}}, U_{\mathcal{T}}, R_{\mathcal{T}}, F_{\mathcal{T}} \rangle$ , the product  $\mathcal{K} \otimes \mathcal{T}$  is a TGTA  $\langle S, I, U, R, F \rangle$  where

- $S = S_{\mathcal{K}} \times S_{\mathcal{T}}$ ,
- $I = \{(s, s') \in I_{\mathcal{K}} \times I_{\mathcal{T}} \mid L_{\mathcal{K}}(s) \in U_{\mathcal{T}}(s')\}$ ,
- $\forall (s, s') \in I, U((s, s')) = \{L_{\mathcal{K}}(s)\}$ ,
- $R = \{(s, s'), k, f, (d, d') \mid (s, d) \in R_{\mathcal{K}}, (s', k, f, d') \in R_{\mathcal{T}}, k = L_{\mathcal{K}}(s) \oplus L_{\mathcal{K}}(d)\}$
- $F = F_{\mathcal{T}}$ .

**Property 8.** We have  $\mathcal{L}(\mathcal{K} \otimes \mathcal{T}) = \mathcal{L}(\mathcal{K}) \cap \mathcal{L}(\mathcal{T})$  by construction.

## B Model Checking Using TGBA

When doing model checking with TGBA the two important operations are the translation of the linear-time property  $\phi$  into a TGBA  $A_{-\phi}$  and the emptiness check of the product of the Kripke structure  $\mathcal{K}$  with  $A_{-\phi}$ : this product  $\mathcal{K} \otimes A_{-\phi}$  is a TGBA. Numerous algorithms translate LTL formulæ into TGBA [13, 4, 1, 26]. We use Couvreur's one [4] with some optimizations [6].

Testing a TGBA for emptiness amounts to the search of a strongly connected component that contains at least one occurrence of each acceptance condition. It can be done in two different ways: either with a variation of Tarjan or Dijkstra algorithm [4] or using several nested depth-first searches to save memory [26]. The latter proved to be slower [5], so we are using Couvreur's SCC-based emptiness check algorithm [4]. Another advantage of the SCC-based algorithm is that their complexity does not depend on the number of acceptance conditions.

Algorithm. 1 presents an iterative version of Couvreur's algorithm [4]. This algorithm computes on the fly the maximal Strongly Connected Components: it performs a Depth-First Search (DFS) for SCC detection and then merges the SCCs belonging to the same maximal SCC into a single SCC. After each merge, if the union of all acceptance conditions occurring in the merged SCC is equal to  $F$ , then an accepting run is found. *todo* is the DFS stack. It is used by the procedure `DFSpush` to push the states of the current DFS path and the set of their successors that have not yet been visited.  $H$  maps each visited state to its rank in the DFS order, and  $H[s] = 0$  indicates that  $s$  is a dead state (i.e.,  $s$  belongs to a maximal SCC that has been fully explored). Figure 9 illustrates a run of this algorithm on a small example.

The SCC stack stores a chain of partial SCCs found during the DFS. For each SCC the attribute *root* is the DFS rank ( $H$ ) of the first state of the SCC, *acc* is the set of all

```

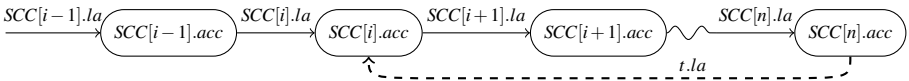
1 Input: A product TGBA  $G = \langle S, I, R, F \rangle$ 
2 Result: true if and only if  $\mathcal{L}(G) = \emptyset$ 
3 Data: todo: stack of  $\langle \text{state} \in S, \text{succ} \subseteq R \rangle$ 
   SCC: stack of
    $\langle \text{root} \in \mathbb{N}, \text{la} \subseteq F, \text{acc} \subseteq F, \text{rem} \subseteq S \rangle$ 
   H: map of  $S \mapsto \mathbb{N}$ 
   max  $\leftarrow 0$ 
4 begin
5   foreach  $s^0 \in I$  do
6     DFSpush( $\emptyset, s^0$ )
7     while  $\neg \text{todo.empty}()$  do
8       if  $\text{todo.top().succ} = \emptyset$  then
9         DFSpop()
10      else
11        pick one  $\langle s, \_, a, d \rangle$  off  $\text{todo.top().succ}$ 
12        if  $d \notin H$  then
13          DFSpush( $a, d$ )
14        else if  $H[d] > 0$  then
15          merge( $a, H[d]$ )
16        if  $\text{SCC.top().acc} = F$  then
17          return false
18      return true
19 DFSpush( $\text{la} \subseteq F, s \in S$ )
20   max  $\leftarrow \text{max} + 1$ 
21   H[ $s$ ]  $\leftarrow \text{max}$ 
22   SCC.push( $\langle \text{max}, \text{la}, \emptyset, \emptyset \rangle$ )
23   todo.push( $\langle s, \{ \langle q, l, a, d \rangle \in R \mid q = s \} \rangle$ )
24 DFSpop()
25    $\langle s, \_ \rangle \leftarrow \text{todo.pop}()$ 
26   SCC.top().rem.insert( $s$ )
27   if  $H[s] = \text{SCC.top().root}$  then
28     foreach  $s \in \text{SCC.top().rem}$  do
29       H[ $s$ ]  $\leftarrow 0$ 
30     SCC.pop()
31 merge( $\text{la} \subseteq F, t \in \mathbb{N}$ )
32   r  $\leftarrow \emptyset$ 
33   acc  $\leftarrow \text{la}$ 
34   while  $t < \text{SCC.top().root}$  do
35     acc  $\leftarrow \text{acc} \cup \text{SCC.top().acc}$ 
36            $\cup \text{SCC.top().la}$ 
37     r  $\leftarrow r \cup \text{SCC.top().rem}$ 
38     SCC.pop()
39   SCC.top().acc  $\leftarrow \text{SCC.top().acc} \cup \text{acc}$ 
   SCC.top().rem  $\leftarrow \text{SCC.top().rem} \cup r$ 

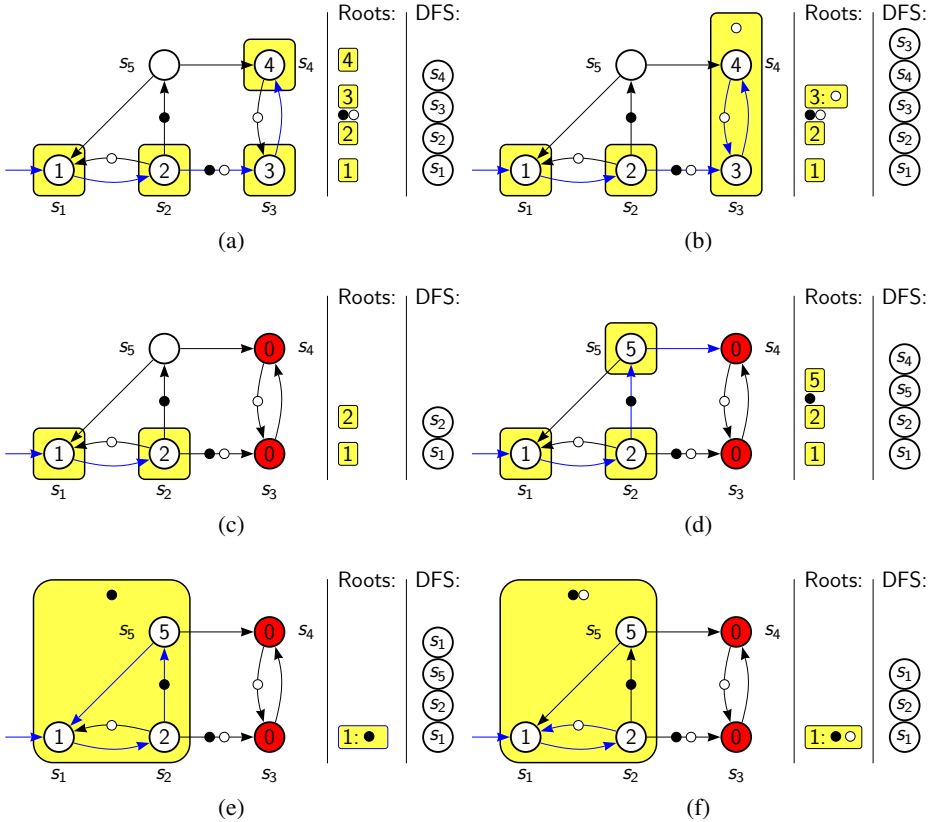
```

Algorithm 1. Emptiness check algorithm for TGBA

acceptance conditions belonging to the SCC, *la* is the acceptance conditions of the transition between the previous and the current SCC, and *rem* contains the fully explored states of the SCC. Figure 8 shows how *acc* and *la* are used in the SCC search stack.

The algorithm begins by pushing in *SCC* each state visited for the first time (line 12), as a trivial SCC with an empty *acc* set (line 22). Then, when the DFS explores a transition  $t$  between two states  $s$  and  $d$ , if  $d$  is in the SCC stack (line 14), therefore  $t$  closes a cycle passing through  $s$  and  $d$  in the product automaton. This cycle “strongly connects” all SCCs pushed in the *SCC* stack between  $\text{SCC}[i]$  and  $\text{SCC}[n]$ : the two SCCs that respectively contains the states  $d$  and  $s$  ( $\text{SCC}[n]$  is the top of the *SCC* stack). All the SCCs between  $\text{SCC}[i]$  and  $\text{SCC}[n]$  are merged (line 15) into  $\text{SCC}[i]$ . The merge of acceptance conditions is illustrated by Fig. 8: a “back” transition  $t$  is found between  $\text{SCC}[n]$  and  $\text{SCC}[i]$ , therefore the latest SCCs (from  $i$  to  $n$ ) are merged. The acceptance conditions of the merged SCC is equal to the union of  $\text{SCC}[i].\text{acc} \cup \text{SCC}[i+1].\text{la} \cup \text{SCC}[i+1].\text{acc} \cup \dots \cup \text{SCC}[n].\text{la} \cup \text{SCC}[n].\text{acc} \cup t.\text{la}$ . If this union is equal to  $F$ , then the merged SCC is accepting and the algorithm return *false* (line 17): the product is not empty.

Fig. 8. SCC stack: the use of the SCCs fields *la* and *acc*



**Fig. 9.** Six intermediate steps in a run of algorithm [11](#). The states  $s_1, \dots, s_5$  are labeled by their value in  $H$ . The stack of roots of SCCs (the *root* stack in the algorithm) and the DFS search stack (induced by the recursive calls to `DFSpush()`) are displayed on the side. An interpretation of the SCC stack in term of SCCs is given as yellow blobs on the automaton.

(a) Initially the algorithm performs a DFS search by declaring each newly encountered state as a trivial SCC. (b) When the transition from  $s_4$  to  $s_3$  is processed, the algorithm detects that  $H[s_3] \neq 0$  which means the transition creates a cycle and all SCCs between  $s_4$  and  $s_3$  are merged. (c) When the DFS exits the non-accepting  $\{s_3, s_4\}$  SCC, it marks all its states as dead ( $H[s] = 0$ ). (d) When the DFS tries attempt to visit a dead state, it ignores it. (e) Visiting the transition from  $s_5$  to  $s_1$  will merge three SCCs into one, but it does not yet appear to be accepting because the white acceptance has not been seen. (f) Finally visiting the transition from  $s_2$  back to  $s_1$  will contribute white acceptance condition to the current SCC, and the algorithm will stop immediately because it has found an SCC labeled by all acceptance conditions.

## C Model Checking Using BA

Since a BA can be seen as a TGBA by pushing acceptance conditions from states to outgoing transitions, the emptiness check from Algorithm. 1 also works. Other algorithms, specific to BA, are based on two nested depth-first searches. The comparison of these different emptiness checks raised many studies [11, 24, 5], and for this work we only consider the SCC-based algorithm presented here.

## D Emptiness Check Using TA

Testing Automata require a dedicated algorithm because there are two ways to detect an accepting cycle in the product:

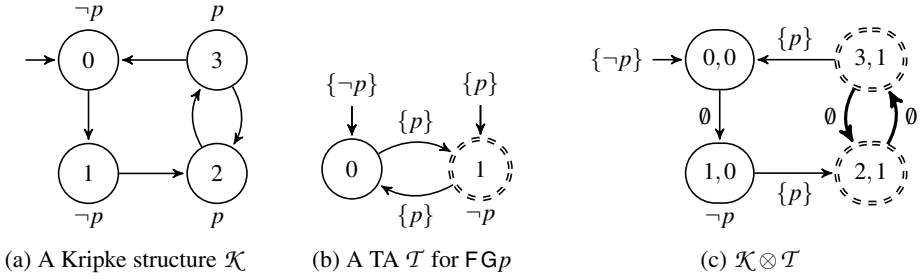
- Büchi acceptance: a cycle containing at least one Büchi-acceptance state (F) and at least one non-stuttering transition (i.e., a transition  $(s, k, s')$  with  $k \neq \emptyset$ ),
- livelock acceptance: a cycle composed only of stuttering transitions and livelock acceptance states (G).

```

1 Input: A product  $\mathcal{K} \otimes \mathcal{T} = \langle S, I, U, R, F, G \rangle$ 
2 Result: true if and only if  $\mathcal{L}(T) = \emptyset$ 
3 Data: todo: stack of  $\langle \text{state} \in S, \text{succ} \subseteq R \rangle$ 
      SCC: stack of  $\langle \text{root} \in \mathbb{N}, lk \in 2^{AP}, k \in 2^{AP}, \text{acc} \subseteq F, \text{rem} \subseteq S \rangle$ 
      H: map of  $S \mapsto \mathbb{N}$ 
      max  $\leftarrow 0$ , Gseen  $\leftarrow \text{false}$ 
4 begin
5   if  $\neg \text{first-pass}()$  then return false
6   if Gseen then return second-pass()
7   return true
8 first-pass()
9   foreach  $s^0 \in I$  do
10    DFSpush1( $\emptyset, s^0$ )
11    while  $\neg \text{todo.empty}()$  do
12      if todo.top().succ =  $\emptyset$  then
13        DFSpop()
14      else
15        pick one  $\langle s, k, d \rangle$  off todo.top().succ
16        if  $d \notin H$  then
17          DFSpush1( $k, d$ )
18        else if  $H[d] > 0$  then
19          merge1( $k, H[d]$ )
20          if  $(\text{SCC.top().acc} \neq \emptyset) \wedge$ 
21             $(\text{SCC.top().k} \neq \emptyset)$  then return
22            false
21          if  $(d \in G) \wedge (\text{SCC.top().k} = \emptyset)$  then
22            return false
22 return true
23 DFSpush1( $lk \in 2^{AP}, s \in S$ )
24   max  $\leftarrow \text{max} + 1$ 
25   H[s]  $\leftarrow \text{max}$ 
26   if  $s \in F$  then
27     SCC.push( $\langle \text{max}, lk, \emptyset, \{s\}, \emptyset \rangle$ )
28   else
29     SCC.push( $\langle \text{max}, lk, \emptyset, \emptyset, \emptyset \rangle$ )
30   todo.push( $\langle s, \{ \langle q, k, d \rangle \in R \mid q = s \} \rangle$ )
31   if  $s \in G$  then
32     Gseen  $\leftarrow \text{true}$ 
33 merge1( $lk \in 2^{AP}, t \in \mathbb{N}$ )
34   acc  $\leftarrow \emptyset$ 
35   r  $\leftarrow \emptyset$ 
36   k  $\leftarrow lk$ 
37   while  $t < \text{SCC.top().root}$  do
38     acc  $\leftarrow \text{acc} \cup \text{SCC.top().acc}$ 
39     k  $\leftarrow k \cup \text{SCC.top().k} \cup \text{SCC.top().lk}$ 
40     r  $\leftarrow r \cup \text{SCC.top().rem}$ 
41     SCC.pop()
42   SCC.top().acc  $\leftarrow \text{SCC.top().acc} \cup \text{acc}$ 
43   SCC.top().k  $\leftarrow \text{SCC.top().k} \cup k$ 
44   SCC.top().rem  $\leftarrow \text{SCC.top().rem} \cup r$ 

```

**Algorithm 2.** The first-pass of the Emptiness check algorithm for TA products



**Fig. 10.** Example product between a Kripke structure and a TA. The bold cycle is livelock-accepting.

A straightforward emptiness check would have two passes: a first pass to detect Büchi acceptance cycles, it corresponds to Algorithm. 2 without the test at line 21 and a second pass presented in Algorithm. 3 to detect livelock acceptance cycles. It is not possible to merge these two passes into a single DFS: the first DFS requires the full product exploration while the second one must consider stuttering transitions only. These two passes are an inconvenience when the property is satisfied (no counterexample) since the entire state-space has to be explored twice.

With line 21 included in Algorithm. 2 the first-pass detects both Büchi and some livelock-acceptance cycles. Since in certain cases it may fail to report some livelock-acceptance cycles, a second pass is required to look for possible livelock-acceptance cycles.

This first-pass is based on the TGBA emptiness check algorithm presented in Algorithm. 1 with the following changes:

- In each item  $scc$  of the  $SCC$  stack: the field  $scc.acc$  contains the Büchi-accepting states detected in  $scc$ ,  $scc.lk$  is analogous to  $la$  in Fig. 8 but it stores the *change-set* labeling the transition coming from the previous SCC, and  $scc.k$  contains the union of all *change-sets* in  $scc$  (lines 39 and 43).
- After each merge,  $SCC.top()$  is checked for Büchi-acceptance (line 20) or livelock-acceptance (line 21) depending on the emptiness of  $SCC.top().k$ .

Figure 10 illustrates how the first-pass of Algorithm. 2 can fail to detect the livelock accepting cycle in a product  $\mathcal{K} \otimes \mathcal{T}$  as defined in def. 8. In this example,  $G_{\mathcal{T}} = \{1\}$  therefore  $(3, 1)$  and  $(2, 1)$  are livelock-accepting states, and  $C_2 = [(3, 1) \rightarrow (2, 1) \rightarrow (3, 1)]$  is a livelock-accepting cycle.

However, the first-pass may miss this livelock-accepting cycle depending on the order in which it processes the outgoing transitions of  $(3, 1)$ . If the transition  $t_1 = ((3, 1), \{p\}, (0, 0))$  is processed before  $t_2 = ((3, 1), \emptyset, (2, 1))$ , then the cycle  $C_1 = [(0, 0) \rightarrow (1, 0) \rightarrow (2, 1) \rightarrow (3, 1) \rightarrow (0, 0)]$  is detected and the four states are merged in the same SCC before exploring  $t_2$ . After this merge (line 19), this SCC is at the top of the SCC stack. Subsequently, when the DFS explores  $t_2$ , the merge caused by the cycle  $C_2$  does not add any new state to the SCC, and the SCC stack remains unchanged. Therefore, the test line 21 still return false because the union  $SCC.top().k$  of all change-sets labeling the transitions of  $S$  is not empty (it includes for example  $t_1$ 's label:  $\{p\}$ ). The

```

1 Data: todo: stack of  $\langle state \in S, succ \subseteq S \rangle$ 
      SCC: stack of  $\langle root \in \mathbb{N}, rem \subseteq S \rangle$ 
      H: map of  $S \mapsto \mathbb{N}$ 
      max  $\leftarrow 0$ ; init  $\leftarrow I$ 
2 second-pass()
3   while  $\neg init.empty()$  do
4     pick one  $s^0$  off init
5     if  $s^0 \notin H$  then DFSpush2( $\emptyset, s^0$ )
6     while  $\neg todo.empty()$  do
7       if todo.top().succ =  $\emptyset$  then
8         | DFSpop()
9       else
10        pick one d off todo.top().succ
11        if  $d \notin H$  then
12          | DFSpush2(d)
13          else if  $H[d] > 0$  then
14            | merge2( $H[d]$ )
15            | if ( $d \in G$ ) then return false
16        return true
17 DFSpush2( $s \in S$ )
18   max  $\leftarrow max + 1$ 
19   H[s]  $\leftarrow max$ 
20   SCC.push( $\langle max, \emptyset \rangle$ )
21   todo.push( $\langle s, \{d \in S \mid (s, \emptyset, d) \in R\} \rangle$ )
22   init  $\leftarrow init \cup \{d \in S \mid (s, k, d) \in R, k \neq \emptyset\}$ 
23 merge2( $t \in \mathbb{N}$ )
24   r  $\leftarrow \emptyset$ 
25   while  $t < SCC.top().root$  do
26     | r  $\leftarrow r \cup SCC.top().rem$ 
27     | SCC.pop()
28     | SCC.top().rem  $\leftarrow SCC.top().rem \cup r$ 

```

**Algorithm 3.** The second-pass of the TA emptiness check algorithm

first-pass algorithm then terminates without reporting any accepting cycle, missing  $C_2$ .

Had the first-pass processed  $t_2$  before  $t_1$ , it would have merged the states  $(3, 1)$  and  $(2, 1)$  in an SCC, and would have detected it to be livelock-accepting.

In general, to report a livelock-accepting cycle, the first-pass computes the union of all change-sets of the SCC containing this cycle. However, this union may include non-stuttering transitions belonging to other cycles of the SCC. In this case, the second-pass is required to search for livelock-acceptance cycles, ignoring the non-stuttering transitions that may belong to the same SCC.

The second-pass (Algorithm. 3) is a DFS exploring only stuttering transitions (line 21). To report a livelock-accepting cycle, it detects “stuttering-SCCs” and tests if they contain a livelock-accepting state (line 15).

Ignoring the non-stuttering transitions during the DFS, may lead to miss some parts of the product so any destination of a non stuttering transition is stored in *init* for later exploration (line 22).

In the algorithm proposed by Geldenhuys and Hansen [10], the first pass uses a heuristic to detect livelock-acceptance cycles when possible. This heuristic detects more livelock-acceptance cycles than Algorithm. 2. In certain cases this first pass may still fail to report some livelock-acceptance cycles. Yet, this heuristic is very efficient: when counterexamples exist, they are usually caught by the first pass, and the second is rarely needed. However, when properties are verified, the second pass is always required.

**Optimizations.** In our experimentation, we implement the algorithm proposed by Geldenhuys and Hansen [10] including the heuristic and we have added some improvements to the first pass:

1. If no livelock-acceptance state is visited during the first pass, then the second pass can be disabled: this is the purpose of variable  $Gseen$ . In our experiments, this optimization greatly improves the performance of the TA approach in the cases where the formula is verified.
2. A cycle detected during the first pass is also accepted if it contains a livelock-acceptance state  $(s_{\mathcal{X}}, s_{\mathcal{T}})$  such that  $s_{\mathcal{T}}$  has no successor. Indeed, from this state, a run can only executes stuttering transitions. Therefore, a cycle containing this state, is composed only of stuttering transitions: it is a livelock accepting cycle.

## E Proofs for TGTA Construction

*Proof of property 4*  $\square$

$(\subseteq)$  Let  $w = k_0 k_1 k_2 \dots \in \mathcal{L}(\mathcal{G})$  be an execution accepted by  $\mathcal{G}$ . By Def. 3 this execution is recognized by a path  $(s_0, K_0, F_0, s_2)(s_2, K_1, F_1, s_2) \dots \in R_{\mathcal{G}}^0$  of  $\mathcal{G}$ , such that  $s_0 \in I$ ,  $\forall i \in \mathbb{N}$ ,  $(k_i \in K_i)$ , and  $\forall f \in F$ ,  $\forall i \in \mathbb{N}$ ,  $\exists j \geq i$ ,  $f \in F_j$ . By applying (ii) and (i), we can see that there exists a corresponding path  $((s_0, k_0), k_0 \oplus k_1, F_0, (s_1, k_1))((s_1, k_1), k_1 \oplus k_2, F_1, (s_2, k_2)) \dots \in R_{\mathcal{T}}^0$  of  $\mathcal{T}$  such that  $(s_0, k_0) \in I_{\mathcal{T}}$ ,  $k_0 \in U_{\mathcal{T}}((s_0, k_0))$ , and still  $\forall f \in F$ ,  $\forall i \in \mathbb{N}$ ,  $\exists j \geq i$ ,  $f \in F_j$ . By Def. 6 we therefore have  $w \in \mathcal{L}(\mathcal{T})$ .

$(\supseteq)$  Let  $w = w_0 w_1 w_2 \dots \in \mathcal{L}(\mathcal{T})$  be an execution accepted by  $\mathcal{T}$ . By Def. 6 this execution is recognized by a path  $((s_0, k_0), w_0 \oplus w_1, F_0, (s_1, k_1))((s_1, k_1), w_1 \oplus w_2, F_1, (s_2, k_2)) \dots \in R_{\mathcal{T}}^0$  of  $\mathcal{T}$  such that  $(s_0, k_0) \in I_{\mathcal{T}}$ ,  $w_0 \in U_{\mathcal{T}}((s_0, k_0))$ , and  $\forall f \in F$ ,  $\forall i \in \mathbb{N}$ ,  $\exists j \geq i$ ,  $f \in F_j$ . Of course we have  $w_i \oplus w_{i+1} = k_i \oplus k_{i+1}$  but this does not suffice to imply that  $k_i = w_i$ . However (i) tells us that  $w_0 \in U_{\mathcal{T}}((s_0, k_0)) = \{k_0\}$  so  $w_0 = k_0$ , and since  $w_i \oplus w_{i+1} = k_i \oplus k_{i+1}$  it follows that  $w_i = k_i$ . By applying (ii) can now find a corresponding path  $(s_0, K_0, F_0, s_2)(s_2, K_1, F_1, s_2) \dots \in R_{\mathcal{G}}^0$  of  $\mathcal{G}$ , such that  $s_0 \in I$ ,  $\forall i \in \mathbb{N}$ ,  $(w_i = k_i \in K_i)$ , and  $\forall f \in F$ ,  $\forall i \in \mathbb{N}$ ,  $\exists j \geq i$ ,  $f \in F_j$ . By Def. 3 we therefore have  $w \in \mathcal{L}(\mathcal{G})$ .  $\square$

*Proof of property 5*  $\square$

1.  $(\mathcal{T}' \supseteq \mathcal{T})$  Obvious because we are only adding transitions.  $(\mathcal{T}' \subseteq \mathcal{T})$  Let  $R' = R \cup \{(q, \emptyset, F, q) \mid q \in Q\}$ . Consider an accepting execution  $w = k_0 k_1 k_2 \dots \in \mathcal{L}(\mathcal{T}')$  recognized by an accepting path  $\pi'$  on  $\mathcal{T}'$ . Any transition of  $\pi'$  that is not in  $R$  is a self-loop  $(q, \emptyset, F, q)$  that has been added to  $R'$  because an accepting stuttering-SCC exists in  $R$  around  $q$ : so any  $(q, \emptyset, F, q) \in R'$  can be replaced by a sequence of stuttering transitions  $(q, \emptyset, G_0, q_1)(q_1, \emptyset, G_1, q_2) \dots (q_n, \emptyset, G_n, q) \in R^*$  such that  $G_0 \cup G_1 \cup \dots \cup G_n = F$ . The path  $\pi \in R^0$  obtained by replacing all such transitions is an accepting path of  $\mathcal{T}$  that recognizes a word that is stuttering equivalent to  $w$ . Since  $\mathcal{L}(\mathcal{T})$  is stuttering-insensitive, it must also contain  $w$ .  $\square$
2.  $(\mathcal{T}'' \supseteq \mathcal{T})$  Obvious for the same reason.  $(\mathcal{T}'' \subseteq \mathcal{T})$  We consider the case where  $s_0$  is non initial (the initial case is similar). Let  $R'' = R \cup \{(s, k, f, s_n)\}$ . Consider an accepting execution  $w = k_0 k_1 k_2 \dots \in \mathcal{L}(\mathcal{T}'')$  recognized by a path  $\pi''$  on  $\mathcal{T}''$ . Let  $\pi$  be the path on  $\mathcal{T}$  obtained by replacing in  $\pi''$  any occurrence of  $(s, k, f, s_n) \in (R'' \setminus R)$  by the sequence  $(s, k, f, s_0)(s_0, \emptyset, F_1, s_1)(s_1, \emptyset, F_2, s_2) \dots (s_{n-1}, \emptyset, F_n, s_n) \in R^*$ . The path  $\pi \in R^0$  is also an accepting path of  $\mathcal{T}$  that recognizes a word that is stuttering equivalent to  $w$ . Since  $\mathcal{L}(\mathcal{T})$  is stuttering-insensitive, it must also contain  $w$ .  $\square$

3.  $\mathcal{L}(\mathcal{T}^\dagger) = \mathcal{L}(\mathcal{T})$  by application of the previous two properties, therefore  $\mathcal{L}(\mathcal{T}^\dagger)$  is a stuttering-insensitive language.  $\mathcal{L}(\mathcal{T}''')$  is also a stuttering-insensitive language because  $\mathcal{T}'''$  is obtained from  $\mathcal{T}^\dagger$  that recognizes a stuttering-insensitive language, by adding stuttering self-loops on all its states before removing all stuttering-transitions that are not self-loops.

To prove that two stuttering-insensitive languages are equal, it is sufficient to verify that they contain the same words of the following two forms:

- $w = k_0 k_1 k_2 \dots$  with  $\forall i \in \mathbb{N}, k_i \oplus k_{i+1} \neq \emptyset$  (non-stuttering words), or
- $w = k_0 k_1 k_2 \dots (k_n)^\omega$  with  $\forall i < n, k_i \oplus k_{i+1} \neq \emptyset$  (terminal stuttering words)

All other accepted words can be generated by duplicating letters in the above words. Since we have only touched stuttering transitions, it is clear that the non-stuttering words of  $\mathcal{L}(\mathcal{T})$  are the non-stuttering words of  $\mathcal{L}(\mathcal{T}''')$ .

We now consider the case of a terminal stuttering word  $w = k_0 k_1 k_2 \dots (k_n)^\omega$  with  $\forall i < n, k_i \oplus k_{i+1} \neq \emptyset$ .

( $\mathcal{T}''' \subseteq \mathcal{T}^\dagger$ ) The path  $\pi'''$  that recognizes  $w$  in  $\mathcal{T}'''$  has the form  $(s_0, k_0 \oplus k_1, F_0, s_1)(s_1, k_1 \oplus k_2, F_1, s_2) \dots (s_n, \emptyset, F, s_n)^\omega$  where all transitions are necessarily from  $\mathcal{T}^\dagger$  because we have only added in  $\mathcal{T}'''$  transitions of the form  $(s, \emptyset, \emptyset, s)$ .  $\pi'''$  is thus also an accepting path of  $\mathcal{T}^\dagger$  and  $w \in \mathcal{L}(\mathcal{T}^\dagger)$ .

( $\mathcal{T}''' \supseteq \mathcal{T}^\dagger$ ) The path  $\pi^\dagger$  that recognizes  $w$  in  $\mathcal{T}^\dagger$  does only stutter after  $k_n$ . Because this is an accepting path, it has a lasso-shape, where the cyclic part is only stuttering and accepting. Let us denote it  $\pi^\dagger = (s_0, k_0 \oplus k_1, F_0, s_1)(s_1, k_1 \oplus k_2, F_1, s_2) \dots (s_{n-1}, k_{n-1} \oplus k_n, F_{n-1}, s_n)(s_n, \emptyset, F_n, s_{n+1}) \dots [(s_m, \emptyset, F_m, s_{m+1}) \dots (s_l, \emptyset, F_l, s_m)]^\omega$ , with  $\forall i < n, k_i \oplus k_{i+1} \neq \emptyset$ .

Thanks to property [5.1](#), the accepting cycle  $[(s_m, \emptyset, F_m, s_{m+1}) \dots (s_l, \emptyset, F_l, s_m)]$  of  $\pi^\dagger$  can be replaced by an accepting self-loop  $(s_m, \emptyset, F, s_m)$ . And thanks to property [5.2](#), the transitions from  $s_{n-1}$  to  $s_m$  can be replaced by a single transition  $(s_{n-1}, k_{n-1} \oplus k_n, F_{n-1}, s_m)$ . The resulting path  $\pi''' = (s_0, k_0 \oplus k_1, F_0, s_1)(s_1, k_1 \oplus k_2, F_1, s_2) \dots (s_{n-1}, k_{n-1} \oplus k_n, F_{n-1}, s_m)(s_m, \emptyset, F, s_m)^\omega$  is an accepting path of  $\mathcal{T}'''$  that accepts  $w$ , so  $w \in \mathcal{L}(\mathcal{T}''')$ . □

4. This is a classical optimization on Büchi automata. □



# A Domain Specific Language Approach for Genetic Regulatory Mechanisms Analysis

Nicolas Sedlmajer, Didier Buchs, Steve Hostettler,  
Alban Linard, Edmundo López Bóbeda, and Alexis Marechal

Université de Genève, 7 route de Drize, 1227 Carouge, Switzerland

**Abstract.** Systems biology and synthetic biology can be considered as model-driven methodologies. In this context, models are used to discover emergent properties arising from the complex interactions between components. Most available tools propose simulation frameworks to study models of biological systems. Simulation only explores a limited number of behaviors of these models. This may lead to a biased view of the system. On the contrary, model checking explores all the possible behaviors. The use of model checking in the domain of life sciences is limited. It suffers from the complexity of modeling languages designed by and for computer scientists. This article describes an approach based on Domain Specific Languages. It provides a comprehensible, yet formal, language called GReg to describe genetic regulatory mechanisms and their properties, and to apply powerful model checking techniques on them. GReg's objective is to shelter the user from the complexity of those underlying techniques.

## 1 Introduction

Nowadays, the most widespread practices in biology, to observe the behavior of a living system, are *in vivo* and *in vitro* experiments. There is clearly an emerging field of research including systems biology and synthetic biology where experiments are partially performed *in silico* (*i.e.*, by means of techniques from computer science). In systems biology, the models are used to formalize the knowledge gathered from the study of living systems. Such models allow to predict and identify abnormal or emergent behaviors. This approach eases the understanding of the system and the design of new experiments. The synthetic biology approach covers the design and construction of reliable biological functions or systems not found in nature. In this domain, models are designed from a specification (*i.e.*, a desired behavior) and then used to construct and verify newly engineered systems. This approach is similar to software development.

Investigation through formal models of biological systems is not as widely spread as in other natural sciences such as chemistry and physics. This is partly due to the complexity of the living systems, the strenuousness to formalize biological concepts, and the difficulty to perform experiments on living systems to validate models. [17] presents a detailed survey of the formal languages used to model genetic regulatory systems. Another reason is that building formal models requires

a deep knowledge of formalisms that are often too complex for non-experts. To overcome this difficulty, we propose to use the Domain Specific Language (DSL) approach [13]. A DSL is a language designed to be understandable by a domain expert and, at the same time, translatable into a formal language.

This paper presents the Gene Regulation Language (GReg), a DSL for the modeling of genetic regulatory mechanisms. GReg models are transformed to logical regulatory networks [33,8]. Regulatory networks are abstractions of genetic regulatory mechanisms modeled through biological rules over a set of interconnected modules. The occurrence of interesting events in the biological system is represented as logical properties expressed on the states of these modules. This is similar to the kind of properties computer scientists validate on hardware and software systems (*e.g.*, deadlocks and invariants).

Several authors tackled the verification of biological system's properties using model checking techniques [11]. They rely on the expression of models in a formal language. This makes the model checking approach impractical for people who do not master such formal languages.

For instance, Fages *et al.* [7] introduce the chemical abstract machine BioCham. The purpose of their language is to model interaction networks. This language is more general than strictly necessary for the modeling of regulatory mechanisms. Moreover, to verify properties on such models, the user must design formulae in a temporal logic that is a complex formal language. To harness that problematic, Monteiro *et al.* [21] suggest to develop a query language closer to the domain. The syntax of GReg's query language uses similar properties patterns, namely occurrence/exclusion, consequences and sequences. A major difference between GReg and the other languages that model genetic regulatory mechanisms is that its modeling and querying syntaxes are close to the domain. Furthermore, GReg's modularity leverages the reusability and extensibility of the presented approach. It helps to adapt the language concepts, the concrete syntax as well as the transformation to an underlying computational model to the evolution of the user requirements.

Among the tools available in this domain, the main analysis approach is *simulation*. Simulation is generating and analyzing a limited sample of possible system behaviors. This technique is not convenient when looking for rare or abnormal behaviors (*e.g.*, cancer). A possible approach, in this case, is to use *model checking* instead of simulation. Model checking consists in generating and analyzing the complete set of possible behaviors of a model. Such technique suffers from the huge number of possible states of biological systems. This problem is well-known to the model checking community, where it is called the *state space explosion* [34]. In both cellular interactions and software systems the state space explosion is due to their concurrent nature and the size of the variables domains. Therefore, techniques developed for the model checking of hardware and software systems are adapted to biological interactions. For instance, the symbolic encoding of the state space [6].

In this paper we show a work in progress in our group. We present GReg, an application of the DSL approach for modeling and analyzing biological

systems based on formal modeling and reasoning. Advanced techniques for defining Domain Specific Languages, giving their semantics and analyzing them using symbolic model checking are presented. The paper is organized as follows:

First, we describe in Section 2 the usage and creation of Domain Specific Language. Then, in Section 3, we put our work in context by describing the related work. Subsequently, in Section 4 we delineate precisely the part of the biological domain that GReg covers. The following two sections describe GReg. In the first place, we describe the structure of the language in Section 5 and then we outline a translation from GReg to Petri Nets and mention an example of a possible extension in Section 6. A complete yet simple example is presented in Section 7. Finally, we present the future work and the conclusion in Section 8.

## 2 DSL Approach

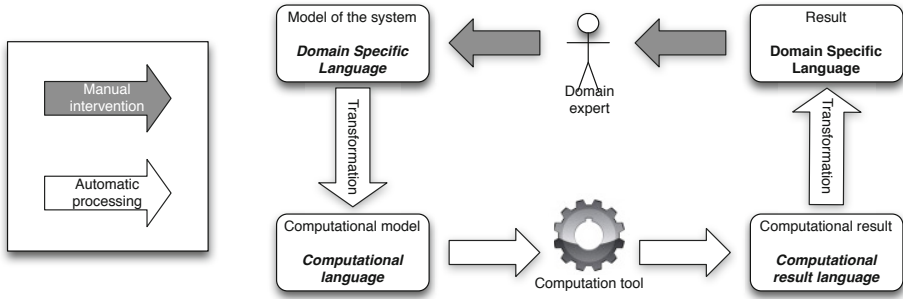
A DSL is a programming or specification language tailored for a given domain; it presents a reduced set of terms closely related to this domain. This approach has been very effective in domains such as software engineering, finance, and physics. Using a DSL has three main objectives. First, learning the language should be easy for someone with enough domain knowledge, even if this person does not have previous knowledge of programming or specification languages. Second, the number of errors made by a novice user should be drastically reduced as the expressivity of the language is reduced to the minimum necessary. Third, creating a DSL is a simpler task than creating a full language, as there exists many tools supporting the approach (*e.g.*, Eclipse Modeling Project (EMP)[10]).

In our approach, presented in Figure 1, the DSL semantics are defined by transformation into a formal language where complex operations (*e.g.*, model checking) can be performed. Often a pre-existing language might be selected to this end. Usually, the scope of the target language is broader than the scope of the DSL. This allows using the same target language and its associated tools for different DSLs. The results obtained in the computational language are translated back to the DSL and returned to the user.

After the creation of the initial model, all the following steps must be fully automated, to hide the underlying complexity from the end user. In the case of model checking, the computation tool in Figure 1 is a model checker, and the result is the answer of whether the system verifies or not a given property, along with an eventual counter-example. In this case, the model checker will return a counter-example specified in its own language. An automatic transformation must translate this result back to the domain of the user, to effectively shield him from the possibly complex computational formal language.

### 2.1 The Design of DSLs

The person in charge of the DSL creation is a language engineer. This person should obviously have a certain knowledge about the language creation process, but he/she should also master the computational language, to define correct



**Fig. 1.** DSL computational process

and efficient transformations. Furthermore, he/she should be in contact with at least one domain expert, to settle the requirements and verify the correctness and completeness of the language created. The creation of a DSL follows a set of specific steps. First, the language engineer identifies the abstract concepts of the domain. These concepts include the basic elements of the domain, the interactions between these elements and the precise boundaries of the considered domain. Based on these concepts, the language engineer defines a set of expressions used to create a specific model based on these abstract concepts. This is called the concrete syntax. Finally, the language engineer sets forth the semantics of the language, usually by transformation to an existing language. Domain experts validate the three steps of the DSL creation process: the domain must be correctly defined, the expressions of the concrete syntax must be close to the already existing languages in the domain, and the execution must return the expected results. This creation/validation process often leads to an iterative development of the language.

A main advantage of a modular structure like the one in Figure 1 is its flexibility. We could replace the computational domain while keeping exactly the same language. For instance, GReg models are currently translated to Petri Nets [27] (see section 6). We could define a transformation from GReg to Systems Biology Markup Language (SBML), without changing the language. As SBML takes text files for input, we would need a tool to produce text files from GReg models, like the model to text transformation tool XPand [10]. Moreover, DSLs can be extended using metamodeling composition techniques, as in [19]. Such extension mechanisms allow the creation of a core DSL, with a minimum of expressivity, and a set of possible extensions that are applied to build more complex languages. Note that such extensions must work both on the specification language level and in the transformations. This eases the creation of new related languages, as most of the basic notions that were previously defined are reused. In Section 5, we present the current version of GReg. The language is simple, but it may be used as a building block for more complex languages.

## 2.2 The Design of a DSL for Biological Process

There are various DSLs tailored to the biological processes, *e.g.*, SBML [14] and SBGN [22]. These two well-known languages cover a wide range of systems, mainly in the bio-chemical domain. GReg, instead, focuses on a more precise specification of the genetic regulatory mechanisms as described in Section 4.

While creating GReg, we followed the design steps mentioned before according to the biological mechanisms. For its implementation we used the Eclipse Modeling Project [10] approach, which includes tools like the Eclipse Modeling Framework (EMF), XText, the Atlas Transformation Language (ATL) and XPand. We first created a meta-model of the domain using EMF, and we defined a concrete syntax with XText. XText provides a set of tools to create an editor for a given language, with some user-friendly features such as syntax highlighting, on the fly syntax checking and auto-completion. As a computational platform we chose AIPiNA[5]. AIPiNA is a model checking tool for Algebraic Petri Nets (APNs) [35]. It aims to perform efficient model checking on models with extremely large state spaces, using  $\Sigma$  Decision Diagrams (DDs) [3] to tackle the state explosion problem. AIPiNA's input languages were also defined using the EMP approach. This allowed us to use ATL to define the semantic transformations. GReg is thus fully integrated in the Eclipse/EMP framework.

## 3 Related Work

In this section we compare six well-known tools with our approach. We first define a few criteria such as the kind of analysis, the supported formalism and the supported exchange format.

**Domain Language.** To be productive, the syntax of the input language should be as close as possible to the actual domain of the user. This input language can be textual (like in tools that use SBML [14]) or graphical (like Systems Biology Graphical Notation (SBGN) [22]).

**Simulation & Model Checking.** Although there are many tools adapted to biological process design and simulation, only a few of them allow exhaustive exploration of the state space. While simulation is very useful during model elaboration, an exhaustive search may help to discover pathological cases that would have never been explored by simulation.

**Discrete & Continuous.** Continuous models are often used to model the biological systems, but they are less appropriate to model checking techniques than discrete models. Discrete formalisms allow a complete exploration of the state space while preserving the qualitative properties of the system, as mentioned in [33].

**Exchange Format.** The supported interchange format is an important feature as it allows us to bridge the gap between different tools and therefore enables the user to use the most adapted tool to hand. SBML is a common interchange format based on XML. It is used to describe biochemical reactions, gene regulation and many other biological processes.

Table 1 presents a comparison of existing tools based on the previous criteria.

**Table 1.** Tool comparison table

	Formalism				Tooling			Available Model Checkers	Exchange Format
	Close Domain Input Language	Query Language	Discrete	Continuous	Stochastic	Simulation	Model Checking		
GReg	✓	✓	✓	✗	✗	✗	✓	AIPiNA	GReg
GINsim	✓	✗	✓	✗	✗	✓	✓	NuSMV	GINML
GNA	✓	✗	✓	✓	✗	✓	✓	NuSMV	GNAML
BioCham	✗	✗	✓	✓	✓	✓	✓	NuSMV	SBML
Snoopy	✗	✗	✓	✓	✓	✓	✓	IDD-CSL	SBML
Cell Illustrator	✗ <sup>1</sup>	✗	✓	✓	✗	✓	✗		CSML
BioTapestry	✓	✗	✓	✗	✗	✓	✗		SBML

**Gene Interaction Network simulation (GINsim) [25]** is a tool for the modeling and simulation of genetic regulatory networks. It models genetic regulatory networks based on a discrete formalism [9,24]. These models are stored using the XML-based format GINML. The simulation computes a state transition graph representing the dynamical behavior network. GINsim uses a graphical DSL called Logical Regulatory Graph (LRG) [8]. Models in LRG are graphs, where nodes are regulatory components (*i.e.*, molecules and genes) and arcs are regulatory interactions (*i.e.*, activation and repression) between the nodes.

**Genetic Network Analyser (GNA) [1]** is a tool for the modeling and simulation of genetic regulators networks. The simulator uses piecewise-linear differential equations to generate a state transition graph that describes the qualitative dynamics of the network [18]. It also offers model checking.

**Biochemical Abstract Machine (BioCham) [12]** is a software for modeling and analyzing biochemical systems. The system is described using chemical reaction rules. This software proposes different simulators (boolean, differential, and stochastic). It embeds a temporal logic language used to formalize and validate temporal properties [7].

**Cell Illustrator [32]** is an example of a commercial simulation tool for continuous and discrete domains. The graphical formalism is based on Petri Nets, called Hybrid Functional Petri Nets with extensions (HFPNe), which add the notions of continuous and generic processes and quantities [23]. The XML-based exchange file format used in Cell Illustrator is called CSML.

<sup>1</sup> Note that, we do not consider the tool's functionalities, only the modeling language (HFPNe).

**Snoopy [31]** is a framework based on Petri Nets formalisms for the modeling and analysis of biomolecular networks. Several Petri Nets classes are available, namely: PN, Stochastic PN and Continuous PN [28].

**BioTapestry [30]** is a free and open source software to create and refine graphical models of genetic regulatory networks. BioTapestry offers a symbolic representation of genes, their products, and their interactions [20].

Let us compare GReg with the above tools. GReg proposes two key features: it has a built-in query language (GQL), and it proposes more fine grained models of genetic regulatory mechanisms compared to other existing languages or tools. For example LRGs only proposes two types of regulation: activation and repression. SBML also proposes only two types of regulation: trigger (activation), inhibition (repression). The tools depending on these formalisms, like GINsim or GNA, have to use this abstraction. As stated previously, GReg models may be transformed to SBML. Such transformation would induce a loss of information, as GReg defines four types of regulation (see section 4). Note that a transformation from SBML to GReg would also lose some information, as GReg does not handle the kinetic information.

BioCham is a modeling tool that uses chemical concepts that are low-level when applied to the biological domain. Nevertheless, its expressivity is greater than GReg's chemical language. Moreover, BioCham has been thoroughly used and tested. Thus, we could define transformations from our model to BioCham's chemical languages, which would take advantage of BioCham's features while keeping our high-level language concepts.

Pedersen *et al.* [26], similarly to BioCham present a language with detailed transformational semantics to Petri Nets (PNs). However, the syntax of the proposed language looks like a programming language, hence it is not adapted to non expert users.

Snoopy and Cell Illustrator are both tools derived and using the Petri Net formalism. PNs are good modeling formalism when it comes to concurrent and asynchronous processes, however it is not adapted to biologists.

## 4 Chemical and Biological Models Covered by GReg

The purpose of GReg is to model genetic regulatory mechanisms controlling the DNA to protein process. This section gives a definition of the domain covered by the current version of GReg. We will describe later some possible extensions of the language, to cover a broader domain. In this section, we first describe molecules and reactions, the basic building blocks of regulatory mechanisms. Then we describe chemical compartments and their hierarchical organization. Finally, we define the regulatory mechanisms themselves.

### 4.1 Molecules and Reactions

Let  $M$  be a non-empty set of *molecules*. In our case,  $M$  does not include the genome of the system, only the proteome and the molecules taken from the

environment. These molecules are located inside chemical compartments, with a given concentration. In the following, we model the concentration of a molecule in the system as a discrete set of levels, as in [33]. If a molecule has  $n$  distinct regulatory roles, we define  $n + 1$  levels. These levels are naturally ordered by concentration, *i.e.*, a level 2 concentration corresponds to a higher concentration than level 1. As the levels are ordered, we constrain the changes of the molecular level to a discrete unit, *i.e.*,  $\Delta l = 1$ .

Molecules interact with each other by means of *chemical reactions*. A reaction consumes and produces some molecules, respectively called *reactants* and *products*, with the participation of a set of *catalysts*. Formally, we define a chemical reaction as an element of  $(M \rightarrow \mathbb{N}) \times (M \rightarrow \mathbb{B}) \times (M \rightarrow \mathbb{N})$ , where the three components correspond to the reactants, products and catalysts respectively ( $\mathbb{N}$  denotes the natural numbers and  $\mathbb{B}$  the Booleans). For each reactant and catalyst, the associated natural number is the concentration level of the respective molecule needed for the reaction to occur. A simple reaction example is  $CO_2@3 + C \rightarrow CO$ , where we say that if molecules  $CO_2$  and  $C$  have at least concentration 3 and 1 respectively, then they will react and produce molecule  $CO$ . Note that we do not consider stoichiometric coefficients for the reactions as we solely work on the concentration levels. Thus, the operator @ in the reaction does not indicate the number of  $CO_2$  molecules that participate in the reaction, but the minimal concentration of  $CO_2$  required for the reaction to take place. In an actual reaction, a catalyst is both a reactant and product. Instead, for readability reasons, we defined a special category for the catalysts, distinct from the products and reactants. Thus, in a reaction, the three sets of catalysts, reactants and products must be mutually disjoint.

A special kind of reactions, called *transport reactions*, defines the transfer of molecules from one location to another. In this case, we add the location to the definition of the reaction. For instance, we may define  $O_2(X) + C(X) \rightarrow CO(Y)$  to indicate that the two reactants in a location  $X$  produce carbon dioxide in the location  $Y$ . These locations are in fact chemical compartments, defined below.

## 4.2 Chemical Compartments

In our system, molecules are located inside *chemical compartments*, where they undergo a specific set of reactions. We define the abstract notion of a chemical compartment as a pair  $\langle Lvl, R \rangle$ .  $Lvl$  is a function  $M \rightarrow \mathbb{N}$  that defines the concentration level of each molecule inside the compartment (an absent molecule will be associated to a level 0).  $R$  is a set of reactions that take place in the compartment. This means that each compartment has a different set of reactions.

Chemical compartments have a hierarchical organization:

- At the top we consider a cell network, a simple chemical compartment that contains a set of cells. We note a cell network as a tuple  $\nu = \langle Lvl, R, \Phi \rangle$ , where  $\Phi$  is the set of cells in the network.
- A cell  $\phi = \langle Lvl, R, \mu, \Omega \rangle \in \Phi$  contains a possibly empty set of organelles (functional subunits of the cell)  $\Omega$ , and a genetic regulatory mechanism  $\mu$



(defined below). This regulatory mechanism is optional. For instance, an eukaryote cell contains a special organelle, the nucleus, which contains the regulatory mechanism. On the other hand, a prokaryote cell does not have a nucleus, and the regulatory mechanism is directly located in the cell.

- An organelle  $\omega = \langle Lvl, R, \mu \rangle \in \Omega$  is the lowest compartment in the compartment hierarchy. Organelles may also contain a genetic regulatory mechanism  $\mu$ . Examples of organelles with mechanisms are nuclei and mitochondria.

Cells and organelles are separated from their environment by a membrane (*i.e.*, selective barrier) allowing molecule transfer between them. Note that this model allows the construction of currently non observed cells, like prokaryote cells with nucleus, or eukaryote cells with multiple nuclei. The validity of the specification is delegated to the biologist (*i.e.*, domain expert).

### 4.3 Genetic Regulatory Mechanisms

A *gene* is a portion of genomic sequence (DNA or RNA) that stores the information relative to the production of a given set of molecules. The production of the molecules coded in a gene is regulated by some other molecules: some of them increase the rate of production, some others decrease it, and some modify the set of molecules produced. For this, the gene contains regulation sites, where specific molecules are bound to perform the regulation. Our idealized gene structure (Figure 2) is composed of two regions: promoter and transcribed regions. Formally, we define a gene  $\gamma$  as a pair  $\langle M_\gamma, \Sigma \rangle$  where  $M_\gamma \subseteq M$  is the non-empty set of molecules coded by the gene, and  $\Sigma$  is a set of regulation sites.

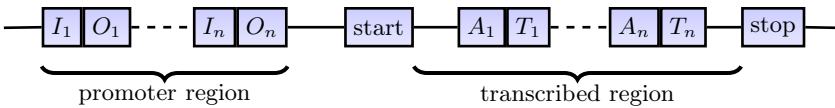


Fig. 2. Idealized gene structure

There are four types of regulatory sites. To illustrate each one of them, we will use a small example of a gene  $\gamma$  that encodes a molecule  $m$  with the regulation of four molecules  $a, b, c$  and  $d$ .

**Initiation (I)** An initiation site defines which molecules increase the production rate of the encoded molecules in  $M_\gamma$ . Such molecules are called *activators*. A different combination of activators regulates each level of the production rate. This combination is expressed as a Boolean formula over the levels of the activators. For instance, we could define our gene  $\gamma$  to produce molecule  $m$  up to level 1 with the activator  $a$  at level 2, up to level 2 with a combination of activators  $a$  or  $b$ , each at level 3, and up to level 3 with the activator  $a$  and  $b$  at level 3. For the molecule  $m$  to reach a concentration level of 3, it must first reach level 1 (activator  $a$  is needed), then level 2 (one of  $a$  or  $b$  is needed), and finally level 3 (both  $a$  and  $b$  are needed). Initiation sites are located in the promoter region of the gene.

**Operator (O)** An operator site defines the molecules that block the production of the encoded molecules. These molecules are called *repressors*. Unlike initiation sites, only one combination of molecules regulates an operator site. In our example, if  $c$  is a repressor of  $\gamma$ , molecule  $m$  will not be produced at all, independently of the activators, and its concentration will decrease in time. Operator sites are located in the promoter region.

**Termination (T)** A termination site interrupts the transcription process when the correct molecules are bound, hence modifying the set of produced molecules. Termination sites are located in the transcribed region. In our example, we may have a termination site intended for molecule  $d$ . If  $d$  is present at the right concentration level, the transcription process of  $\gamma$  is interrupted and a new molecule  $n$  is produced instead of  $m$ .

**Anti-termination (A)** The action of anti-termination sites is the only regulation considered in this paper that takes into account the physical position (*i.e.*, locus) of genes inside chromosomes. If the correct set of molecules is bound to an anti-termination site, the transcription of the next gene in the chromosome is enabled in addition to the current gene. Anti-termination sites are located in the transcribed region.

This definition allows to define currently non observed regulation site positions, as we do not impose an order between initiation and operator sites. The same applies for termination and anti-termination sites. This allows to design artificial genes, like in synthetic biology. Genes may be organized in chromosomes. A *chromosome* is defined by a sequence of *loci*, where each locus contains a sequence of genes. Note that the same gene may be present on one or more chromosomes, and also at different loci of the same chromosome. Moreover, not all genes are in a chromosome (*e.g.*, a virus). Thus, we define a *genetic regulatory mechanism* as a pair  $\langle C, \Gamma_\mu \rangle$ , where  $C$  is a set of chromosomes (which themselves contain a set of genes) and  $\Gamma_\mu$  is a set of genes that are not located in a chromosome. Regulatory mechanisms are contained in cells and organelles. To sum up, a genetic regulatory mechanism describes the production of molecules coded by genes. The molecules, produced from a given gene or taken from the environment, can regulate the production of other genes or themselves, forming a complex relation system. As stated before, having a precise syntax of the system, like the one we gave in this section, is necessary to perform model checking. For non-expert users, manipulating such mathematical notations can be a strenuous task. The objective of GReg is to help the users to express their systems in a non-ambiguous notation, using the domain vocabulary.

## 5 GReg: Gene Regulation Language

GReg is composed of three specific DSLs tailored for genetic regulatory mechanisms. This section describes these three languages. Two of them are used to specify models of the biological domain defined in Section 4. First we present GReg Mechanism Language (GML), the language used to define the genetic regulatory mechanisms, and then we show GReg Network Language (GNL), used to model the location of regulatory mechanisms in hierarchical structures of

chemical compartments. Finally we introduce GReg Query Language (GQL), used to specify the queries to be executed in the models specified with GReg, in order to verify their expected properties. Note that the full definition of the languages is to be found in [29].

To present GML, we use an excerpt of the lac operon model taken from [15]. Listing 1 shows the overall structure of a GReg mechanism specification.

The mechanism is named (`lac_operon`). It specifies the **molecules** occurring in the mechanism, and the chemical **reactions** among these molecules. The GReg description also specifies the **genes** with their properties and organization into **chromosomes**. The **use** keyword is used to import a mechanism from another file, allowing us to reference the molecules and genes declared in the imported mechanism. This allows an easy instantiation of particular combinations of mechanisms.

The **molecules** section specifies the molecules occurring in the mechanism. For instance, in Listing 2, the `lac_operon` mechanism uses molecules `lactose`, `allolactose`, `lacI`, `lacZ`, `lacY`, `lacA`, `cAMP`, `CAP`, *etc.* Molecules are only described by their names, as it is the only information relevant in our language.

```
mechanism lac_operon is
  molecules
    lactose, allolactose,
    lacI, lacZ, lacY, lacA,
    cAMP, CAP
    ...
end lac_operon
```

**Listing 2.** Molecules declaration

ation, a GML description specifies the chemical reactions that take part in the mechanism. Listing 3 presents the **reactions** that take part in the lac operon mechanism.

```
mechanism lac_operon is
  ...
  reactions
    induction : lacI + allolactose → _
    allo      : lactose @ 2 → allolactose cat lacZ @ 2
    ...
end lac_operon
```

**Listing 3.** Reactions declaration

```
mechanism lac_operon is
  use --- use of a mechanism
  --- use of other mechanisms
  molecules
  --- declaration of molecules
  reactions
  --- declaration of reactions
  chromosomes
  --- declaration of chromosomes
  gene --- declaration of a gene
  --- declaration of other genes
end lac_operon
```

**Listing 1.** GReg mechanism specification

The DSL approach emphasizes specification of only the required information for the particular domain. No molecules other than the ones described here can be used in the mechanism. This constraint is useful for the user creating a GReg specification, as spelling errors in molecule names are detected. After the molecules declaration,

Each reaction has a name, for instance `induction` in Listing 3. Unlike usual chemical notations, we do not use stoichiometric coefficient, because the concentration of the molecules is modeled as sequences of discrete levels. Instead, the `@` keyword is used to specify the minimal levels of the reactants and catalysts needed for a reaction. For instance, `induction` reaction can only occur if `lactose` and `lacZ` are at least at concentration level 1. As the result of the chemical reaction is to increment the products level, it is not possible to use the `@` keyword with a product. By default, molecule levels are valued 1.

A reaction can be either irreversible ( $\rightarrow$ ) or reversible ( $\leftrightarrow$ ). In our example, the reaction `induction` is a degradation of `lacI` and `allo lactose`. A degradation is an irreversible reaction where products are not interesting, thus the reactants are simply removed from the system. Each reaction can also have catalysts specified using the keyword `cat`. For instance, `allo` is a reaction catalyzed by `lacZ`.

Listing 4 presents the `genes` specification. For instance, `rep` is a minimal gene (*i.e.*, not regulated). A minimal gene defines at least the molecules it `codes`.

If they are relevant, regulation sites are also specified in a section with the `sites` keyword. The `lac` gene defines a regulated gene with two regulation sites `I` and `O`, together with the molecule acting on them. Note that GReg also allows to define several regulation sites for `I`, `O`, `A`, `T` (Induction, Operator, Anti-termination and Termination sites, see Section 4.3). Each regulation site may define a name which is used to specify the site's properties.

For instance, `I` and `O` sites of the `lac` gene are named `s1` and `s2`.

As presented previously for chemical reactions, the `@` keyword allows the specification of the minimal required molecules levels acting at a regulation site. For instance, molecule `lacI` represses gene `lac` if its concentration level is at least 2. Otherwise, `lacI` has no effect in this gene. As several molecules can act simultaneously on one regulation site, GReg allows to combine molecules with Boolean operators `and`, `or` and `xor` for a regulation site. For instance, the `I` site of gene `lac` specifies that `cAMP` `and` `CAP` are required to activate this site. The `=` operator allows to specify the target level (the level attributed to the gene's products) once this site is active. By default, target levels are valued 1.

The role of a `T` site is to interrupt the transcription process, GReg allows to specify the reduced set of molecules produced when these sites are active.

The `chromosomes` section, illustrated in Listing 5, is used to specify one or more chromosomes. A chromosome defines a sequence of loci, a locus is defined by two braces. Note that the genes order in each locus does not matter.

```

mechanism lac_operon is
...
gene rep
  codes lacI
end rep
gene lac
  codes lacZ, lacY, lacA
  sites
    I s1 : cAMP and CAP = 1
    O s2 : lacI @ 2
end lac
end lac_operon

```

Listing 4. Genes declaration

```

mechanism lac_operon is
  ...
  chromosomes
    c : {rep}, {lac,lac'}
  ...
end lac_operon

```

**Listing 5.** Chromosomes declaration

to describe networks of cells containing these mechanisms. Listing 6 shows the overall structure of a GReg network specification.

A network definition specifies the **molecules** that may be exchanged between its cells, as well as the chemical **reactions** among these molecules. It also describes the transport reactions between a cell and the network. The GNL description also specifies the **cells** along with their properties.

Defining chromosomes is mandatory when using anti-termination (**A**) sites. The definition of an **A** site indicates the next gene target level whose transcription will be enabled if the site is activated.

So far we have seen how to describe genetic regulatory mechanisms using GML. Now we present GNL, the language used

```

network C_elegans is
  molecules
    -- declaration of molecules
  reactions
    -- declaration of reactions
  cell -- declaration of a cell
    -- declaration of other cells
end C_elegans

```

**Listing 6.** GReg network specification

Listing 7 presents the overall structure of a **cell**. The cell is named (**E\_coli**).

```

cell E_coli is
  use -- use of a mechanism
    -- use of other mechanisms
  molecules
    -- declaration of molecules
  reactions
    -- declaration of reactions
  organelle -- decl. of an org.
    -- decl. of other organelles
end E_coli

```

**Listing 7.** GReg cell specification

ilar definition as the one presented for cells in Listing 7. Note that we do not allow organelle nesting inside organelles (see Section 4).

GNL and GML are used to model cell networks and their regulatory mechanisms. The objective of creating such models is to verify some properties on them, with the use of an underlying model checker. For this, we need a language to define such properties. This language is called GQL. Listing 8 shows an example of GQL

```

use "lac_operon.gml"
levels
  11 : lacZ = 1
  12 : lacZ = 0, lacI = 2
queries
  bool a : exists 11
  paths b : paths 11 , 12
  paths c : paths 11 .. 12

```

**Listing 8.** GQL queries specification

specification. A GQL file has two sections: the **levels** and the **queries**. The **levels** section is used to define some combination of molecule levels. In Listing 8, 11 specifies only a level for **lacZ** among all molecules defined in **lac\_operon**. The

**queries** section uses the **levels** defined above to express complete queries. The **exists** query returns true if predefined level exists. The **paths** query is used to retrieve all paths from the state space matching the sequence of predefined levels. In our example, query **b** returns the paths where **12** is a direct successor of **11**. Query **c** returns the paths where **12** is a successor of **11**, without requiring it to be a direct successor.

```
use "lac_operon.gml"
use "lac_operon.gql"
initially lac_operon has
  lactose = 2
  lacI = 1
execute
  if a then (b and c)
```

**Listing 9.** GReg configuration

Models created with GML, GNL and GQL use a GReg configuration specification to define the initial levels of the molecules and the actual queries to be computed by the model checker, from a list defined in a GQL file. This allows to easily repeat experiments for the same mechanism with several initial quantities. The configuration

specification is composed of two sections. Listing 9 shows an example of such specification. The first one starts with the **initially** keyword and defines the genes or molecules initial levels. The second starts with the **execute** keyword and is used to specify which queries will be executed by the model checker.

## 6 Semantics of GReg

As mentioned in Section 2, the semantics of DSLs are defined through transformation into a computational language. For GReg, we implemented a transformation to AlPiNA [5], a model checker for Algebraic Petri Nets (APNs) [35]. After a brief introduction to Petri Net, this section gives some translation patterns from GReg to APNs. Due to space limitations, we avoid here to describe the full transformation. Instead, we limit ourselves to some chosen transformation patterns that show the essence of the approach. The details of the transformation and its implementation are to be found in [29]. Finally, this section sketches briefly a possible extension of the language.

Figure 3 presents a general overview of the transformation from GReg to AlPiNA. Note that, as described in Section 2, DSLs have a modular structure separating the specification language from the computational mechanisms. As described in Section 5, GReg is composed of three languages: GML, which is used to describe the genetic regulatory mechanisms, GNL for the chemical compartments hierarchy (see Section 4.2) and GQL, for the expected properties of the system. Together, the composition of these three languages forms GReg. An automatic transformation is used to produce a PN for AlPiNA from a model created with these languages. The regulatory mechanisms described with GNL are translated to AlPiNA's APNs. The queries expressed with GQL are translated to AlPiNA's properties. On the other hand, AlPiNA does not handle hierarchical models yet, and thus we did not define a translation from GNL files. We are currently working on an extension of AlPiNA to handle hierarchical models. Moreover, another formalism like CO-OPN[2] may be used instead of AlPiNA

to handle models where the hierarchy is important. The possibility to add easily language extensions is a prominent feature of GReg. This matter is briefly discussed in Section 6.3.

AIPiNA is used to perform model checking on the models obtained by transformation, by evaluating the properties on the corresponding APN. It produces a result, which is either a positive answer (the model satisfies the property), or a counterexample, *i.e.*, a state of the system that violates the property. To be consistent with the DSL approach described in Section 2, this counterexample must be translated again to the DSL. This means that a marking of an APN must be translated back to a state of a logical regulatory network. For the moment, we did not implement such translation. In Figure 3, the dashed figures are the ones we plan to implement in the future. All other concepts have been implemented, the details can be found in [29].

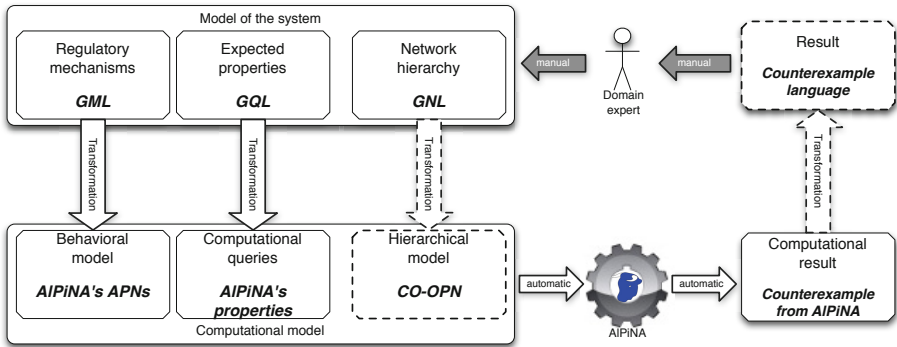


Fig. 3. DSL computational process for Greg using AIPiNA

### 6.1 Petri Nets Presentation

Petri Nets are a graphical formalism adapted to the representation of concurrent and asynchronous processes. They were initially invented for the description of chemical processes, then developed in computer science, and nowadays used in biology in several tools and formalisms. A Petri Net is a bipartite graph composed of places (circles), that represent the local states of processes or resources, and transitions (rectangles) that represent actions.

Places contain tokens, which represent processes or resources. The Petri Nets used in this article (Algebraic Petri Nets [35]) are a particular kind of colored Petri Net: their tokens contain values (integers for instance). Figure 4 shows an example of a Petri Net and all its allowed executions. A transition can be executed (fired) if it can pick one token with the correct value in all its pre-places. It then generates tokens in all its post-places. Figure 4 shows transitions that can be fired using ✓, whereas transitions that cannot be fired are annotated with ✗. Each Petri net of the figure represents one state of the system.

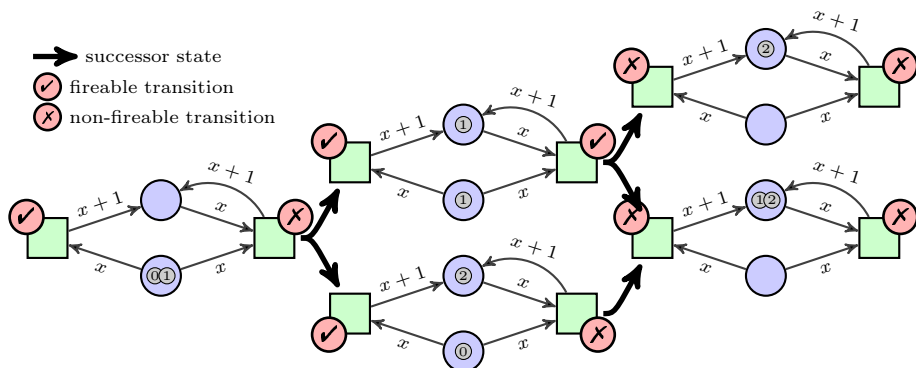


Fig. 4. Example of Petri Net and its possible executions

## 6.2 Transformation Patterns

This section presents the patterns used in the transformations from a GReg specification to APNs. In the nets below, the tokens represent molecules. Their values are only natural numbers, representing the current level of the molecule. For simplicity, we present only patterns where one gene codes for a unique protein. The other cases result in more complex patterns with more transitions.

Transitions are labeled with guards, which are a conjunction of simple conditions. For the sake of readability, the guards are written using usual arithmetic notation instead of ALPiNA's syntax.

Figure 5 presents a gene with two initiation (**I**) sites. The first transition increases the level of **M** from 0 to 1 if activator **B** is present. The third transition does the same from level 1 to 2 if **C** is present. When none of the activators (**B**, **C**) are present, the level of the gene may also be decremented until it reaches zero (second and fourth transitions).

Note when several initiation (**I**) sites are defined with identical levels, they are all combined using logical disjunctions in order to produce a guarded transition.

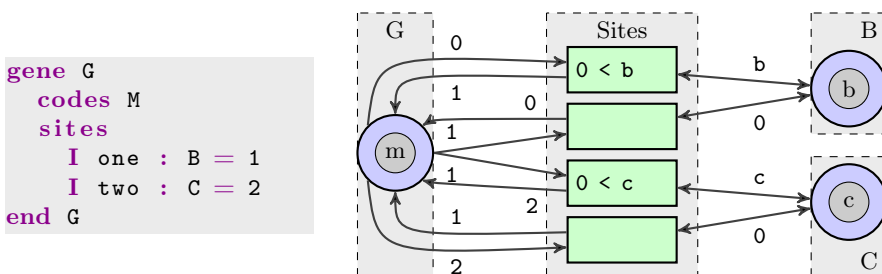


Fig. 5. Pattern 1

Figure 6 presents a gene with one initiation (**I**) site and one operator (**O**) site. When the activator **B** is present **and** the repressor **C** is absent, the level of



```

gene G
  codes M
  sites
    I act : B
    O rep : C
end G
    
```

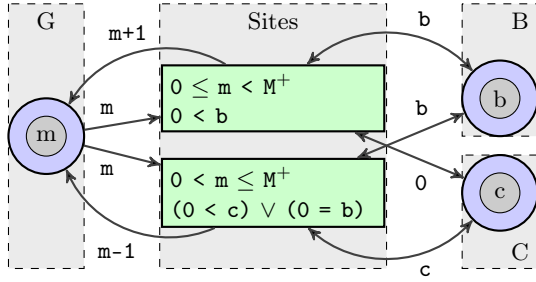


Fig. 6. Pattern 2

M increases. Inversely, the level of the gene decreases when the activator B is absent **or** the repressor C is present.

Figure 7 presents a gene with one termination (T) site. When the repressor B is present ( $0 < b$ ), the product N is produced instead of M.

```

gene G
  codes M
  sites
    T rep : B = N
end G
    
```

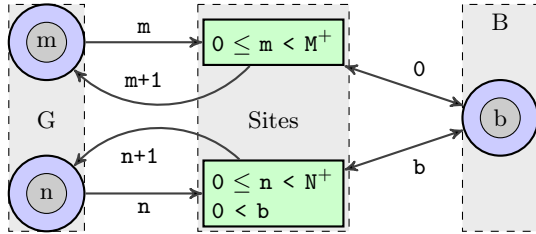


Fig. 7. Pattern 3

Figure 8 presents two genes (G,H) contiguous on the same chromosome. The gene G specifies an anti-termination (A) site, that, when activated, will also allow the transcription of the next gene H. In Figure 8 we omitted the production of gene G as it is not regulated, because this gene does not define a regulation site such as I, O or T and has no preceding gene in chromosome C.

```

chromosomes
  C : { G }, { H }
gene G
  codes M
  sites
    A act : B
end G
gene H
  codes N
end H
    
```

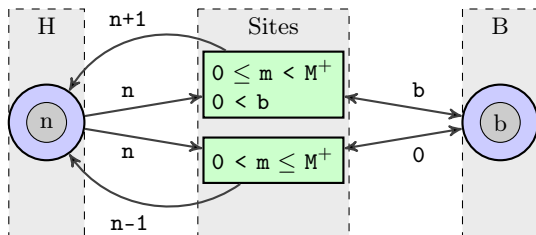


Fig. 8. Pattern 4

### 6.3 Extending GReg

As stated previously, the extensibility of GReg is one of its paramount features. We cannot fully develop this feature in this article. Instead, this section briefly describes one possible extension. As concepts such as time and probabilities play an important role in biology, the kinetic information of the gene transcription and other chemical reactions is a good candidate for a language extension.

As mentioned in Section 3, it is possible to change GReg's computational language. The framework described in Figure 3 is helpful to easily add new concepts to the language while reusing the existing syntactical and semantic elements defined in GReg. Let us sketch how to adapt the language and the transformation to support a new concept such as kinetic information.

1. The first step is to modify the syntax of GReg to support information about the rate of the reactions. For instance, adding a **promoter** keyword to the gene specification, allows to specify the gene transcription rate when no regulation occurs. Similarly, site specifications would be extended to specify transcription rate under regulation. Furthermore, the specification of the reactions should indicate a rate with a keyword **K**.
2. The second step is to choose a new computation model and associated model checker that supports time. As APNs (*i.e.*, the current underlying formalism), do not support timed computations, a choice would be to use timed colored Petri nets and CPN-Tools as a computation tool [16].
3. Finally, the transformation from GReg to the computation language must be changed accordingly. As timed colored Petri nets are similar to APNs this mainly consists in adding more information to the transformed model.

## 7 Example

We present a simple example of a genetic regulatory mechanism with three genes expressed in GReg, taken from [33]. Gene Y is activated by the product of gene X. Genes X and Z are repressed by the products of genes Z and Y respectively. The products of genes X, Y and Z are molecules x, y and z respectively. Figure 9 presents a graphical (LRG) model derived from this example, and Listing 10 presents the equivalent textual model defined in (GReg).

We show one transformation from this model to Petri Nets that produces an APN shown in Figure 10. This APN is obtained by composing three times a pattern similar to the one in Figure 6, by means of fusion of places. Note that the APN in Figure 10 is the folding (an equivalent but more compact model) of the P/T called Multi-level Regulatory Petri net (MRPN) in [8]. Unlike the APNs resulting from our transformations, MRPN only considers the levels of the genes, without taking into account the levels of the molecules. If needed, this abstraction could easily be done on the transformation from GReg to Algebraic Petri Nets, by replacing all the molecules levels with genes levels. As stated before, the DSL approach allows us to define multiple transformations to different computational languages, each with different considerations.

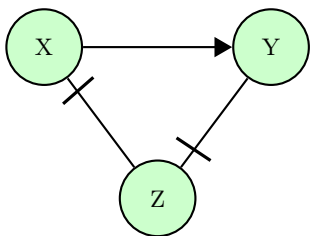


Fig. 9. LRG model of example

```

mechanism example is
  molecules x, y, z
  gene X codes x
    sites O : z
  end X
  gene Y codes y
    sites I : x
  end Y
  gene Z codes z
    sites O : y
  end Z
end example

```

Listing 10. GReg model of Figure 9

From these models AIPiNA is able to compute the state space and to identify all deadlocks without requiring any additional input from the user. A *deadlock* is a situation where no more events can occur in the system. Strictly speaking, in real biological systems there are no deadlocks but *livelocks*, a situation where events still occur but without changing the state of the system. The states where such situations occur are usually called stable states or attractors.

The example in Figure 9 and Listing 10 has eight states reachable from an initial marking in which all molecules are at level zero  $(x,y,z) = (0,0,0)$ . The state transition graph is shown in Figure 11. Model checking of the example finds two stable states:  $(1,1,0)$  and  $(0,0,1)$ .

The user defines the expected system properties with GQL queries. Listing 11 defines five levels (11 to 15) and four queries. The first query returns the set of states where the level of  $x$  and  $y$  is equal to one and stores them in variable  $a$ . To compute this, the query is transformed into an AIPiNA property, shown in Listing 12. The result obtained is the set of states  $\{(1,1,0);(1,1,1)\}$ .

The second query returns the set of paths in which the first state has a level of  $x$  equal to one, and the second state has a level of  $y$  equal to one. The result is stored in variable  $b$ . Four paths are obtained, all with a length of one :  $(1,0,0) \rightarrow (1,1,0)$  ;  $(1,0,1) \rightarrow (1,1,1)$  ;  $(1,1,1) \rightarrow (1,1,0)$  ;  $(1,1,1) \rightarrow (0,1,1)$

```

use "example.greg"
levels
  11 : x = 1, y = 1
  12 : x = 1
  13 : y = 1
  14 : x = 0, y = 0, z = 0
  15 : x = 1, y = 1, z = 1
queries
  states a : at 11
  paths b : paths 12, 13
  paths c : cycles 14 .. 15
  nat d : length shortest c

```

Listing 11. GReg query example

```

s1 : exists($x in x, $y in y :
(($x equals suc(zero)) and ($y equals suc(zero))) = false

```

Listing 12. AIPiNA property expression of **at** query in Listing 11

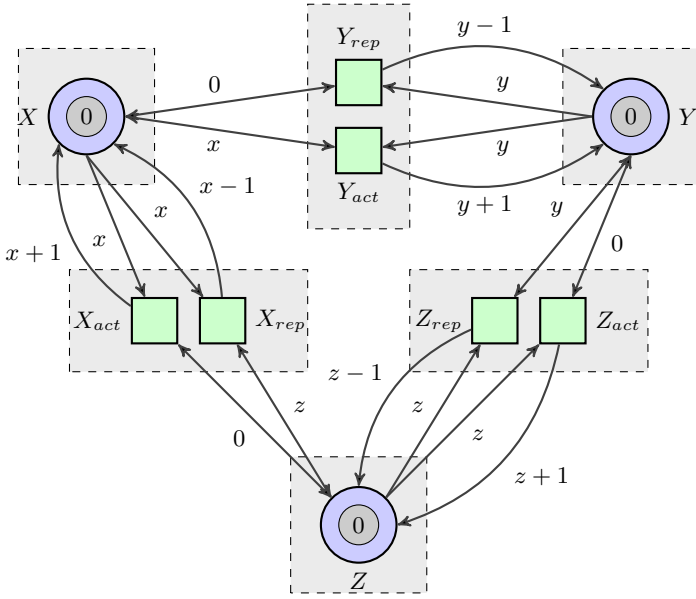


Fig. 10. APN model of example in Figure 9 and Listing 10

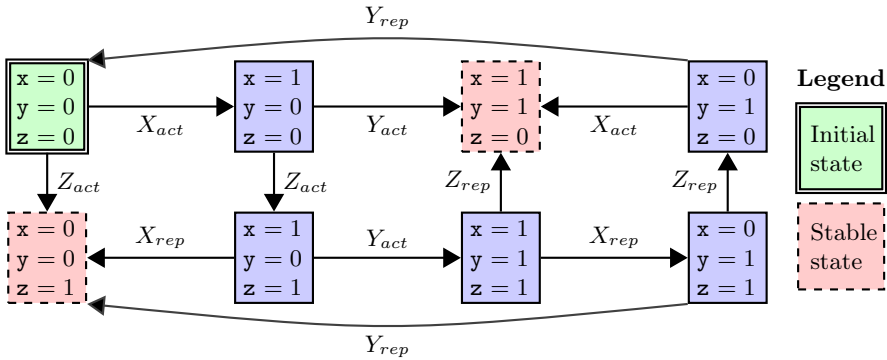


Fig. 11. State transition graph of example in Figure 9 and Listing 10

The third query returns the set of cycles where the first state has a level matching 14, and there exists a subsequent state whose level matches 15. The result is stored in variable  $c$ . Note that the result of the `cycles` query is a set of paths. Only one cycle is obtained :

$(0,0,0) \rightarrow (1,0,0) \rightarrow (1,0,1) \rightarrow (1,1,1) \rightarrow (0,1,1) \rightarrow (0,1,0) \rightarrow (0,0,0) \rightarrow \dots$

The last query is an example where operators are combined to build new queries and reuse previous results. It combines the two operators `length` and `shortest` and applies them on the set of paths  $c$ . This query should be read as "return the length of the shortest path in the given set of paths". In our example, the length of the cycle is six.

## 8 Conclusion and Future Work

This paper presents Gene Regulation Language (GReg), a DSL dedicated to the modeling of genetic regulatory mechanisms. GReg is given as an example of the DSL-based verification process. We describe the creation of a language tailored to the understanding of domain experts, and how this language can be translated into a formal model where model checking can be applied. Model checking is a very well known verification technique used in computer science. Its main advantage over simulation is the complete exploration of the state space of the model, thus allowing to discover rare but potentially interesting events. A query language to express the properties of such events is embedded with GReg.

The languages and transformations shown in this article have been implemented and tested on several toy examples. We also asked biologists to assess the expressivity and usability of the language. Although the first feedback seems promising, there is much room for improvement. We foresee three main axes of future development: improving the expressivity of the modeling and query languages, assessing the usability of the approach and exploring the mitigation of the state space explosion.

**Extending the Expressivity of GReg.** Concepts such as time and probabilities (*e.g.*, kinetic information) play an important role in biology and are therefore good candidates for a language extension. As briefly sketched in Section 6.3, the framework described in Section 2 is very helpful to easily add new concepts to the language while reusing the existing syntactical and semantic elements defined in GReg. As there exists no model checker that is yet capable of managing every dimensions (*e.g.*, time, probabilities, complex data types) we have to choose some of the dimensions and to project the GReg model on the chosen target computational model.

**Improving its Usability.** Textual domain specific languages constitute a first step towards democratization of formal methods. Although highly efficient, textual languages are usually not as intuitive as graphical languages. On the other hand, graphical domain specific languages are especially good in the early phase of the modeling and for documentation, but they are often less practical when the model grows. The tools in EMP that we used to create GReg allow us to define a graphical version of the same language, thus keeping the best of both worlds. Another way to ease the modeling phase is to allow import/export of models from/to other formalisms and standards such as SBML. Finally, as it is important for the end user to fully understand and trust the model, we will add simulation in addition to model checking. As it focuses on a restricted set of behaviors, simulation is faster than model checking. This is especially useful while designing the model.

**Mitigating the State Space Explosion.** So far, we have done little experimentation in this area for real biological processes. This weakness is being worked on, using studies found in the literature and adapted to GReg. We are also working on a more detailed comparison to other tools dedicated to biological problems, like GINsim. Nevertheless, we conducted several studies

on usual IT protocols and software models that show that AIPiNA can handle huge state spaces [4]. This suggests promising results in the regulatory mechanisms domain.

The development of GReg is a work in progress, we would like to set up more collaborations with biologists interested in exploiting formal techniques from computer science to discover rare events. We think that we can make, in the near future, a useful contribution to life sciences based on advanced techniques borrowed from computer science.

## References

1. Besson, B., de Jong, H., Monteiro, P., Page, M.: GNA, <http://ibis.inrialpes.fr/article122.html>
2. Buchs, D., Guelfi, N.: A Formal Specification Framework for Object-Oriented Distributed Systems. *IEEE Transactions on Software Engineering* 26(7), 635–652 (2000)
3. Buchs, D., Hostettler, S.: Sigma Decision Diagrams. In: TERMGRAPH 2009: 5th International Workshop on Computing with Terms and Graphs, pp. 18–32. No. TR-09-05, Università di Pisa (2009)
4. Buchs, D., Hostettler, S., Marechal, A., Risoldi, M.: AIPiNA: A Symbolic Model Checker. In: Lilius, J., Penczek, W. (eds.) PETRI NETS 2010. LNCS, vol. 6128, pp. 287–296. Springer, Heidelberg (2010)
5. Buchs, D., Hostettler, S., Marechal, A., Risoldi, M.: AIPiNA: An Algebraic Petri Net Analyzer. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 349–352. Springer, Heidelberg (2010)
6. Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.: Symbolic Model Checking:  $10^{20}$  States and Beyond. *Information and Computation* 98 (1992)
7. Chabrier, N., Chiaverini, M., Danos, V., Fages, F., Schachter, V.: Modeling and Querying Biomolecular Interaction Networks. *Theoretical Computer Science* 325(1), 25–44 (2004)
8. Chaouiya, C., Kludel, H., Pommereau, F.: A Modular, Qualitative Modeling of Regulatory Networks Using Petri Nets. In: *Modeling in Systems Biology, Computational Biology*, vol. 16, pp. 253–279. Springer (2011)
9. Chaouiya, C., Remy, E., Mossé, B., Thieffry, D.: Qualitative Analysis of Regulatory Graphs: A Computational Tool Based on a Discrete Formal Framework. In: Benvenuti, L., De Santis, A., Farina, L. (eds.) *Positive Systems*. LNCIS, vol. 294, pp. 119–126. Springer, Heidelberg (2003)
10. Eclipse Foundation: Eclipse Modeling Project, <http://www.eclipse.org/modeling>
11. Clarke, E.M., Grumberg, O., Peled, D.A.: *Model checking*. MIT Press, Cambridge (1999)
12. EPI Contraintes, INRIA Paris-Rocquencourt: BioCham, <http://contraintes.inria.fr/BioCham/>
13. Fowler, M.: *Domain-Specific Languages*. Addison-Wesley Professional (2010)
14. Hucka, M., Bergmann, F., Hoops, S., Keating, S., Sahle, S., Schaff, J., Smith, L., Wilkinson, B.: *The Systems Biology Markup Language (SBML): Language Specification for Level 3 Version 1 Core* (2010), available from Nature Precedings

15. Jacob, F., Monod, J.: Genetic Regulatory Mechanisms in the Synthesis of Proteins. *Journal of Molecular Biology* 3, 318–356 (1961)
16. Jensen, K., Kristensen, L., Wells, L.: Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer (STTT)* 9, 213–254 (2007)
17. Jong, H.D.: Modeling and simulation of genetic regulatory systems: a literature review. *Journal of Computational Biology* 9(1), 69–105 (2002)
18. Jong, H.D., Geiselman, J., Hernandez, C., Page, M.: Genetic Network Analyzer: qualitative simulation of genetic regulatory networks. *Bioinformatics* 19(3), 336–344 (2003)
19. Pedro, L.: A Systematic Language Engineering Approach for Prototyping Domain Specific Languages. Ph.D. thesis, Université de Genève (2009), thesis # 4068
20. Longabaugh, W., Davidson, E., Bolouri, H.: Visualization, documentation, analysis, and communication of large-scale gene regulatory networks. *Biochimica et Biophysica Acta (BBA) - Gene Regulatory Mechanisms* 1789(4), 363–374 (2009)
21. Monteiro, P.T., Ropers, D., Mateescu, R., Freitas, A.T., de Jong, H.: Temporal Logic Patterns for Querying Qualitative Models of Genetic Regulatory Networks. *Bioinformatics* 24(16), i227–i233 (2008)
22. Moodie, S., Novere, N.L., Demir, E., Mi, H., Villeger, A.: Systems Biology Graphical Notation: Process Description language Level 1 (2011), available from Nature Precedings, <http://dx.doi.org/10.1038/npre.2011.3721.4>
23. Nagasaki, M., Doi, A., Matsuno, H., Miyano, S.: A Versatile Petri Net Based Architecture for Modeling and Simulation of Complex Biological Processes. *Genome Informatics* 15(1), 180–197 (2004)
24. Naldi, A., Berenguier, D., Fauré, A., Lopez, F., Thieffry, D., Chaouiya, C.: Logical Modelling of Regulatory Networks with GINsim 2.3. *Biosystems* 97(2) (2009)
25. Naldi, A., Chaouiya, C., Thieffry, D.: GINsim, <http://gin.univ-mrs.fr/>
26. Pedersen, M., Plotkin, G.D.: A Language for Biochemical Systems: Design and Formal Specification. In: Priami, C., Breitling, R., Gilbert, D., Heiner, M., Uhrmacher, A.M. (eds.) *Transactions on Computational Systems Biology XII*. LNCS, vol. 5945, pp. 77–145. Springer, Heidelberg (2010)
27. Reisig, W.: Petri nets. An Introduction. *EATCS Monographs on Theoretical Compute Science*, vol. 4. Springer (1985)
28. Rohr, C., Marwan, W., Heiner, M.: Snoopy - a unifying Petri net framework to investigate biomolecular networks. *Bioinformatics* 26(7), 974–975 (2010)
29. Sedlmajer, N.: GReg: a domain specific language for the modeling of genetic regulatory mechanisms. Master's thesis, University of Geneva (2012), <https://smv.unige.ch/student-projects/finished-projects/files/sedlmajer/>
30. Institute for Systems Biology and the Davidson Lab at Caltech: BioTapestry, <http://www.BioTapestry.org/>
31. Brandenburg University of Technology: Snoopy, <http://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Snoopy>
32. The University of Tokyo: Cell Illustrator, <http://www.cellillustrator.com/>
33. Thomas, R.: Regulatory Networks seen as Asynchronous Automata: a Logical Description. *Journal of Theoretical Biology* 153, 1–23 (1991)
34. Valmari, A.: The State Explosion Problem. In: Reisig, W., Rozenberg, G. (eds.) *APN 1998*. LNCS, vol. 1491, pp. 429–528. Springer, Heidelberg (1998)
35. Vautherin, J.: Parallel Systems Specifications with Coloured Petri Nets and Algebraic Specifications. In: Rozenberg, G. (ed.) *APN 1987*. LNCS, vol. 266, pp. 293–308. Springer, Heidelberg (1987)

# Verifying Parallel Algorithms and Programs Using Coloured Petri Nets

Michael Westergaard

Department of Mathematics and Computer Science,  
Eindhoven University of Technology, The Netherlands  
`m.westergaard@tue.nl`

**Abstract.** Coloured Petri nets have proved to be a useful formalism for modeling distributed algorithms, i.e., algorithms where nodes communicate via message passing. Here we describe an approach for automatic extraction of models of parallel algorithms and programs, i.e., algorithms and programs where processes communicate via shared memory. The models can be verified for correctness, here to prove absence of mutual exclusion violations and to find dead- and live-locks. This makes it possible to verify software using a model-extraction approach using coloured Petri nets, where a formal model is extracted from runnable code. We extract models in a manner so we can also support a model-driven development approach, where code is generated from a model, enabling a combined approach, supporting extracting a model from an abstract description and generation of correct implementation code. We illustrate our idea by applying the technique to a parallel implementation of explicit state-space exploration.

Our approach builds on having a coloured Petri net model corresponding to the program and using the model to verify properties. We have already treated generation of code from coloured Petri nets, so in this paper we focus on the translation the other way around. We have an implementation of the translation from code to coloured Petri nets.

## 1 Introduction

Parallel and distributed computing address important problems of scalability in computer science, where some problems are too large or complex to be handled by just one computer. Until now, the focus has mostly been on distributed algorithms, i.e., algorithms running on multiple computers communicating via a network, as access to parallel computers, i.e., computers capable of running multiple processes communicating via shared memory (RAM), has been limited. For this reason, there are many papers on modeling distributed algorithms, such as network protocols [1-4]. With the advance of cheap multi-core processors and cheap multi-processor systems, access to multiple cores has become more common, and the development and analysis of algorithms for parallel processing becomes very interesting. As parallel computing allows much faster communication between processes, tasks that were not previously feasible or efficient to do



concurrently, become interesting. In this paper, we present our experiences developing parallel algorithms with synchronization mechanisms. The algorithms are verified by means of *coloured Petri nets* (CPNs) [5]. This work was motivated by our need for a parallel state-space exploration algorithm. In this paper, we provide an approach that allows us to extract a model for analysis from a program or abstractly described algorithm in a systematic way. We do this in a way that allows us to automatically generate a (skeleton) implementation of the algorithm subsequently. We use a simple state-space algorithm as an example, but the approach has also been used for other parallel algorithms, such as parts of a protocol for operational support [6], and is generally applicable for other parallel algorithms as well. The method can also be used for non-parallel algorithms, but we focus on the new challenges arising when moving from sequential to parallel processing.

Classically, formal models can partake in a development in two different ways: by extracting an implementation from a model, which we call *model-driven software engineering*, or by extracting a model from an implementation, which we call *model-extraction*. Our focus in this paper is on model-extraction but in a way that allows us to also do code generation, thereby allowing a new combined approach. The model-driven engineering approach is shown in Fig. 1 (top) and shows that we start with a model that is verified according to one or more properties. If it satisfies the desired properties, we can extract a program, otherwise we refine the model. Examples of this approach are within hardware synthesis [7,8], using a CPN simulator to drive a security system [9], or general code generation from a restricted class of CPNs [10]. The model-extraction approach is shown in Fig. 1 (bottom). Here, we do not start with a model, but rather with a program. From the program, we extract a model and verify it for correctness.

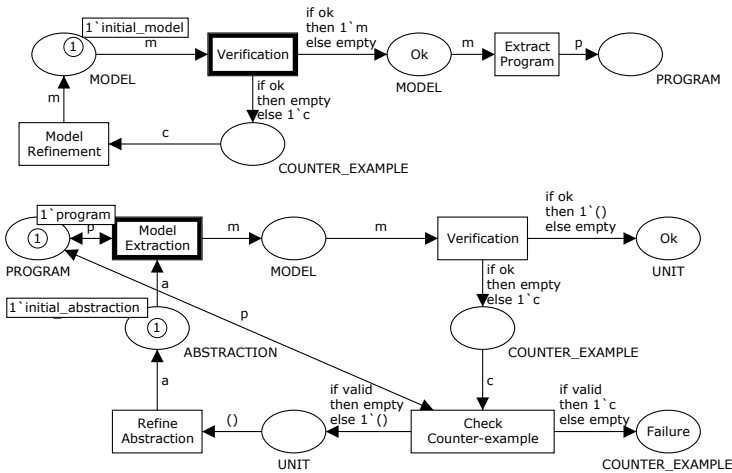


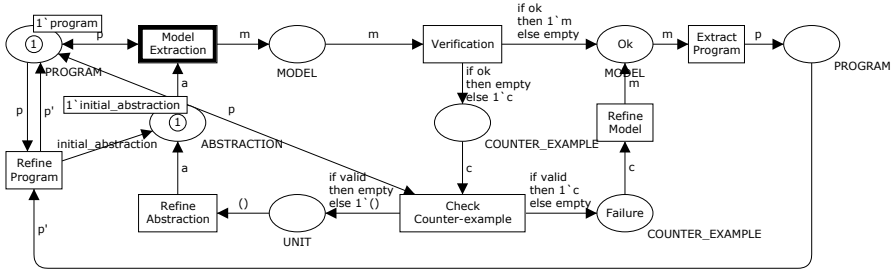
Fig. 1. Model-driven software engineering (top) and model-extraction (bottom)

If an error is found, the resulting error-trace is replayed on the original program to determine if it can be reproduced there. If not, the abstraction used to extract the model is refined and the cycle restarts. This approach is rarely used in the high-level Petri net world, but is employed by, e.g., FeaVer [11] to translate C code to PROMELA code usable in SPIN [12], Java PathFinder [13] to translate Java programs to PROMELA, SLAM [14] for automatically translating C device drivers to boolean programs, BLAST [15] for model-checking C programs, and many other tools.

The model-driven software engineering and model-extraction approaches have different strengths and weaknesses. The main strength of the model-driven software engineering approach is that it is possible to verify an algorithm before implementation and we can even get a guaranteed-correct (template) implementation with little or no user-interaction. The disadvantage is that the approach is of little use for already existing software. The model-extraction approach precisely alleviates this by extracting a model from an existing implementation automatically, ensuring there is correspondence between the model and implementation. The main disadvantage is that we need an implementation of a, perhaps faulty, algorithm before analysis can start.

We would like to provide a translation supplying as many of the strengths of these approaches as possible. Here we aim at supporting model-extraction in a way that allows subsequent code generation. In [10] we introduce the sub-class of CPNs called *process-partitioned coloured Petri nets* (PP-CPNs), which allows us to generate executable code from a PP-CPN model, thus supporting the model-drive software engineering approach. The focus of this paper is to support the model-extraction approach. We aim at doing so in a way that the extracted model later can be used for code generation, i.e., we make sure that the extracted models are PP-CPNs. This has the advantage that not only do we support both approaches in Fig. 1, we also provide foundations for a third merged approach, shown in Fig. 2. Here, we can do round-trip engineering, where we take an implementation as input, verify and correct it on the level of a model, and, instead of manually updating the implementation, use the automatically generated one as input for the next iteration. We can of course also do the modification directly in the code based on the counter-example if desired. When we find no more errors, we directly transfer the model to Ok for program extraction. This is only necessary if we wish to extract a program in a language different from the input language. The model never terminates, reflecting that the development and error checking is an ongoing process.

The use of (a slightly restricted class of) CPNs allows us to refine data-structures as much as required and even using actual data-structures of the original algorithm or program. We assume that we additionally have or can derive an abstraction of all data-types used. Derivation of abstractions of the data-types used can be done by the user or automatically using counter-example guided abstraction refinement (CEGAR) [16] as implemented in SLAM [14] and BLAST [15]. CEGAR automatically improves abstractions by replaying errors found in an abstract model on the original program and using information about



**Fig. 2.** Approach combining model-driven software engineering and model-extraction

why a given error-trace cannot be replayed in the original program to refine the abstraction. Here we are not concerned with abstraction refinement, and assume the user takes care of this. In this paper we focus on model-extraction, and present an implementation able to automatically generate a PP-CPN model from a program in a simple but expressive language.

The rest of this paper is structured as follows: in the next section, we introduce process-partitioned coloured Petri nets as defined in [10] and a simple algorithm for state-space generation which we use as a running example to illustrate our idea. In Sect. 3, we introduce our approach to generating PP-CPN models from algorithms using a naive parallel version of the algorithm presented in Sect. 2. In Sect. 4, we identify a problem in the original parallelization, fix the problem and show that the problem is no longer present in a modified version. Finally, in Sect. 5, we sum up our conclusions and provide directions for future work. Readers are assumed to have some familiarity with general coloured Petri nets but not PP-CPNs. An earlier version of this paper has been published as [17]. The earlier version did not have an implementation of the translation, so the description of the translation is much more detailed in this version. We also provide more details on abstraction refinement.

## 2 Background

In this section, we briefly introduce CPNs and process-partitioned CPNs as defined in [10]. We also give a simple algorithm for explicit state-space generation which we use as an example in the remainder of the paper.

**Coloured Petri Nets.** The definition of process-partitioned CPNs use the definitions and notation of CPNs from [5, Chap. 4] as a basis. CPNs consist of *places*, *transitions*, and *arcs*. Places are typed and arcs have expressions that may contain variables. The model-driven software engineering approach shown in Fig. 1 is a CPN, and here places are drawn as ovals, transitions as rectangles, and they are connected with arcs. Places and transitions can have names; in Fig 1 names are drawn inside the figure representing places/transitions, e.g., we have a place named Ok (the remaining places do not have names) and a transition

named *Verification*. Places have *types*, corresponding to types of variables in normal programming languages; the place types are typically written below or below and to the right of places; in Fig. 11 the place *Ok* has type *MODEL*. Places additionally can have a *marking*, a multi-set of *tokens* (values) residing on the place. We write the marking of a place in a rectangle close to the place and the total number of tokens in a circle on the place; in Fig. 11 the unnamed place to the top left has a marking of 1'initial\_model, i.e., a single token with value *initial\_model*. All other places contain no tokens, which is per convention not shown. The marking of a place before executing transitions is the *initial marking*; the model in Fig. 11 is shown in its initial marking.

Places and transitions are connected by arcs having *arc expressions*, e.g., if *ok* then 1'm else empty on the arc from *Verification* to *Ok*. Arc expressions are expressions that may contain free typed *variables*. We use the terms *input arc* and *output arc* to refer to all arcs having a given place/transition as source and destination, respectively. We extend this to also include *input place* and *output place* to denote all source places of input arcs of a transition and target places of output arcs.

A transition and an assignment of values to all free variables on arcs surrounding the transition is called a *binding element* or just a *binding*. We write a binding by writing the name of the transition and a list of assignments to all variables in brackets, like *Verification*(m=initial\_model,ok=true,c=[]). A binding is enabled in a marking if evaluating the expressions on all input arcs result in a multi-set of tokens that is a subset of the tokens residing on the corresponding input place in the marking. In the example in Fig. 11, the binding element *Verification*(m=initial\_model,ok=true,c=[]) is enabled as the unnamed place at the upper left does contain a token with the value *initial\_model*. The binding element *Model Refinement*(m=initial\_model,c=[]) is not enabled, as the counter example place contains no tokens (and thus in particular no token with value []). A binding can be *executed*, by removing all tokens from input places according to evaluations of the corresponding arc expressions and adding new tokens to all output places according to arc expressions on output arcs. We say that a transition is enabled if there are any enabled bindings of the transition. In Fig. 11 only *Verification* is enabled, and enabling of transitions is indicated by a bold outline. Transitions can additionally have *guards*, an extra expression written in square brackets next to it. These further limit the enabling as they have to evaluate to true for the transition to be enabled. No transition in Fig. 11 has a guard, but the transition *assign* in Fig. 6 (i) has guard [*id1' = id2*].

**Hierarchical Coloured Petri Nets.** CPNs have a module concept, where *subpages* are represented by *substitution transitions*, inducing a hierarchy of the pages. Graphically, we draw substitutions as transitions using a double outline. The model in Fig. 6 (c) has two substitution transitions, *Then* and *Else*. Parameters to subpages are specified as *port places* that have a direction (*In*, *Out*, or *I/O*). The model fragment in Fig. 6 (c) has three parameters, *S* of type *In*, and *E* and *R* of type *Out*. Port places are assigned to *socket places* on the page of the

substitution transition. These assignments are not shown explicitly graphically, but are often obvious from context (port and corresponding socket places have the same name or there is only one port place with the correct type). For example, we would use the fragment in Fig. 6 (c) as a subpage of the S1 substitution transition of the fragment in Fig. 6 (b). The port place S of fragment (c) would be assigned to the socket place S of (b), E of (c) to the unnamed place of (b), and R to R. The semantics is defined as replacing the substitution transition by the contents of the subpage, merging places in a port/socket relationship. We call this procedure *flattening*. This can be done automatically and is reversible, so in the remainder of this paper, we allow introducing and removing hierarchy whenever convenient, using it only to aid in presentation.

**Process-Partitioned Coloured Petri Nets.** In [10] we introduce the notion of *process-partitioned CPNs* (PP-CPNs). All models in the following sections of this paper are PP-CPN models. While this is important for the generation of code from extracted models, it is not important for the actual extraction, so we shall not go into too much detail about the nature of PP-CPNs, but only give a general idea of this subclass. Interested readers are invited to refer to [10] for a full formal definition of PP-CPNs.

PP-CPNs are CPNs, which are partitioned into separate kinds of processes. In this paper, we are only interested in models containing a single kind of process, so we just look at *process subnets* (Def. 2 in [10]). A single process subnet is a PP-CPN, but not necessarily the other way around. In this paper, whenever we talk about PP-CPNs, we assume they consist of exactly one process subnet. A process subnet is a CPN with a distinguished *process colour set* serving as a process identifier. The models in Fig. 7 are examples of PP-CPNs (we provide a detailed description of the models in Sect. 3). The process colour set of this model is PROCESS. The places of a process subnet are partitioned into *process places*, *local places*, and *shared places* (in [10], we additionally introduce *buffer places* for asynchronous communication between processes, but these are not used here). These places correspond to the control flow, local variables, and shared variables of programs. Process places must be typed by the process colour set (in the example, entry and exit and all places  $s_i$  are process places), local places must be typed by a product of the process colour set and any other type (in the example, s, ss, and the condition places), and shared places can have any type (in the example, waiting and visited).

In the initial marking, exactly one of the process places contains all tokens of the process colour set and all remaining process places are empty (modeling that all processes start in the same location in the program). In Fig. 7 the entry place contains all (two) processes and none of the other process places contain any tokens. Local places initially contain exactly one token for each process so that if we project onto the component of the process colour set, we obtain exactly one copy of all values of the set (modeling that all local variables must be initialized). This is seen on s, ss, and the condition places of Fig. 7.

All shared places contain exactly one value (modeling that shared variables must be initialized). This is seen on `visited` and `waiting` in Fig. 7. All arc expressions must ensure that tokens are preserved.

We have chosen to adopt the notion from [10] that we cannot create new processes or destroy processes even though nothing in our approach breaks if we allow dynamic instantiation and destruction of processes. This is mainly for simplicity as we did not need dynamic instantiation in our examples. We allow a slightly relaxed syntax for PP-CPNs here compared to [10], as we allow using constants on input arcs, relying on pattern matching to determine enabling of transitions depending on input data instead of guards. This is done for legibility of patterns for conditionals and can easily be undone by instead using a variable and checking for the correct value in the guard.

**State-space Generation.** State-space generation is a means of analysis of formal models, such as the ones specified by means of CPNs. A simple implementation is shown in Fig. 3. We start in the initial marking of the model and compute all enabled bindings. We then systematically execute each, note the markings we reach by executing bindings, store them in `WAITING`, and repeat the procedure for each of these newly discovered markings. To also terminate in case of loops, we store all markings for which we have already computed successors in `VISITED` and avoid expanding them again. We often call a marking a *state* in the context of state-space analysis.

```

1: WAITING := MODEL.initial()
2: VISITED := MODEL.initial()
3: while not WAITING.isEmpty() do
4:   s := WAITING.head()
5:   WAITING := WAITING.remove(s)
6:   // Do any handling of s here
7:   for all b in s.enabled() do
8:     ss := s.execute(b)
9:     if not VISITED.contains(ss) then
10:       WAITING := WAITING.add(ss)
11:       VISITED := VISITED.add(ss)
12:     endif
13:   endfor
14: endwhile

```

Fig. 3. State-space exploration algorithm

### 3 Approach

We introduce our approach to verifying parallel algorithms by a parallel version of the algorithm for generating state-spaces shown in Fig. 3. The basic idea is to use the loop of Fig. 3 for each process and share the use of `WAITING` and `VISITED`, naturally with appropriate locking. From this algorithm, we illustrate our approach to extract a PP-CPN model. The approach is completely general, though.

A simple way to parallelize Fig. 3 is shown in Fig. 4. The comments in lines 1, 6, and 23 are reintroduced in a subsequent refinement. In this first version, we initialize as before (ll. 2–3). We have moved the main loop to a separate procedure, *computeStateSpace*. We perform mostly the same loop as before (ll. 8–17), but instead of testing for emptiness and picking an element of the queue

in three steps, we do so using a procedure *pickAndRemoveElement* (ll. 7 and 16). The implementation of *pickAndRemoveElement* (ll. 22–32) does the same as we did before, except we return a bottom element *notFound* if no elements are available, and use that in the condition of the loop (l. 8). This forces us to perform the pick in two places: before the first invocation of the loop (l. 7) and at the end of the loop (l. 16). Handling of states (l. 9) and iteration over all enabled bindings (ll. 10–15) is the same as before. Now, instead of checking if a state is a member of VISITED and conditionally adding it to the set, we do both in a single step as shown in the procedure *addCheckExists* (ll. 12 and 34–40). We do this under the assumption that adding an element to the set does nothing if the element is already there. If the state was not already in VISITED, we add it to WAITING (l. 13). The reason for this re-organization is that we now assume that *pickAndRemoveElement*, *addCheckExists*, and the access to WAITING in line 25 are atomic. We ensure this by acquiring locks in *pickAndRemoveElement* and *addCheckExists* (ll. 24 and 35). This allows us to start two instances of *computeStateSpace* in parallel in line 22. We will not argue for the correctness of either Fig. 3 or Fig. 4, but note that it is easy to convince ourselves that if one is correct so is the other, with the assumption that *pickAndRemoveElement* and *addCheckExists* happen atomically.

```

1: // bool WAITING
2: WAITING := MODEL.initial()
3: VISITED := MODEL.initial()
4:
5: proc computeStateSpace() is
6:   // bool s
7:   s := pickAndRemoveElement()
8:   while not s = notFound do
9:     // Handle s here
10:    for all b in s.enabled() do
11:      ss := s.execute(b)
12:      if addCheckExists(ss) then
13:        WAITING := WAITING.add(ss)
14:      endif
15:    endfor
16:    s := pickAndRemoveElement()
17:  endwhile
18: endproc
19:
20: computeStateSpace() ||
    computeStateSpace()

22: proc pickAndRemoveElement() is
23:   // bool s
24:   lock pick
25:   if WAITING.isEmpty() then
26:     return notFound
27:   endif
28:   s := WAITING.head()
29:   WAITING := WAITING.remove(s)
30:   return s
31: unlock
32: endproc
33:
34: proc addCheckExists(s) is
35:   lock add
36:   contains := VISITED.contains(s)
37:   VISITED := VISITED.add(s)
38:   return not contains
39: unlock
40: endproc

```

**Fig. 4.** Naive parallel state-space algorithm

### 3.1 Model Extraction

To go from Fig. 4 to a PP-CPN model, we first extract the control-flow of the algorithm including generating representations of data, and then we refine the update of data until we can prove the desired properties of the model.

The main idea of our translation is shown in Fig. 5. We start with a program, and perform some simplifications of it to ease translation. We then translate the simplified program to a CPN model using templates for each construct. Finally, we simplify the resulting model to ease analysis.

We assume we have a parsed program with the syntax seen below. All algorithms presented in this paper adhere to the syntax. We use this syntax instead of a stock language for two reasons: To keep the syntax independent of a concrete language and for ease of parsing for simple prototyping. While this may seem off as our goal is to verify existing programs, it suffices for a prototype implementation, and our translation works on abstract syntax only, so adapting to a concrete syntax is only as much work as writing the parser. We allow unparsed content in our programs; this should be further defined to improve abstraction refinement, but this is not our primary focus here, so we have decided to allow this for simplicity.

Our syntax is simple but expressive, and contains most of the elements one would expect in a programming language. A program is a list of procedure definitions, parallel procedure invocations, and statements (which are assumed to be atomic except for procedure invocation). A procedure definition has a list of parameters (which may or may not have a specified type) and comprises a list of statements. Types can either be specific types or any type identifier. Statements are either definitions of variables, assignments of expressions to variables, if statements, while loops, for all loops, repeat loops, sections protected by a lock, returns from a procedure, or an expression. Expressions can be parenthesized or negated, and are either function calls, identifiers, or anything (unparsed content). Expressions can always be followed by a list of method invocations.

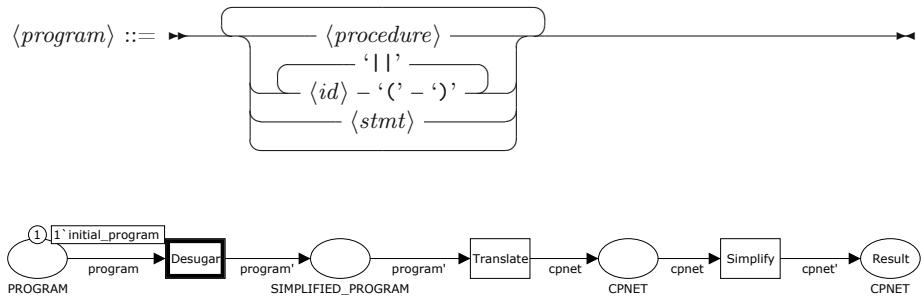
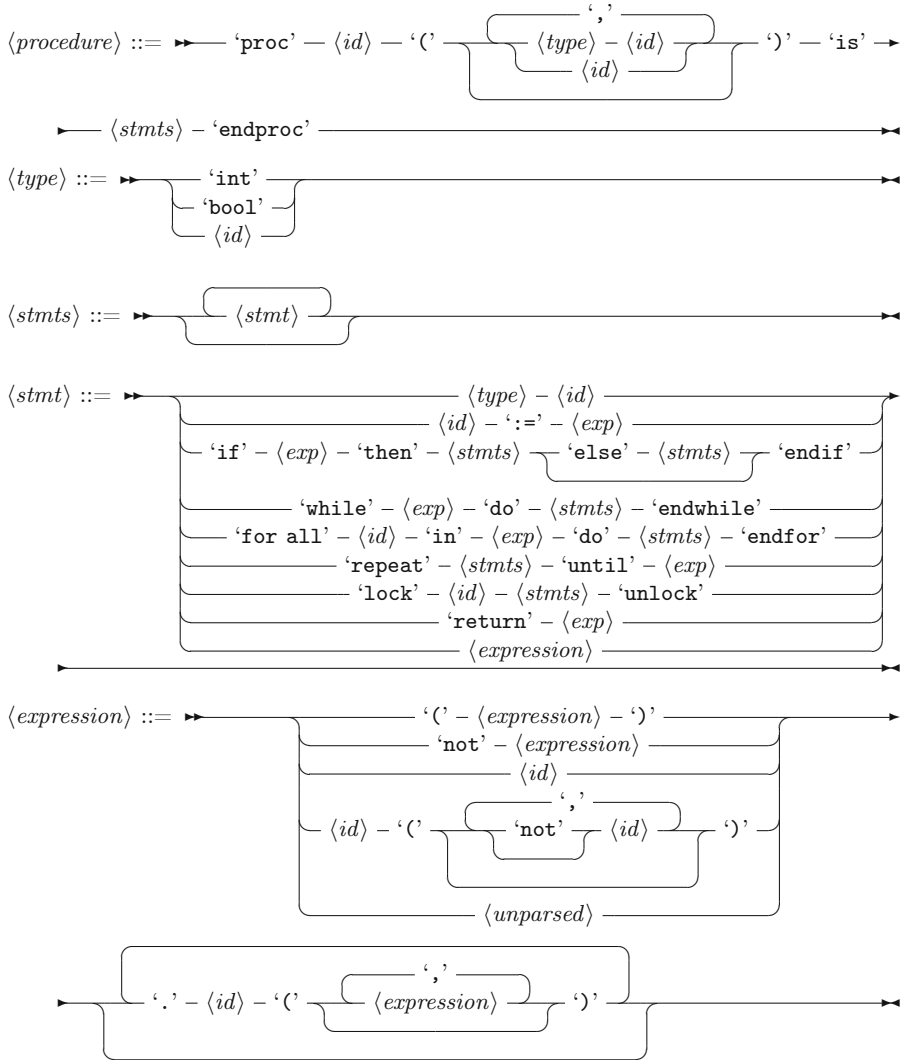


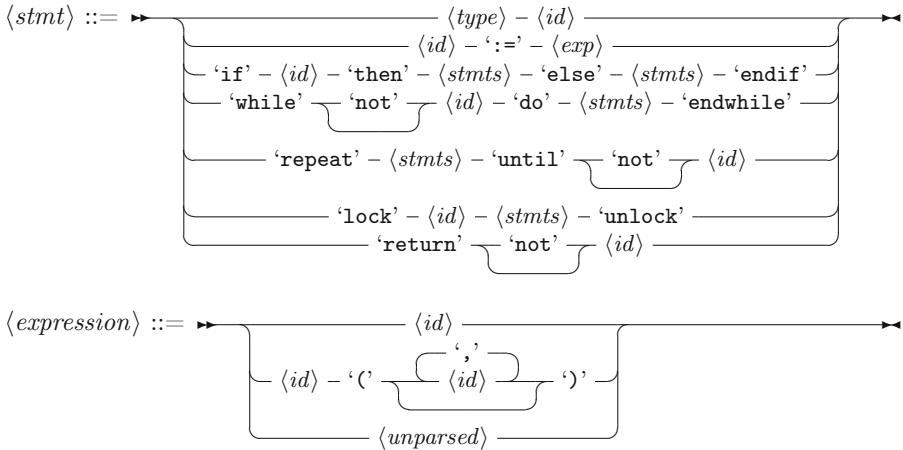
Fig. 5. Translation approach





While our grammar is useful for humans, it is not suitable for translation to CPN models. Instead, we rewrite our programs according to syntactical rules, to simplify the programs. We replace for all loops with while loops, i.e., ‘for all’  $\langle \text{id} \rangle$  ‘in’  $\langle \text{expression} \rangle$  ‘do’  $\langle \text{stmts} \rangle$  ‘endfor’  $\equiv$  ‘all\_’  $\langle \text{id} \rangle$  ‘:=’  $\langle \text{expression} \rangle$  ‘while’ ‘all\_’  $\langle \text{id} \rangle$  ‘.’ ‘hasMore’ ‘( ‘)’ ‘do’  $\langle \text{id} \rangle$  ‘:=’ ‘all\_’  $\langle \text{id} \rangle$  ‘.’ ‘getFirst’ ‘( ‘)’  $\langle \text{stmts} \rangle$  ‘endwhile’. Then, we remove all method invocations (as they are not supported by Standard ML, the inscription language of CPN Tools). We do this by replacing method invocation by simple invocation, i.e.,  $\langle \text{exp} \rangle$  ‘.’  $\langle \text{id} \rangle$  ‘( ‘  $\langle \text{exp}_1 \rangle$  ‘,’ ... ‘,’  $\langle \text{exp}_n \rangle$  ‘)’  $\equiv$   $\langle \text{id} \rangle$  ‘( ‘  $\langle \text{exp} \rangle$  ‘,’  $\langle \text{exp}_1 \rangle$  ‘,’ ... ‘,’  $\langle \text{exp}_n \rangle$  ‘)’’. We also force the use of the else path in if statements, introducing one with

an empty statement list if necessary. Finally, we simplify statements, so only variables or negated variables are used in expressions other than assignments. This is done using several rules, introducing temporary variables as needed. One such rule is ‘while’  $\langle exp \rangle$  ‘do’  $\langle stmts \rangle$  ‘endwhile’  $\equiv$  ‘condition’ ‘:=’  $\langle exp \rangle$  ‘while’ ‘condition’ ‘do’  $\langle stmts \rangle$  ‘condition’ ‘:=’  $\langle exp \rangle$  ‘endwhile’. We could also translate repeat loops to while loops, but have chosen not to as this yields simpler models. We allow negations in most conditions (but not in if statements). Thus, we get programs adhering to a simplified grammar replacing the  $\langle stmt \rangle$  and  $\langle exp \rangle$  productions with:



Extracting the control-flow of a procedure consists of creating the process places and transitions of the model. We do that using templates, very similar to the workflow-patterns [18] for low-level Petri nets. In Fig. 6 we show the patterns necessary to translate programs using our simple pseudo-code language to a PP-CPN model. The first patterns (a-f) define the basic control flow and are atomic actions, sequence of statements, conditional, while loop, repeat loop, and critical section, corresponding to productions 3–6 of  $\langle stmt \rangle$  and  $\langle stmts \rangle$ . The type P is the process colour set and for each pattern, the place S is the start place, E the end place, and R a special place pointing to the end of the procedure. All places created are process places except for the Mutex place, which is a shared place, and the condition places, which are local places. The intuition of each fragment is that we start a block in S and execute towards E. At any point, we may also do early termination, going to R. Fragments (a-e) should be self-explanatory; in fragment (f) we make sure to release the mutex no matter how we leave the critical section. Also, the Mutex place is a shared place, which means it is shared among all instances of the fragment. The top level of each procedure is shown as fragment (g), where we see we get all input parameters (if any),  $p\_id1 \dots p\_idk$ , all global variables ( $g\_id1 \dots g\_idm$ , and all local variables ( $l\_id1 \dots l\_idn$ ). On normal termination we move a token from E to R to ensure a single exit point,

and store the return value on Return Value. Explicit returns are handled by fragment (h), which diverts control to the R place and updates the Return Value. All global and local variables, parameters, and the return value place is also passed on to each fragment, but this is not shown here for simplicity.

In the initial abstraction, we approximate the type of all variables with UNIT, and local and shared places are not connected to transitions for reading. All conditions are of type BOOL and assigned values non-deterministically. For each variable, we introduce a place with the same name as the variable and two CPN variables, one with the same name as the original variable and a version with a

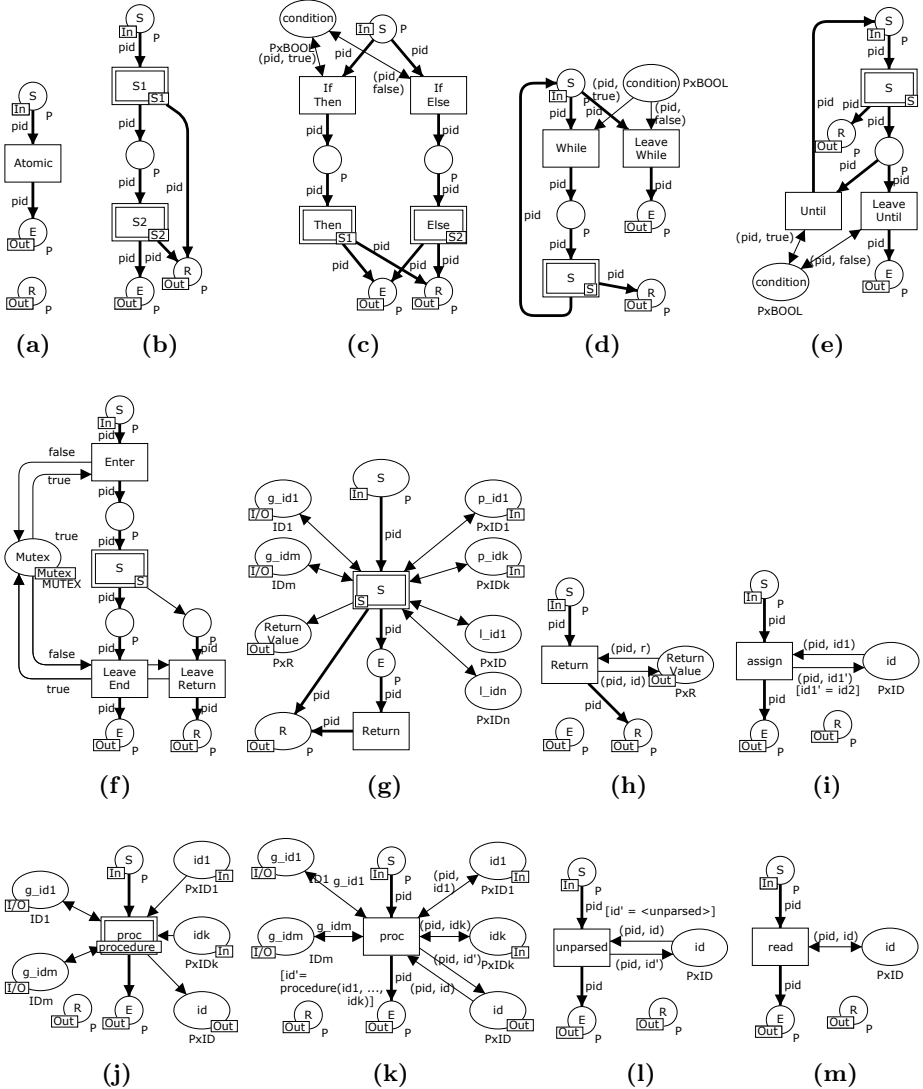


Fig. 6. Patterns for control structures

prime. The first is used for reading old values and the primed version is used for updating with new values. The following fragments are all described as if they are using local variables; they can all use global variables, in which case the process id component is removed. Updating values is handled by fragment (i). Procedure invocation is handled by fragments (j) and (k). (j) handles the case where a procedure is defined in the program and (k) when it is not (i.e., constitute a library function or a function whose action we abstract away). Invocation in both cases assigns values ( $id1 \dots idk$ ) for each of the parameters of the procedure, and binds the return value to the correct return value ( $id$ ). We note that procedure fragments only have one exit for processes, so we only connect the E place to the invocation, leaving the R place unconnected (so a `return` only returns from the inner-most procedure). Pattern (l) and (m) are used for abstraction refinement and are introduced in the next subsection.

Applying this extraction to the *computeStateSpace* procedure of Fig. 4, we obtain Fig. 7 (left). The model has been manually laid out, we have simplified the generated names for legibility, and we have hidden the process id flowing along the bold arcs. The details are not important and the model is provided only to show the unreduced output of the translation. We translate procedures without use of hierarchy in our implementation, so all places and transitions pertaining to a single procedure are contained in a single page (but we have a page for each procedure). We can see the two loops, one from the branch-off around the middle of the long chain and back to the beginning and one from the bottom to the middle. We can see 3 branches, corresponding to the two loops and the if statement. We note that due to the program simplification, we get significantly more transitions than we have lines in the procedure. All transitions have a name from the line number used to generate it and a sequence number to ensure all transitions have unique names. The model in Fig. 7 (left) is good for getting a basic understanding of the control flow of the system, but impractical for analysis and detailed understanding as a lot of steps do not contribute to the behavior, especially the long lines of transitions not accessing any data and that unconditionally move from one state to the next. This also causes state-space explosion without providing benefits for analysis. It is therefore desirable to remove such transitions. We additionally merge lock releases with the last transitions of each procedure, also omitting an unconditional step (not applicable in this model). Doing this, we obtain the model in Fig. 7 (right). We can now see the 3 procedure calls as substitution transitions and global, local, and 4 generated variables as the marked places.

**Abstraction Refinement.** The initial abstraction allows execution of traces not allowed in the original program. In the model in Fig. 7 (right), it is possible to first terminate process 1 and have process 2 continue computation, i.e., execute:

```
// Process 1 enters and leaves pickAndRemoveElement in l7_10
l8_11⟨pid=computeStateSpace(1)⟩ →
```



```

l8_12⟨pid=computeStateSpace(1),c=false⟩ →
l8_14⟨pid=computeStateSpace(1)⟩ → // Terminate process 1
// Process 2 enters and leaves pickAndRemoveElement in l7_10
l8_11⟨pid=computeStateSpace(2)⟩ →
l8_12⟨pid=computeStateSpace(2),c=true⟩ →
l8_14⟨pid=computeStateSpace(2)⟩ → // Process 2 continues

```

This is not possible in Fig. 4. The model does find all possible interleavings of the process, though, so if the state-space does not contain any erroneous states, neither will the program.

Abstraction refinement consists of using more elaborate types on local and shared places and of applying pattern (m) of Fig. 6 to bind identifiers instead of non-determinism. Refinements must limit the behavior of previous models (which formally must be able to simulate any refinement if we ignore local and shared places), and must be true to the original program. If we add an explicit type to a variable in our program, our tool automatically applies pattern (m) for variables in all expressions. Furthermore, our tool will change the type of the place corresponding to the declared type accordingly. We need to parse (parts of) expressions in order to find all places a variable is read. If *⟨unparsed⟩* content is used, it is inserted verbatim in the model without any processing using pattern (l) of Fig. 6. This is done for simplicity of implementation of our prototype and means that an expression using the *⟨unparsed⟩* production reads a variable, this is not detected.

Here we do a simple refinement to avoid the situation where one process can decide that *Waiting* is empty just to have the other immediately afterward decide it is not. For this, we refine the type of *Waiting* to a *BOOL*, indicating whether the *Waiting* set is empty or not, and refine the type of *s* to indicate whether *notFound* was returned from *isEmpty*. Refining *s* and *Waiting* we obtain the model in Fig. 8. At the left, we see the refined model for *computeStateSpace* with changes highlighted. Due to the size of the model, we have just zoomed in on the changed parts. We have manually added place-/transition-names corresponding to Fig. 7 (right) to process places and transitions. We have more transitions in the refined model as they no longer can be reduced away due to refinement. To the right, we see the generated code for *pickAndRemoveElement* (top) and *pickAndRemoveElement* (bottom). All global variables are shared among all pages, local variables on subpages correspond to the actual parameters on *computeStateSpace*, the return value is mapped as well, and so are the entry-/exit-points. We have two instances of *pickAndRemoveElement* with different entry-/exit-point mappings. All ports are of type *I/O* due to a technicality in our translation, but as the port type carries no semantics, this is no problem. We get the refinement by explicitly declaring the types of *s* and *Waiting* in lines 1, 6, and 23 in Fig. 4.

Refining a variable means the place corresponding to a variable is accessed and updated according to the program. The value of *waiting* is initially *false* (as we add the initial state in line 2 of Fig. 4) and we do not care about the initial value



for the transition `l25_5` setting `c'` based on the value of `isEmpty(waiting)`. We implement the functions `isEmpty`, `head` (used in `l28_9`), and `add` (used in `l13_31`). `isEmpty` is the identity function (as the abstract value of `waiting` models whether the data-structure is empty). `head` returns the negation of its parameter (if the set is empty we cannot return a value, otherwise we can), and `add` always returns `false` (as adding an element always makes the data-structure non-empty). We have removed generated guards for `l29_10` and `l10_39` to make the result non-deterministic. We have explicitly added a reference to `s` in `computeStateSpace` transitions `l8_14` and `l8_36` as it is used as part of an unparsed expression (`s = notFound`). We can no longer execute the erroneous trace on this refined model.

## 4 Analysis

The main reason we started this work in the first place was to analyze parallel algorithms. Our focus is on new problems arising when creating parallel algorithms, not on proving correctness of serial algorithms. We therefore assume that algorithms are correct under certain mutual-exclusion assumptions, and search for such violations. We are also interested in potential dead- and live-locks. Assuming a valid refinement, we ensure absence of safety violations in the model guarantees absence in the real program as we can simulate all executions of the algorithm. This includes proving absence of mutual-exclusion violations. We cannot use our approach to ensure the absence of dead-locks, as we deal with over-approximations of the possible interleavings and further restricting the behavior may introduce new dead-locks, but we can still find dead-locks and remove them from the implementation.

We can do state-space analysis of the derived models from Figs. 7 and Fig. 8, and obtain state-spaces with 47,141 states for the original, unreduced model (Fig. 7 (left)), 18,139 states for the reduced model (Fig. 7 (right)), showing that reduction is important, and 15,934 states for the refined model (Fig. 8). We easily prove that the mutex property is not violated for the two critical regions.

**Dead-locks and Live-locks.** As all processes have a distinguished start and end-state, we can recognize dead-locks and live-locks in the model. A *dead-lock* is a state without successors (a *dead state*) where not all processes reside on E. We can find dead-locks in CPN Tools using the query:

```

1 fun terminal node =
2   Mark.computeStateSpace 'E 1 node = P.all ()
4 List.filter (fn n => not (terminal n)) (ListDeadMarkings ())

```

The function `terminal` tests whether the Marking of E on the page `computeStateSpace` (the top level) contains all tokens of P and line 4 removes all terminal states from the dead states (markings in CPN Tools terminology) and only returns non-terminal states. None of the models in Figs. 7 and Fig. 8 have dead-locks; the models in Fig. 7 have 64 dead states, where all process ids reside on E and the conditions have various values. The model in Fig. 8 has 48 dead states, where



all process ids reside on E, `Waiting` is true, `s` is `notFound` for all processes, and conditions have various values, except for the ones controlling termination.

Live-locks are harder to recognize. We only consider live-locks in the absence of dead-locks. A model has a *strong live-lock* if the dead states of the model do not constitute a *home space*, i.e., if it is not always possible to reach one of the dead states. A strong live-lock in the model does not necessarily imply a live-lock in the original algorithm, but can be used to identify parts of the original program that should be further investigated. None of the models in Fig. 7 have strong live-locks. We can test this in CPN Tools using the query:

```
1 HomeSpace (ListDeadMarkings ())
```

A model may have a *weak live-lock* if its state-space has a loop. A loop may also just indicate that a loop may execute an unbounded number of times. Both models in Fig. 7 and the refined one in Fig. 8 have loops. This is caused by the loops in the algorithm and are perfectly acceptable. A particular interesting kind of live-lock is a loop reachable from a state where E contains tokens. This means that even after one of the processes has terminated, the amount of work done by another process is unbounded. We have already seen that Fig. 7 exhibits this due to too abstract modeling, i.e., that process 1 may decide that `Waiting` is empty initially and terminate, just to have 2 decide it is non-empty and continue computation. We have seen this is not possible in the original algorithm, which caused us to refine the model to Fig. 8. We would therefore expect that no such live-lock was present in the refined model. Maybe surprisingly, one such does exist. We can find this by searching for all nodes where E contains tokens which are reachable from themselves via a non-trivial loop. In CPN Tools, we can find these using the query:

```
1 fun selfReachable node =
2 let
3   val nodes = OutNodes node
4   fun test [] = node
5     | test (n::rest) =
6       if n = node
7       then test rest
8       else if (SecReachable (n, node))
9             then n
10            else test rest
11 in
12   test nodes
13 end
14
15 fun predicate node =
16   if Mark.computeStateSpace 'E 1 node = empty
17   then false
18   else selfReachable node <> node
19
20 PredAllNodes predicate
```

The function `selfReachable` computes all successors of the given node and for each of these, unless it is the node itself, test whether the given node is reachable from the successor. If the node is, the successor is returned, otherwise the original node is. The `predicate` tests if E at the top level is empty. If not, it tests if the node is non-trivially reachable from itself. Finally, we apply this predicate to the entire state space. We can use early termination by replacing line 20 by:

```
20 PredNodes (EntireGraph, predicate, 1)
```

We can replace the 1 by any number of examples we want. We can get a witness using:

```
1 ArcsInPath(1, List.hd (PredNodes (EntireGraph, predicate, 1)))
```

We can get the shortest path to such a node using:

```
1 fun shortest (node::nodes) =
2 let
3   fun test path [] = path
4     | test path (n::nn) =
5       let
6         val path' = ArcsInPath(1, n)
7       in
8         if List.length path > List.length path'
9         then test path' nn
10        else test path nn
11      end
12 in
13   test (ArcsInPath (1, node)) nodes
14 end
15
16 shortest (PredAllNodes predicate)
```

Here, the function `shortest` uses a helper function, `test`, which computes a shortest path to a node and compares with the current globally shortest path and returns the shortest one of the two. We can get a list of binding elements for analysis using:

```
1 List.map ArcToBE (shortest (PredAllNodes predicate))
```

For our example, we get that such a path is achieved by having one process enter *pickAndRemoveElement* and end up setting `Waiting` to true at l29\_10 (which is the first time a choice occurs). Then, the process leaves *pickAndRemoveElement* and the other process starts. The just started process enters *pickAndRemoveElement*, discovers that `Waiting` is empty, leaves *pickAndRemoveElement* and terminates. Now, the first process continues, sets `Waiting` to false in l13\_31, only stopping when it has performed all the work on its own.

This is also possible in the algorithm in Fig. 4, and even quite likely as the two processes will test `Waiting` initially, one of them will consume the only element it contains initially, and other processes terminate. This also occurred in reality in our first implementation.

To fix this, we notice that the reason one process terminates prematurely in the previous example is that it decided to terminate while the other can still add new states to `Waiting`. The idea of an improved algorithm is to ensure that no processes may terminate when others may produce new states. This prompts us to make the improvement seen in Fig. 9. We reuse *addCheckExists* and *pickAndRemoveElement* from Fig. 4, and define a new procedure for picking, *pickWithCounter* (ll. 27–36) which is used in place of the original *pickAndRemoveElement* (ll. 43 and 52). We use `MAYADD` as a counter of the number of processes which may add new states to `WAITING`. We add an additional loop around the previous main loop ensuring we only quit when `MAYADD` is zero.

We use the same approach to translate the program to a CPN model, using the same refinements for `s` and `Waiting` and no abstraction of `MAYADD`. We

```

1: bool WAITING
2: WAITING := MODEL.initial()
3: VISITED := MODEL.initial()
4: int MAYADD
5: MAYADD := 0
6: ...
27: proc pickWithCounter() is
28:   bool s
29:   lock withCounter
30:     s := pickAndRemoveElement()
31:   if not s = notFound then
32:     MAYADD := MAYADD + 1
33:   endif
34:   return s
35: unlock
36: endproc
37:
38: computeStateSpace() ||
    computeStateSpace()
40: proc computeStateSpace() is
41:   bool s
42:   repeat
43:     s := pickWithCounter()
44:   while not s = notFound do
45:     // Do any handling of s here
46:     for all b in s.enabled() do
47:       ss := s.execute(b)
48:       if addCheckExists(ss) then
49:         WAITING := WAITING.add(ss)
50:       endif
51:     endfor
52:     s := pickWithCounter()
53:     MAYADD := MAYADD - 1
54:   endwhile
55:   until MAYADD=0
56: endproc

```

**Fig. 9.** More involved state-space algorithm

obtain a model with a state-space with 143,372 nodes, 56 dead states, no dead-locks, and no live-locks. We have no weak live-locks reachable from a state where E contains tokens. Due to the translation being done automatically instead of manually, we actually found an error in the algorithm described in [17]. That algorithm is the same as the one in Fig. 9 except the one in [17] had lines 52 and 53 swapped, which allows one process to terminate while the other is between lines 52 and 53. In [17] we did not catch that due to the manual construction of the model, but with our automatic translation we did.

Verification of the two process case has given us confidence that the algorithm will work with any number of processes. We have also investigated an extended version additionally adding a check-point mechanism where all threads are paused while WAITING and VISITED are written to disk in a consistent configuration.

We also used the method to verify the implementation of a slightly simplified version of the protocol for operational support developed in [6,19]. The protocol supports a client which sends a request to an operational support service, which mediates contact to a number of operational support providers. The protocol developed in [6,19] has support for running all participants on separate machines, but we are satisfied with an implementation running the operational support server and providers on the same machine. We therefore have to send fewer messages, but need to access shared data on the server. We devised a fine-grained locking mechanism and used the method devised in this paper to prove that it enforces mutual exclusion and causes no dead-locks, increasing our confidence that the implementation works.

## 5 Conclusion and Future Work

We have given an approach for correct implementation of parallel algorithms. The approach allows users to extract a model from an algorithm written in an implementation or abstract language and verify correctness using state-space analysis. The approach also facilitates the generation of skeleton implementation code from the verified model using the approach from [10] as we rely on process-partitioned coloured Petri nets. Finally, we can also combine the two approaches, which facilitates writing an algorithm in an abstract language, extract a model for verification, and then extract a skeleton implementation. We have implemented a translation from a simple language to CPN models. The power of having an implementation is seen by the fact that we caught an error in the algorithm published earlier in [17] which erroneously had two lines swapped.

Verification of software by means of models is not new. Code-generation from models have been used in numerous projects. The approach has been most successful for generating specification of hardware from low-level Petri nets and other formalisms to synthesize hardware such as computer chips [7,8]. The approach has also been applied to high-level Petri nets to generate lower level controllers [9] and more general software [10]. Model extraction was pioneered by FeaVer [11], which made it possible to extract PROMELA models from C code using user-provided abstractions, and Java PathFinder [13] which did the same for Java programs. The approach has successfully been refined using counter-example guided abstraction refinement (CEGAR) [16] which was first implemented by Microsoft SLAM [14], which extracts and automatically refines abstractions from C code for Microsoft Windows device drivers, and refined by BLAST [15]. While the tools for model-extraction support a full development cycle by abstraction refinement and reuse for modified implementations, the idea of combining the two approaches is to the best of our knowledge new. The combination allows some interesting perspectives. The perspective we have focused on in this paper is the ability to write an algorithm in pseudo-code, extract a model from the code, and generate an implementation in a real language. Another perspective is supporting a full cycle as well, where we extract a model from a program, find and fix an error in the model, and emit code that is merged with the original code, supporting a cycle where we do not need to fix problems in the original code but can do so at the model level. The use of coloured Petri nets instead of a low-level formalism allows us to use the real data-types used in the program instead of abstractions, much like how FeaVer allows using C code as part of PROMELA models, but with the added bonus that the operations are a true part of the modeling language rather than an extension that requires some trickery to handle correctly.

The work presented here is far from done. While our prototype allows us to verify simple but non-trivial examples, it cannot cope with more complicated systems. The main problem is that the state-space is growing very large, even just for the example seen in Fig. 9 with 2 processes (which has a state-space with 143,372 states). We can alleviate this by more reduction rules and more sophisticated translations. For example, we currently only consider transitions

on single pages when removing unconditional paths. Also, updates to local variables are done concurrently, which is sub-optimal. We can alleviate this by using partial order reduction [20,21] or by using transition priorities and giving such transitions high priority. Manually giving all transitions accessing only local data high priority reduces the state-space of the resulting model to 49,162 states.

We would also like to make the implementation more intelligent with respect to how types are handled. Currently, only directly declared types are taken into account. This means the prototype incorrectly generates an unrefined type on Return Value of *addCheckExists* in Fig. 8 even though we can see it is used as a boolean in line 12 of Fig. 4. It would be nice to use type unification to avoid this. It would also be interesting to make more advanced type analysis, propagating refinements, so we only have to refine *s* either in line 6 or in line 23 of Fig. 4. This would also allow us to not automatically refine the conditions of all conditionals, further simplifying generated models. We also want to improve how complicated expressions are parsed, i.e., reduce how often the *<unparsed>* production is used. This would remove some manual labor (like adding arcs to places for *l8\_14* and *l8\_36* in Fig. 8 due to line 8 of Fig. 4 containing unparsed content). More importantly, more intelligent parsing would allow us to do simple abstraction automatically, only asking users when unknown functions are called. In the longer term, it would be interesting to assist users with this using replay on the original program.

Our current method focuses on parallel algorithms with a fixed number of identical processes, but there is nothing in our approach preventing us from extending this to also handle distributed settings with asynchronous communication using buffer places and different kinds of processes; the code generation in [10] even supports that out of the box. While the fixed number of processes used in this paper works well for simple algorithms, more advanced algorithms may need to spawn processes. Nothing in our approach inherently forbids this, but the code generation in [10] does not support this out of the box. We believe that it should be quite easy to devise a construction for starting new processes and adapt the code generation to handle this.

Our current approach does not support recursive (or mutually recursive) procedures, as we create a static sub-page for each procedure call. This is done for simplicity in the prototype, but could easily be changed to explicit hand-over of control among several top-level pages. This is similar to dynamic process instantiation and interprocess communication and can be implemented by adding to each procedure setup and teardown transitions initializing and removing local variables, and maintaining a mapping between process identifiers.

## References

1. Billington, J., Wilbur-Ham, M., Bearman, M.: Automated protocol Verification. In: Proc. of IFIP WG 6.1 5th International Workshop on Protocol Specification, Testing, and Verification, pp. 59–70. Elsevier (1985)
2. Kristensen, L.M., Jørgensen, J.B., Jensen, K.: Application of Coloured Petri Nets in System Development. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 626–685. Springer, Heidelberg (2004)

3. Kristensen, L.M., Jensen, K.: Specification and Validation of an Edge Router Discovery Protocol for Mobile Ad Hoc Networks. In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.) INT 2004. LNCS, vol. 3147, pp. 248–269. Springer, Heidelberg (2004)
4. Espensen, K., Kjeldsen, M., Kristensen, L.: Modelling and Initial Validation of the DYMO Routing Protocol for Mobile Ad-Hoc Networks. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 152–170. Springer, Heidelberg (2008)
5. Jensen, K., Kristensen, L.: Coloured Petri Nets – Modelling and Validation of Concurrent Systems. Springer (2009)
6. Westergaard, M., Maggi, F.M.: Modeling and Verification of a Protocol for Operational Support Using Coloured Petri Nets. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 169–188. Springer, Heidelberg (2011)
7. Yakovlev, A., Gomes, L., Lavagno, L.: Hardware Design and Petri Nets. Kluwer Academic Publishers (2000)
8. IEEE Standard System C Language Reference Manual. IEEE-1666
9. Rasmussen, J.L., Singh, M.: Designing a Security System by Means of Coloured Petri Nets. In: Billington, J., Reisig, W. (eds.) ICATPN 1996. LNCS, vol. 1091, pp. 400–419. Springer, Heidelberg (1996)
10. Kristensen, L.M., Westergaard, M.: Automatic Structure-Based Code Generation from Coloured Petri Nets: A Proof of Concept. In: Kowalewski, S., Roveri, M. (eds.) FMICS 2010. LNCS, vol. 6371, pp. 215–230. Springer, Heidelberg (2010)
11. The FeaVer Feature Verification System webpage,  
<http://cm.bell-labs.com/cm/cs/what/feaver/>
12. Holzmann, G.: The SPIN Model Checker. Addison-Wesley (2003)
13. Havelund, K., Presburger, T.: Model Checking Java Programs Using Java PathFinder. STTT 2(4), 366–381 (2000)
14. Ball, T., Rajamani, S.: The SLAM project: debugging system software via static analysis. In: Proc. of POPL, pp. 1–3. ACM Press (2002)
15. Beyer, D., Henzinger, T., Jhala, R., Majumdar, R.: The Software Model Checker BLAST: Applications to Software Engineering. STTT 7(5), 505–525 (2007)
16. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. J. ACM 50, 752–794 (2003)
17. Westergaard, M.: Towards Verifying Parallel Algorithms and Programs using Coloured Petri Nets. In: Proc. of PNSE. CEUR Workshop Proceedings, vol. 723, pp. 57–71. CEUR-WS.org (2011)
18. van der Aalst, W., van Hee, K.: Workflow Management: Models, Methods, and Systems. MIT Press (2002)
19. Westergaard, M., Maggi, F.: Modelling and Verification of a Protocol for Operational Support using Coloured Petri Nets. Submitted to Fundamenta Informaticae
20. Peled, D.: All from One, One for All: On Model Checking Using Representatives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993)
21. Valmari, A.: Stubborn Sets for Reduced State Space Generation. In: Rozenberg, G. (ed.) APN 1990. LNCS, vol. 483, pp. 491–515. Springer, Heidelberg (1991)

# Report on the Model Checking Contest at Petri Nets 2011

Fabrice Kordon<sup>1</sup>, Alban Linard<sup>2</sup>, Didier Buchs<sup>2</sup>, Maximilien Colange<sup>1</sup>,  
Sami Evangelista<sup>3</sup>, Kai Lampka<sup>4</sup>, Niels Lohmann<sup>5</sup>, Emmanuel Paviot-Adet<sup>1</sup>,  
Yann Thierry-Mieg<sup>1</sup>, and Harro Wimmel<sup>5</sup>

<sup>1</sup> LIP6, CNRS UMR 7606, Université P. & M. Curie – Paris 6  
4, place Jussieu, F-75252 Paris Cedex 05, France  
{Fabrice.Kordon,Maximilien.Colange,  
Emmanuel.Paviot-Adet,Yann.Thierry-Mieg}@lip6.fr

<sup>2</sup> Centre Universitaire d'Informatique, Université de Genève  
7, route de Drize, CH-1227 Carouge, Switzerland  
{Alban.Linard,Didier.Buchs}@unige.ch

<sup>3</sup> LIPN, CNRS UMR 7030, Université Paris 13  
99, av. J-B Clément, 93430 Villetaneuse, France  
sami.evangelista@lipn.univ-paris13.fr

<sup>4</sup> Department of Information Technology, Uppsala University, Sweden  
kai.lampka@it.uu.se

<sup>5</sup> Universität Rostock, 18051 Rostock, Germany  
niels.lohmann@uni-rostock.de, Wimmel@Informatik.Uni-Oldenburg.de

**Abstract.** This article presents the results of the Model Checking Contest held within the SUMo 2011 workshop, a satellite event of Petri Nets 2011. This contest aimed at a fair and experimental evaluation of the performances of model checking techniques applied to Petri nets.

The participating tools were compared on several examinations (state space generation, deadlock detection and evaluation of reachability formulæ) run on a set of common models (Place/Transition and Symmetric Petri nets). The collected data gave some hints about the way techniques can scale up depending on both examinations and the characteristics of the models.

This paper also presents the lessons learned from the organizer's point of view. It discusses the enhancements required for future editions of the Model Checking Contest event at the Petri Nets conference.

**Keywords:** Petri Nets, Model Checking, Contest.

## 1 Introduction

When verifying a system with formal methods, such as Petri nets, one may have several questions such as:

“When creating the model of a system, should we use structural analysis or an explicit model checker to debug the model?”

“When verifying the final model of a highly concurrent system, should we use a symmetry-based or a partial order reduction-based model checker?”

“When updating a model with large variable domains, should we use a decision diagram-based or an abstraction-based model checker?”

Results that help to answer these questions are spread among numerous papers in numerous conferences. The choice of the models and tools used in benchmarks is rarely sufficient to answer these questions. Benchmark results are available a long time after their publication, even if the computer architecture has changed a lot. Moreover, as they are executed over several platforms and composed of different models, conclusions are not easy.

The objective of the Model Checking Contest @ Petri nets is to compare the efficiency of verification techniques according to the characteristics of the models. To do so, the Model Checking Contest compares tools on several classes of models with scaling capabilities, *e.g.*, values that set up the “size” of the associated state space.

Through a benchmark, our goal is to identify the techniques that can tackle a given type of problem identified in a “typical model”, for a given class of problem (*e.g.*, state space generation, deadlock detection, reachability or causal analysis, etc.).

The first edition of the Model Checking Contest @ Petri nets took place within the context of the SUMo workshop (International Workshop on Scalable and Usable Model Checking for Petri Nets and other models of concurrency), co-located with the Petri Nets and ACSD 2011 conferences, in Newcastle, UK. The original submission procedure was published early March 2011 and submissions gathered by mid-May 2011. After some tuning of the execution environment, the evaluation procedure was operated on a cluster early June. Results were presented during the SUMo workshop, on June 21st, 2011<sup>1</sup>.

Let us mention similar events we are aware of. The *Hardware Model Checking Contest*<sup>2</sup> started in 2007 focuses on circuit verification by means of model checking. It is now associated with the CAV (Computer Aided Verification) and FLOC (Federated Logic Conference) conferences. This event ranks the three best tools according to a selected benchmark. It is an almost yearly event (2007, 2008, 2010 and 2011).

The *Timing Analysis Contest*<sup>3</sup> within PATMOS 2011 (International Workshop on Power and Timing Modeling, Optimization and Simulation) also considers the verification of electronic designs with a focus on timing analysis.

The *Verified Software Competition*<sup>4</sup> hold within the Verified Software: Theories, Tools and Experiments (VSTTE) conference [24], in August 2010. This

<sup>1</sup> This presentation can be found at <http://sumo.lip6.fr/MCC-2011-report/MCC-2011-report.pdf>, and raw data of the benchmarks at <http://sumo.lip6.fr/MCC-2011-report/MCC-results-data.zip>

<sup>2</sup> <http://fmv.jku.at/hwmcc11/index.html>

<sup>3</sup> <http://patmos-tac.inesc-id.pt>

<sup>4</sup> <http://www.vscomp.org>



competition was held as a forum where researchers could demonstrate the strengths of their tools through the resolution of five problems. The main objective of this event was to evaluate the efficiency of theorem proving tools against SAT-solving.

The *Satisfiability Modulo Theories Competition*<sup>5</sup> takes place within the context of the CAV conference. Since 2005, its objective is to evaluate the decision procedures for checking satisfiability of logical formulas.

Finally, the *SAT Competitions*<sup>6</sup> proposes to evaluate the performance of SAT solvers. This event occurs yearly since 2002 and identifies new challenging benchmarks each year.

With respect to these existing events, the Model Checking Contest at Petri Nets puts emphasis on the specification of parallel and distributed systems and their qualitative analysis. So far, we consider Petri Nets as input specification (later editions might also consider other formalisms suitable for concurrency).

The goal of this paper is to report the experience from this first Model Checking Contest. It reflects the vision of the MCC'2011 organizers, as it was first presented aside the conferences in Newcastle, revised and augmented by feedback from the tool developers who participated in this event. All tool developers are listed in Section 9.

The article is structured as follows. Section 2 presents the evaluation methodology, before a brief presentation of the models in Section 3 and the participating tools in Section 4. Then, Sections 5 to 7, detail some observations we made about the efficiency of techniques with regards to their implementation in the participating tools. Finally, Section 8 discusses some issues risen by this first edition of the Model Checking Contest as well as some clues for the organization of next editions.

---

All over the paper, we outline in this way the lessons learned from the first edition. These lessons constitute changes to be applied in the next edition of this event.

## 2 Evaluation Methodology

The Model Checking Contest is based on one major assumption: there is no silver bullet in model checking. Thus, the choice of the techniques should depend on:

- the characteristics of the model, for instance its formalism (Place/Transition net or Symmetric net), the marking bounds (safe or unsafe nets), or the number of synchronizations in a model;
- the action to be performed by the model checker, for instance state space generation, deadlock detection, or evaluation of reachability, CTL or LTL formulæ;
- the possible interaction between techniques, for instance abstractions with partial orders, or decision diagrams with symmetries;
- the position in the development process, for instance when the model is being designed, during its debugging, or during its final checking.

<sup>5</sup> <http://www.smtcomp.org>

<sup>6</sup> <http://www.satcompetition.org>

There is already plenty of publications on theoretical complexity of model checking algorithms. Theoretical complexity is sometimes misleading, as it can hide huge constants, that make algorithms unusable in practice. Moreover, the efficiency of tools varies a lot, even when they use the same techniques.

An experimental evaluation of tools efficiency is thus required in model checking. Articles about tools also provide benchmarks that have numerous drawbacks [22]:

- They usually cover only some selected tools on some selected models. Thus, benefits of involved algorithms for some model characteristics cannot be evaluated.
- The choice of the tools used for comparison is sometimes biased, because the authors may not know other good competing tools, or because they could not convert their models to process them.
- There is no guarantee that the comparison is fair, because the authors of the article may not know other tools as well as their own tool. For instance, some settings can require some expertise to be set appropriately.
- As benchmarks are performed on several architectures, they also cannot be compared between articles.

## 2.1 The Overall Procedure

For the first edition of the Model Checking Contest, only a subset of all actions provided by model checkers were requested. They are called “examinations”:

- computation of the state space with a report on its size;
- computation of the deadlocks with a report on their number;
- evaluation of reachability formulæ to detect whether a state, depicted by a logic formula, can be reached from the initial state or not (10 satisfiable ones and 10 unsatisfiable ones) with a report on the computed results (true or false).

Examinations were run on several parameterized models. Each model has a “scaling value”, used to increase its complexity, for instance the number of philosophers in the Dining Philosophers problem. For each model, each scaling parameter value defines a *model instance*. The Model Checking Contest provided all model instances in Petri Net Markup Language format. As tools were allowed to give their own version of the models, as long as examinations return the same results, we fixed all scaling values before the contest, and provided them in advance to the participants.

Model checking has two “enemies”: memory consumption when it comes to store large state spaces (or portion of state space) and computation time, when the number of states grows. There is usually a trade-off between lower memory footprint and lower computation time. Thus, our objective was to measure both memory and CPU usage.

The “examinations” requested for the contest were performed thanks to an invocation script that iterated invocation of each tool over models instances. This invocation script is presented in Algorithm 1.

```

Input:  $M$ , a set of scalable models to be processed
foreach  $m \in M$  do
  Operate a prologue for  $m$ 
  foreach  $v$ , scaling values for  $m$  do
    Operate state space generation on  $m$  for scale value  $v$ 
    Operate deadlock computation on  $m$  for scale value  $v$ 
    Operate check of satisfiable reachability formulæ on  $m$  for scale value  $v$ 
    Operate check of unsatisfiable reachability formulæ on  $m$  for scale value  $v$ 
  Operate an epilogue for  $m$ 

```

**Algorithm 1.** Actions performed for each tool by the invocation script

When a tool fails on a model for a particular scaling value, we still try the tool for higher values. This ensures that a tool can fail for any reason on a scaling parameter, and still compete for other values. It has a cost, as we cannot exit the loop and thus save overall computation time for the contest. As we can expect an increase in the number of participating tools, the number of models and the number of scaling values, we might consider this optimization in the next Model Checking Contest.

*Prologue and epilogue.* A prologue is executed prior to any execution performed on a model. This prologue enables one to prepare data for a given model if necessary. For example, **LoLA** used this prologue to compile itself on the executing machine, thus avoiding library compatibility problems.

When all examinations have been processed for all the scaling values of a given model, an epilogue is executed, typically to enable tools to delete temporary files.

There is no time or memory usage measures for the prologue and the epilogue actions. These actions are considered as “preparations” for the contest. Tool developers were free to put any preprocessing of the model into the prologue. Some tools like **LoLA** compile the tool during the first execution of the prologue. Some others include their preprocessing inside the examinations: for instance **PNXDD** unfolds the model and computes a DD variable order during examinations.

To measure the whole computation time for all tools, we should, in the next edition, measure time and memory spent in the prologue and epilogue, making the computation time more comparable between tools

*Examination.* Other actions required by the model checkers are executed in a confined environment, to restrict the total execution time of the full evaluation. Both time and memory are reported in a log file. The way confinement and measures were performed is presented in Section [2.2](#).

Operating a command is performed through a wrapper script customized by the tool submitter. This script must report results of the examination in a standardized and structured way. The results are the number of states in state space generation, the number of deadlocks, or the evaluation of a formula. Moreover, the tool must list the techniques used to work on the examination.

These techniques may be specific to a given examination, the value of a scaling parameter or the processed model.

## 2.2 Confinement and Measure of CPU and Memory Usage

Evaluation of tools was performed on a cluster of 24 hyper-threaded 2.4 GHz bi-Pentium Xeon 32 bits with 2 GBytes of memory and running Linux Mandriva 2010-2. Monitored actions were confined to the following constraints: 1 800 seconds of CPU and 1.75 GBytes of maximum memory<sup>7</sup>.

Prior to the submission deadline, we were using for our tools a solution that appeared to be efficient: `memtime`<sup>8</sup> from the UPPAAL community. This program works similarly to the `time` Unix command, but reports both CPU time and memory allocation up to a maximum that can be easily configured.

Once submissions were collected, we discovered that reported memory usage was only concerning the top process (a shell script due to the wrapper script encapsulation technique), even if memory confinement was working well. We thus only used `memtime` for its confinement capabilities and CPU measures associated with the `memusage` command that provides a log of all allocations in the system. We evaluated memory consumption by parsing these log files. The idea is to show the memory use peak, that corresponds to the “user feeling”.

The solution we elaborated allowed us to perform measures in a satisfactory way. However, it was too intrusive because it is based on an interposition library that overrides memory allocation and has an impact on performances.

Also, the computation of a diagnostic of a failure (time or memory exhaustion) could not be fully automated, some case having to be checked manually from the log themselves like, in `LoLA` or `PNXDD`, where some memory overflow were initially detected as stack overflow.

---

For the next edition, executions will be run on virtual machines (e.g. QMU) that could be monitored from “outside”, thus allowing more flexibility and safety in measures, as well as the support of other operating systems (such as Windows).

## 3 The Selected Models

For this first edition, seven models were selected: three Place/Transition nets, and four colored nets modeled in Symmetric nets. These models were selected from known and reused benchmarks. We provide a brief description of the models here. Due to lack of space, we cannot provide a picture of the models in this article, but it can be found in the Model Checking Contest web site<sup>9</sup>.

Note that for this first edition, the scaling value of all P/T models increased the number of tokens in places, but did not change the structure of the net. On the contrary, the scaling value of all Colored models increased the number of places and transitions in the Equivalence P/T net, but not the number of tokens.

<sup>7</sup> To leave 250 MBytes for the operating system.

<sup>8</sup> <http://www.uppaal.org/>

<sup>9</sup> <http://mcc.lip6.fr/2011>

**The Models.** Let us first provide a brief description of the proposed models.

FMS belongs to the **GreatSPN** and **SMART** [10] benchmarks. It models a Flexible Manufacturing System [9]. The scaling parameter corresponds to the number of initial tokens held in three places. The following values were used: 2, 5, 10, 20, 50, 100, 200, 500.

Kanban [8] models a Kanban system. The scaling parameter corresponds to the number of initial tokens held in four places. The following values were used: 5, 10, 20, 50, 100, 200, 500, 1 000.

MAPK models a biological system: the Mitogen-Activated Protein Kinase Cascade [20]. The scaling parameter changes the initial number of tokens held in seven places. The following values were used: 8, 20, 40, 80, 160, 320.

Peterson models Peterson’s mutual exclusion algorithm [35] in its generalized version for  $N$  processes. This algorithm is based on shared memory communication and uses a loop with  $N - 1$  iterations, each iteration is in charge of stopping one of the competing processes. The scaling factor is the number of involved processes. The following values were used: 2, 3, 4, 5, 6.

Philosophers models the famous Dining Philosophers problem introduced by E.W. Dijkstra in 1965 [41] to illustrate an inappropriate use of shared resources, thus generating deadlocks or starvation. The scaling parameter is the number of philosophers. The following values were used: 5, 10, 20, 50, 100, 500, 1 000, 5 000, 10 000, 50 000, 100 000.

SharedMemory is a model taken from the **GreatSPN** benchmarks [7]. It models a system composed of  $P$  processors competing for the access to a shared memory (built with their local memory) using a unique shared bus. The scaling parameter is the number of processors. The following values were used: 5, 10, 20, 50, 100, 200, 500, 1 000, 2 000, 5 000, 10 000, 20 000, 50 000.

TokenRing is another problem proposed by E.W. Dijkstra [14]. It models a system where a set of machines is placed in a ring, numbered 0 to  $N - 1$ . Each machine  $i$  only knows its own state and the state of its left neighbor, i.e., machine  $(i - 1) \bmod (N)$ . Machine number 0 plays a special role, and it is called the “bottom machine”. A protocol ensuring non-starvation determines which machine has a “privilege” (e.g. the right to access a resource). The scaling parameter is the number of machines. The following values were used: 5, 10, 15, 20, 30, 40, 50, 100, 200, 300, 400, 500.

**Characteristics of the Models.** All the selected models are bounded. Their main characteristics are summarized in Table 1. None of the Place/Transition nets is safe (or 1-bounded) because the scaling parameter affects the initial marking. On the contrary, all colored models are safe (in the colored sense where each color cannot appear more than once in a marking) because the scaling parameter only changes the color types.

We also note some characteristics of our colored models. First, color types are either a range of integers, or cartesian products of them. There are two types of

**Table 1.** Summary of model’s characteristics

		Safe	Cartesian product of color types	Non equal guards	Broadcast function	Succ & pred functions
P/T	FMS					
	Kanban					
	MAPK					
Colored	Peterson	✓	✓	✓	✓	✓
	Philosophers	✓			✓	✓
	SharedMemory	✓	✓	✓	✓	
	TokenRing	✓	✓	✓		✓

guards: the ones using equality only ( $=$  or binding with the same input variable) and others ( $\neq, <, >, \dots$ ) that are interesting because they generate asymmetries in the state space. Arc labels can be a single constant or variable, or use the “broadcast” that is the set containing all values in a color type. Arcs and guards may also use incrementation ( $+n$ ) or decrementation ( $-n$ ) operators. Finally, let us note that the “broadcast” can be used to define the initial marking (it is then a dense one, e.g. all values of the color domain are represented).

When analyzing results of the Model Checking Contest, we observed there was no safe Place/Transition nets and no unsafe Colored nets. In the 2012 Model Checking Contest, we will scale the Petri nets also by their structure, by providing the Place/Transition nets equivalents for all Colored nets. For this first edition we provided two kinds of scaling parameters: one based on the number of tokens in places, the other based on color domains. For the next edition, we should also provide a mix between them, and various models with no scaling parameters, such as industrial cases.

## 4 Participating Tools

Ten tools were submitted. They are summarized in Table 2.

**Table 2.** Summary of data on participating tools

Tool Name	Team	Institution	Country	Contact Name
ACTIVITY-LOCAL	TIK	ETHZ	Switzerland	K. Lampka
ALPiNA	CUI/SMV	Univ. Geneva	Switzerland	D. Buchs
Crocodile	LIP6/MoVe	UPMC	France	M. Colange
ITS-Tools	LIP6/MoVe	UPMC	France	Y. Thierry-Mieg
LoLA	Team Rostock	Univ. Rostock	Germany	N. Lohmann & K. Wolf
PNXDD	LIP6/MoVe	UPMC	France	E. Paviot-Adet
PeTe	Stud. Group d402b	Univ. Aalborg	Denmark	J. Finnemann Jensen
Sara	Team Rostock	Univ. Rostock	Germany	H. Wimmel & K. Wolf
YASPA	TIK	ETHZ	Switzerland	K. Lampka
helena	LIPN/LCR	Univ. Paris 13	France	S. Evangelista

**Tool Description.** We provide here a brief description of the participating tools.

**ACTIVITY-LOCAL**<sup>[10]</sup> [29] works on any type of Place/Transition nets with inhibitor arcs and weighted arcs. It combines decision diagram-based state space encoding with explicit state space exploration. To avoid the Peak problem observed for decision diagrams with incremental generation, this tool composes them in an original way. The transition relation induced by the same transition of the P/T net is encapsulated in its own “submodel”. **ACTIVITY-LOCAL** executes an explicit state space traversal for each of these submodels and inserts the detected state-to-state transitions into the corresponding DD (one per submodel).

To cope with dependencies among the transitions of the P/T net, **ACTIVITY-LOCAL** structures the explicit exploration as a *selective* breadth-first scheme. It only explores the transitions that are in a dependency set of the analyzed transition.

When a local fixed point is reached, **ACTIVITY-LOCAL** performs a symbolic reachability analysis used to elaborate the complete state space. This second step is implemented as a partitioned symbolic reachability analysis [5], using greedy chaining [34] and a new DD operator [28].

**ALPiNA**<sup>[11]</sup> [23] stands for Algebraic Petri nets Analyzer and is a symbolic model checker for Algebraic Petri nets. It can verify various state properties expressed in a first order logic property language.

Algebraic Petri nets (APNs) (Petri nets + Abstract Algebraic Data Types) is a powerful formalism to model concurrent systems in a compact way. Usually, concurrent systems have very large sets of states, that grow very fast in relation to the system size. Symbolic Model Checking (DD-based one) is a proven technique to handle the State Space Explosion for simpler formalisms such as Place/Transition nets. **ALPiNA** extend these concepts to handle algebraic values that can be located in net places.

For this purpose **ALPiNA** uses enhanced DDs such as Data Decision Diagrams and Set Decision Diagrams for representing the place vectors and  $\Sigma$  Decision Diagrams [3] for representing the values of the APN. It also allows to specify both algebraic and topological clusters to group states together and thus to reduce the memory footprint. Particular care has been taken to let users freely model their systems in APNs and in a second independent step to tune the optimization parameters such as unfolding rules, variable order, and algebraic clustering. Compared to Colored net approaches, **ALPiNA** [4] solves problems related to the unbounded nature of data types and uses the inductive nature of Abstract Algebraic Data Types to generalize the unfolding and clustering techniques to any kind of data structure.

**ALPiNA**’s additional goal is to provide a user friendly suite of tools for checking models based on the Algebraic Petri nets formalism. In order to provide great user experience, it leverage on the powerful eclipse platform.

<sup>10</sup> No official distribution yet.

<sup>11</sup> Tool is available at <http://alpina.unige.ch>.

**Crocodile**<sup>12</sup> [12] was initially designed as a demonstration tool for the so-called symbolic/symbolic approach [39]. It combines two techniques for handling the combinatorial explosion of the state space that are both called “symbolic”.

The first “symbolic” technique concerns the reduction of the reachability graph of a system by its symmetries. The method used in **Crocodile** was first introduced in [6] for the Symmetric nets, and was then extended to the Symmetric nets with Bags (SNB) in [18]. A symbolic reachability graph (also called quotient graph) can be built for such types of Petri nets, thus dramatically reducing the size of the state space.

The second “symbolic” technique consists in storing the reachability graph using decision diagrams, leading to a symbolic memory encoding. **Crocodile** relies on Hierarchical Set Decision Diagrams [13]. These present several interesting features, such as hierarchy, and the ability to define inductive operations.

Still under development, **Crocodile** essentially generates the state space of a SNB and then processes reachability properties.

**ITS-Tools**<sup>13</sup> [40] are a set of tools to analyze Instantiable Transition Systems, introduced in [40]. This formalism allows compositional specification using a notion of type and instance inspired by component oriented models. The basic elementary types are labeled automata structures, or labeled Petri nets with some classical extensions (inhibitor arcs, reset arcs...). The instances are composed using event-based label synchronization.

The main strength of **ITS-Tools** is that they rely on Hierarchical Set Decision Diagrams [13] to perform analysis. These decision diagrams support hierarchy, allowing to share representation of states for some subsystems. When the system is very regular or symmetric, recursive encodings [40] may even allow to reach logarithmic overall complexity when performing analysis. Within the contest, the Philosophers and TokenRing examples proved to be tractable using this recursive folding feature.

Set Decision Diagrams also offer support for automatically enabling the “saturation” algorithm for symbolic least fixpoint computations [19], a feature allowing to drastically reduce time and memory consumption. This feature was used in all computations.

**LoLA**<sup>14</sup> [43] is an explicit Petri net state space verification tool. It can verify a variety of properties ranging from questions regarding single Petri net nodes (*e.g.*, boundedness of a place or quasiliveness of a transition), reachability of a given state or a state predicate, typical questions related to a whole Petri net (*e.g.*, deadlock freedom, reversibility, or boundedness), and the validity of temporal logical formulae such as CTL. It has been successfully used in case studies from various domains, including asynchronous circuits, biochemical reaction chains, services, business processes, and parameterized Boolean programs.

<sup>12</sup> Tool is available at <http://move.lip6.fr/software/Crocodile>.

<sup>13</sup> Tool is available at <http://ddd.lip6.fr>.

<sup>14</sup> Tool is available at <http://www.informatik.uni-rostock.de/tpp/lola>.



For each property, **LoLA** provides tailored versions of state space reduction techniques such as stubborn sets, symmetry reduction, coverability graph generation, or methods involving the Petri net invariant calculus. Depending on the property to be preserved, these techniques can also be used in combination to only generate a small portion of the state space.

For the Model Checking Contest, only one configuration of **LoLA** was submitted since, in the beginning, the necessary efforts were not predictable. This configuration was tailored for checking the reachability of a state that satisfies a given state predicate. This check was combined with two reduction techniques. First, a dedicated version of the stubborn sets [37] aimed at exploiting concurrency in the model and that allows to prioritize the firing of those transitions that lead to states closer to the goal state. This method is known to perform extremely well on reachable states while other methods [27] also available in **LoLA** would excel on unreachable states. Second, **LoLA** calculates place invariants to determine so-called implicit places [38]. The marking of such places does not need to be stored explicitly, but can be deduced from the marking of the other places. Typically, this reduction allows to reduce the memory usage by 20% to 60%.

**PNXDD**<sup>[15]</sup> generates the state-space of Place/Transition nets. When Colored nets are used in the Model Checking Contest, equivalent P/Ts are obtained after an “optimized” unfolding [25] (unused places and transitions are detected and suppressed).

State Space storage relies on Hierarchical Set Decision Diagrams [13] (SDDs). These are decision diagrams with any data type associated to arcs (see *e.g.*, [32] for an overview of DD-like structures). If the associated data type is another SDD, hierarchical structures can be constructed.

Since **PNXDD** exploits hierarchy, a state is seen as a tree, where the leaves corresponds to places marking. This particular structure offers greater sharing opportunities than a, for instance, vector based representation. The conception of such a tree is critical to reach good performances and heuristics are being elaborated for this purpose [21]. The one used for the Model Checking Contest is based on [1]: for colored models that do scale via the size of color types, **PNXDD** uses a tree-like version of this heuristic, while the original version is kept when colored models only scale via the number of tokens in the initial marking.

**PeTe**<sup>[16]</sup> is a graphical Petri net modeling and verification tool written in C++/Qt. **PeTe** can answer reachability formulæ using two techniques. First, **PeTe** attempts to disprove reachability of a query using over-approximation. This is done by solving the state equation using integer programming. If a solution is found **PeTe** attempts to tighten the approximation using trap testing as presented in [15]. Detailed description and variations of this approach can be found in [15].

If over-approximation cannot disprove reachability, **PeTe** attempts to prove reachability with straightforward Best-First-Search of the state space, using a

<sup>15</sup> Tool is available at <https://srcdev.lip6.fr/trac/research/NEOPPOD/wiki/pnxdd>.

<sup>16</sup> Tool is available at <https://github.com/jopsen/PeTe>.

simple heuristic presented in [17]. So far, **PeTe** does not support state space generation or deadlock detection. **PeTe** is maintained and available under GNU GPLv3.

**Sara**<sup>17</sup> uses the state equation, known to be a necessary criterion for reachability and in a modified way also for other properties like coverability, to avoid enumerating all possible states of a system [36]. A minimal solution of the state equation in form of a transition vector is transformed into a tree of possible firing sequences for this solution. A firing sequence using all the transitions given in the solution (with the correct multiplicity) reaches the goal.

For tree pruning, partial order reduction is used, especially in the form of stubborn sets [37,26]. If the goal cannot be reached using the obtained solution, places that do not get enough tokens are computed. Constraints are built and added to the state equation (in a CEGAR-like fashion [11]). These constraints modify the former solution by adding transition invariants, temporarily allowing for additional tokens on the undermarked places.

**Sara** detects unreachable goals either from an unsatisfiable state equation or by cutting the solution space to finite size when the repeated addition of transition invariants is known not to move towards the goal. A more involved explanation of the algorithm behind **Sara** can be found in [42].

**YASPA**<sup>18</sup> relies on Zero-Suppressed Decision Diagrams (ZDDs) [33,31] together with a partitioned symbolic reachability analysis [5] and greedy chaining [34]. However, contrary to other decision diagram-based tools it neither depends on pre-generated symbolic representations of state-to-state transitions, nor on the use of standard decision diagram operators. Instead, symbolic reachability analysis is carried out by means of customized ZDD-algorithms that are directly synthesized from the Place/Transition net with inhibitor arcs.

As a key feature the synthesized ZDD operators are organized in a strictly local manner. This is achieved by assigning an identity semantics to those variables of the decision diagram which refer to places that are neither pre nor post condition of a given transition. Moreover, the ZDD operators apply decision diagram-related recursion rules which implement the subtraction, addition and testing of tokens.

By executing these synthesized ZDD operators in a fixed point iteration, **YASPA** delivers the state space and transition relation of the system.

**helena**<sup>19</sup> [16] is an explicit state model checker for High Level Petri nets. It is a command-line tool available available under the GNU GPL.

**helena** tackles the state explosion problem mostly through a reduction of parallelism operated at two stages of the verification process. First, static reduction rules are applied on the model in order to produce a smaller net that – provided some structural conditions are verified – is equivalent to the original one but has a smaller reachability graph. Second, during the search, partial order reduction is employed to limit, as much as possible, the exploration of redundant paths in the reachability graph. This reduction is based on the detection of independent

<sup>17</sup> Tool is available at <http://www.service-technology.org/tools/download>.

<sup>18</sup> Tool is available at <http://www.tik.ee.ethz.ch/~klampka>.

<sup>19</sup> Tool is available at <http://helena-mc.sourceforge.net>.

transitions of the net at a given marking. Other reduction techniques are also implemented by *helena*, e.g., state compression, but were disabled during the contest due to their inadequacy with the proposed models.

**Summary of Techniques Used by Participating Tools.** Altogether, these tools implement numerous techniques, as summarized in Table 3. We note that several tools stack several techniques such as decision diagrams with (sometimes) symmetries, or abstractions with partial orders.

We also note that numerous types of decision diagrams are used in participating tools. YASPA uses customized Zero-Suppressed Multi-Terminal Decision Diagrams [30]. ITS-Tools, PNXDD, Crocodile and ALPiNA are using Hierarchical Set Decision Diagrams [40]. ALPiNA also uses a variant called  $\Sigma$  Decision Diagrams dedicated to algebraic systems [3].

**Table 3.** Summary of techniques used by tools

	<b>Reachability Graph</b>	<b>Deadlock Detection</b>	<b>Formula Evaluation</b>
ACTIVITY-LOCAL	Explicit Decision Diagrams		
ALPiNA	Decision Diagrams	Decision Diagrams	
Crocodile	Symmetries Decision Diagrams		Symmetries Decision Diagrams
ITS-Tools	Decision Diagrams Symmetries (opt)	Decision Diagrams Symmetries (opt)	Decision Diagrams Symmetries (opt)
LoLA			Explicit Partial Orders State Compression
PNXDD	Decision Diagrams		
PeTe			Explicit State Equation
Sara			Abstractions Partial Orders State Equation
YASPA	Decision Diagrams		
helena	Explicit	Explicit Abstractions Partial Orders	

---

||In the next edition, more precision will be required to classify techniques.

## 5 Observations on State Space Generation

This section analyzes the results of the Model Checking Contest for state space generation. It first presents the highest parameter computed by the tools for each model. Then it compares in Section 5.1 the maximum parameter reached, together with the evolution of computation time and memory consumption on

Place/Transition net models, before doing the same analysis on Colored net models in Section 5.2. We have found this distinction to be the most significant for state space generation.

Table 4 summarizes the highest parameter reached by the tools for each model. This table, as well as Tables 5 and 6, should be interpreted using the legend below:

	The tool does not participate.
	The tool participates, but cannot compute for any scaling value.
	The tool participates.
	The tool participates and reaches the best parameter among tools.
	The tool participates and reaches the maximum parameter.
	The tool fails for an unknown reason, after reaching parameter $n$ .
	The tool fails because of memory exhaustion, after reaching parameter $n$ .
	The tool fails because of maximum time is elapsed, after reaching parameter $n$ .

Table 4. Results for the state space generation examination

		ACTIVITY-LOCAL	ALPINA	Crocodile	ITS-Tools	LoLA	PNXDD	PeTe	Sara	YASPA	helena
P/T	FMS	20	10	2	100		200			100	2
	Kanban	20	10		200		500			50	
	MAPK		8		160		160			8	
Colored	Peterson		3				5				2
	Philosophers		500	10	100 000		1 000				10
	SharedMemory		20	20	50 000		100				10
	TokenRing		10		50		15				10

Table 3 shows that almost all the tools competing for state space generation use decision diagrams. So, this technique seems to be the most common choice when doing state space generation. From Table 4, we observe that among DD-based tools, there is a great variation in the maximum scaling parameter reached. The ratio between the value reached by the worst and the best DD-based tools is 1 : 5 for TokenRing and 1 : 10 000 for Philosophers.

Comparing tools for the state space examination is not a trivial task. The Model Checking Contest organizers encountered several problems, all concerning the returned size of the state space, which was initially used to check the answers:

1. For **helena**, both tool developers and the Model Checking Contest organizers agreed to disable all optimizations – structural reductions and state compression – because they lead to the generation of a 1-state reachability graph. It did not seem to make sense in this examination.
2. **Crocodile** on Colored net also returns fewer states, because it is computing the quotient reachability graph.
3. We also noted, for FMS and TokenRing, a variation in the state space size, that was apparently due to some variation in the encoding of the model. Some tools, like **ACTIVITY-LOCAL** and **YASPA** adapted the model taken from **GreatSPN**, for instance by removing instantaneous transitions. These variations did not seem large enough to require the tool developers to check their models and tools.

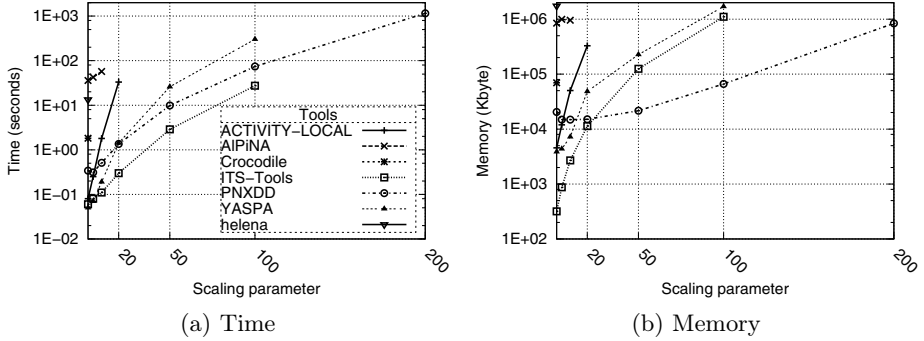


Fig. 1. Memory and time measure for state space generation on the FMS model

## 5.1 Place/Transition Net Models

Note that, for Place/Transition net models, the sizes of the state space for the highest parameter reached by the tools are:

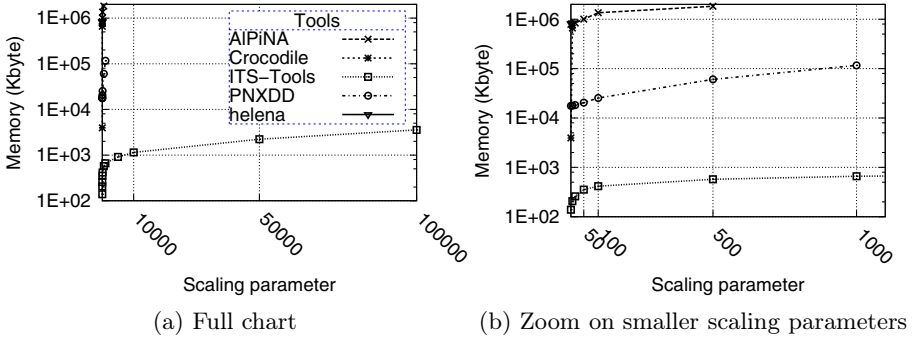
$$\begin{aligned} \text{FMS}_{200} &= 1.95 \times 10^{25} \text{ states} \\ \text{Kanban}_{500} &= 7.09 \times 10^{26} \text{ states} \\ \text{MAPK}_{160} &= 1.06 \times 10^{23} \text{ states} \end{aligned}$$

Figure 1 shows the memory and time evolution of state space generation for the FMS model. It is typical of what we observe for this examination on Place/Transition nets. It experimentally shows that we can divide DD-based tools into two groups: the first one (“bad results”) only reaches  $\text{FMS}_{20}$  (around  $6 \times 10^{12}$  states) at most, the second one (“good results”) reaches  $\text{FMS}_{100}$  (around  $2.7 \times 10^{21}$  states) at most.

The “bad results” group is composed of **Crocodile**, **ALPiNA** and **ACTIVITY-LOCAL**. All these tools are not dedicated to Place/Transition nets: **Crocodile** is intended for Colored nets with bags in tokens, **ALPiNA** is optimized for Algebraic Petri nets, and **ACTIVITY-LOCAL** works on any type of P/T nets with inhibitor arcs and weighted arcs. These three tools do not get better results on the two other P/T models, Kanban and MAPK. **Crocodile** has bad performances because it does not know how to exploit symmetries from Place/Transition nets; moreover, it appeared that management of multisets of tokens needed some improvement. The developers of **ALPiNA** discovered that it has bad performance for P/Ts because the tool is implicitly optimized for safe Petri nets.

On the contrary, tools that handle formalisms closer to the Place/Transition nets obtain good results. **YASPA**, **ITS-Tools** and **PNXDD** handle at least  $\text{FMS}_{100}$ . **YASPA** is a bit less effective on Kanban, and is comparable to the “bad tools” for MAPK. We noted that for Kanban, the results are not consistent with measures made by the author of **YASPA**, that shows similar performance as for FMS (we could not find any explanation for this).

Among the “good results” group, we can see that **PNXDD** has better results than **ITS-Tools** for FMS and Kanban. The explanation is that for these nets, **PNXDD**



**Fig. 2.** Memory measure for state space generation (Philosophers)

does not use hierarchical DDs, contrary to **ITS-Tools**. Because the scaling value only increases the number of tokens, and not the size of the net, the cost of using hierarchy is not covered by the gains it provides.

No tool could reach the maximum scaling value for Place/Transition nets (500 for FMS, 1000 for Kanban, 320 for MAPK). As these numbers have been selected based on known results in papers, this is not surprising. On the contrary **PNxDD** is close to the maximum parameters. For the next Model Checking Contest, scaling parameter of the 2011 models will be increased. This analysis of the examinations should be done each year, in order to increase tools efficiency, as it was observed in the SAT Competition.

## 5.2 Colored Net Models

We provide in Figure 2 the memory measure for the Philosophers model and in Figure 3 the CPU consumption for SharedMemory. Since one technique is very efficient, the leftmost part of the figures show measures for all the scaling parameter while the rightmost part only focus on the subset of values where all tools provide results. These figures are of interest because they show some extreme performance of some techniques in favorable cases.

Execution of tools on colored models showed interesting points:

- **helena** obtains results comparable to some decision diagram-based tools on SharedMemory and TokenRing (see Figure 4 for TokenRing). As all optimizations of **helena** are disabled for this examination, it shows that these DD-based tools are quite inefficient for these models;
- **Crocodile** has heterogeneous results: it is as good as **ALPiNA** on SharedMemory (see Figure 3b), but reaches only a low parameter on Philosophers (see Figure 2b). This is apparently due to a non optimal exploitation of symmetries; optimization could also be performed on the implementation;
- **ITS-Tools** reaches impressive parameters compared to the other tools on the Colored net models it handled.

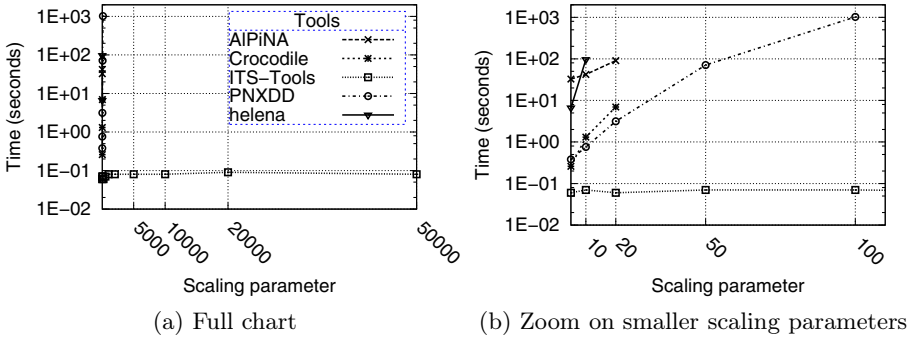


Fig. 3. CPU measure for state space generation (SharedMemory)

Thus, we can divide the tools into two groups: the “quite good” one containing all tools except **ITS-Tools**, and the “excellent” one containing this last tool.

Figures 2a and 3a illustrate the real interest of a technique, used in **ITS-Tools**, associated to hierarchical decision diagrams: “recursive folding” [40]. This technique can be activated for very regular models such as Philosophers and SharedMemory. It consist in splitting recursively the model in subcomponents. First, the system is split in “two halves”, and then each half in “two fourth”, etc. Associated with the hierarchical decision diagrams used in **ITS-Tools**, the result is impressive: this tool is able to process both models for the maximum provided values. In both cases, the number of states exceeds the floating points representation. For smaller parameters, the state space sizes are given below:

$$\begin{aligned} \text{Philosophers}_{10\,000} &= 1.63 \times 10^{4771} \text{ states} \\ \text{SharedMemory}_{10\,000} &= 5.43 \times 10^{4778} \text{ states} \end{aligned}$$

Figures 2b and 3b are a zoom on the left part of figures 2a and 3a. It shows the performances of “second tools” that correspond to the following state space size:

$$\begin{aligned} \text{Philosophers}_{1\,000} &= 1.13 \times 10^{477} \text{ states} \\ \text{SharedMemory}_{100} &= 5.15 \times 10^{47} \text{ states} \end{aligned}$$

Figure 4 shows measures for the TokenRing model where the recursive folding technique cannot be activated. Decision diagram-based tools are much less performant than previously, the largest computed state space holds  $1.98 \times 10^{27}$  states “only”.

Apart the results of **ITS-Tools**, the results of **PNxDD** and **ALPiNA** are useful for another remark. When comparing these two tools on Philosophers, SharedMemory and TokenRing, we see that **ALPiNA** and **PNxDD** have comparable results on Philosophers and TokenRing, whereas **PNxDD** is far better on SharedMemory (due to different hierarchical structures). From this, we can deduce that Hierarchy in decision diagrams offers interesting results.

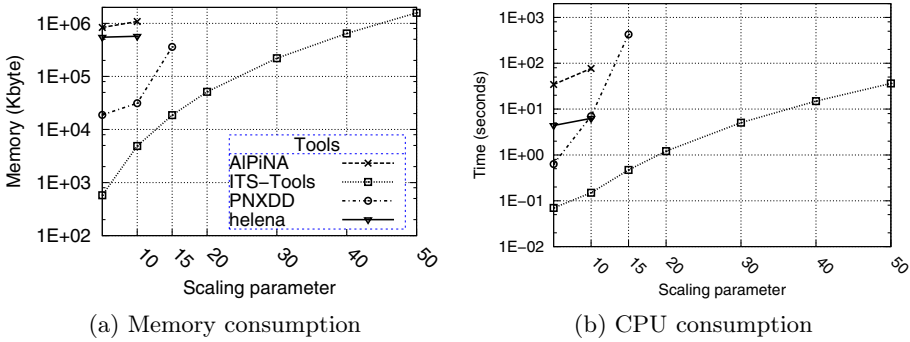


Fig. 4. Measure for state space generation (TokenRing)

There is a large difference between the performances of DD-based tools for Philosophers and SharedMemory (many orders of magnitude for the values of both scaling parameters), whereas the difference is lower for TokenRing. But ITS-Tools and PNxDD should get closer results on TokenRing, as there is no recursive folding for this model. Although both tools use the underlying Place/Transition nets for this colored model, the unfolding of color, as well as the construction of the hierarchical structure, are not the same.

Decision diagram based tools clearly can reach impressive scaling values for state space generation, when using hierarchical decision diagrams together with recursive folding. To do state space generation, we can recommend three tools: PNxDD is very efficient and works both for Place/Transition nets and Colored nets, YASPA is dedicated to Place/Transition nets and has heterogeneous results, and ITS-Tools is extremely efficient on Coloreds but requires to manually transform the model.

Academic models seem easy for the good state space generators. We should provide some industrial models in the next Model Checking Contest, as they are usually not as regular as academic models.

The Peterson model seems reluctant to all the implemented techniques. Only three tools could handle it, and the best processed scaling values are very low: 5 for PNxDD, 3 for ALPiNA and 2 for helena. This corresponds to very small state spaces compared to the ones reached for other models:

$$\text{Peterson}_3 = 2.07 \times 10^4 \text{ states}$$

$$\text{Peterson}_5 = 6.30 \times 10^8 \text{ states}$$

## 6 Observations on Deadlock Detection

The data collected for deadlock detection is summarized in Table 5. It must be read as Table 4. Let us note that only three tools did participate in this examination.



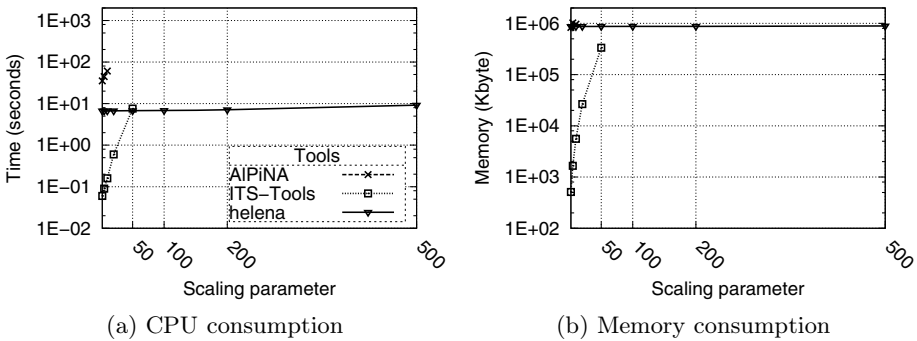
**Table 5.** Results for the deadlock detection examination

		ACTIVITY-LOCAL	ALPiNA	Crocodile	ITS-Tools	LoLA	PNXDD	PeTe	Sara	YASPA	helena
P/T	FMS		10 ☹		50						500
	Kanban		10 ☹		200						10
	MAPK		8 ☹		160						
Colored	Peterson		3 ☹								3 ☹
	Philosophers		100 ☹		100 000						10 ☹
	SharedMemory		20 ?		50 000						10 ☹
	TokenRing		10 ?		50						10 ☹

As this examination required to count the number of deadlocks instead of just discovering one, some tools could not participate. For instance, **LoLA** stops when the first deadlock is found. We should only ask for the detection of at least one deadlock in the next edition of the Model Checking Contest, to have more competing tools. We will also propose to refer to deadlocks in formula to be evaluated.

In Table 4, we see that **helena** has an inconstant behavior. It works very well for FMS, reaching the maximum scaling value in constant time and memory, as shown in Figure 5. On the contrary, this tool handles only small instances for the other models. It shows that abstractions and partial orders provide good results is this case where both CPU and memory usage are almost constant. The abstraction mechanism used by **helena** is based on Berthelot’s structural reductions [2], that remove transitions irrelevant from a concurrency perspective. FMS is a perfect case for that method in that it exhibits a lot of parallelism but little concurrency.

For all models, except FMS and Peterson, and especially Colored ones, the results are close to state space generation: decision diagram based tools obtain



**Fig. 5.** Measures for deadlock examination on FMS

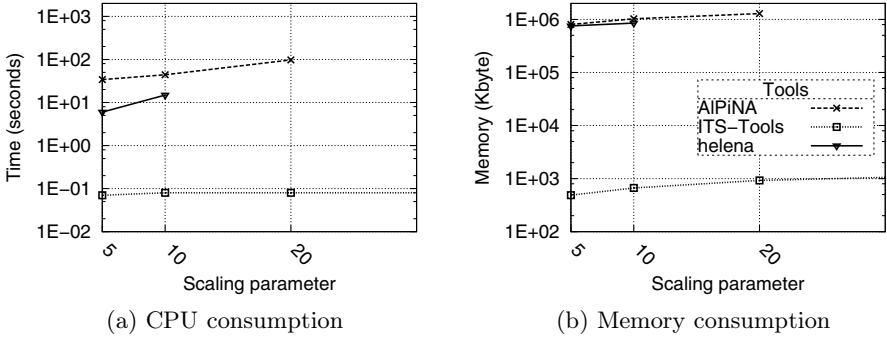


Fig. 6. Measures for deadlock examination on SharedMemory

quite good results, especially **ITS-Tools** when recursive folding can be applied. Figure 6 shows the evolution of CPU time and memory consumption for the SharedMemory model, where recursive folding enables **ITS-Tools** to process all instances. The dominance of decision diagram techniques in this examination is probably due to the fact that, since it was required to report the number of deadlocks instead of just the detection of at least one, tools must investigate the full state space, thus making this examination behave like the state space generation.

## 7 Observations on Reachability Formulas

The data collected for evaluation of satisfiable and unsatisfiable reachability formulae is summarized in Table 6. It must be read as Table 4.

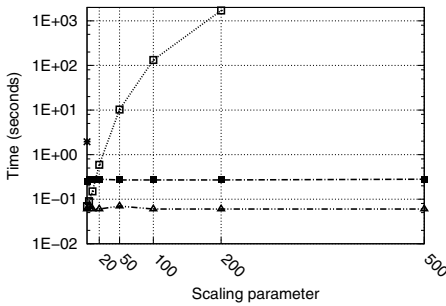
From Table 6, we can clearly state that for Place/Transition nets, there are tools that perform extremely well for satisfiable formulae (**LoLA** and **Sara**) where some others are much better for unsatisfiable formulae (**PeTe** and **Sara**). In that context, decision diagram based tools can perform well (see **ITS-Tools**), but do not reach the maximum values. The reason why **LoLA** does not reach a very high parameter on Kanban is still not understood. **Sara** is clearly interesting for Place/Transition nets, as it reaches the maximum scaling value for both satisfiable and unsatisfiable formulae.

The formulae to verify were only conjunctions of place markings. The efficiency of tools may depend on the operators used in properties. For instance, formulae with disjunctions and inequalities can lead to worse results in **Sara**. The 2012 Model Checking Contest should thus provide more different formulae, and also describe their properties, to get a detailed analysis of the tools' performances. Note also that **Sara** gives an answer very quickly, as seen for FMS in Figure 7.

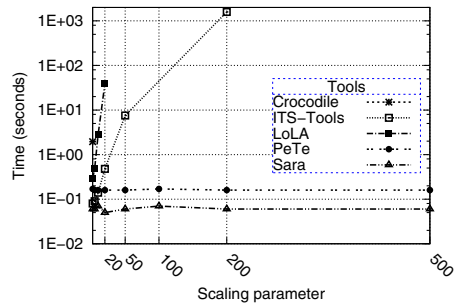
Tools that perform well on satisfiable formulae for P/T nets are **LoLA** and **Sara**. These tools explore the state space and stop their execution as soon as they

**Table 6.** Results for the reachability formulæ examination

		ACTIVITY-LOCAL	ALPINA	Crocodile	ITS-Tools	LoLA	PNXDD	PeTe	Sara	YASPA	heLena
<b>Satisfiable reachability formulæ</b>											
P/T	FMS			2	200	500		500			
	Kanban				200	200		1 000			
	MAPK				160	320		80			
Colored	Peterson			5	5 000	10					
	Philosophers				20 000						
	SharedMemory				40						
	TokenRing										
<b>Unsatisfiable reachability formulæ</b>											
P/T	FMS			2	50	20		500	500		
	Kanban				200	10		1 000	1 000		
	MAPK				160	20		320	320		
Colored	Peterson			5	5 000	10					
	Philosophers				20 000						
	SharedMemory				50						
	TokenRing										



(a) Satisfiable formulæ



(b) Unsatisfiable formulæ

**Fig. 7.** Evolution of CPU consumption for reachability properties on FMS

have found a violation of the property to be verified. Thus, since a satisfiable formula is verified before the full state space is explored, they perform better on satisfiable formulæ than on unsatisfiable ones.

Tools that perform well on unsatisfiable formulæ for P/T nets are **PeTe** and **Sara**. To do so, they first evaluate the state equation of the P/T net against the reachability formula. If the result of such an evaluation shows the formula is structurally unverifiable, the tool does not need to explore, even partially, the state space. Otherwise, exploration to extract a counter-example is necessary.

**Sara** combines the two techniques and is thus quite efficient in both cases. This effect is illustrated in Figure 7 that is representative of the behavior of tools for P/Ts.

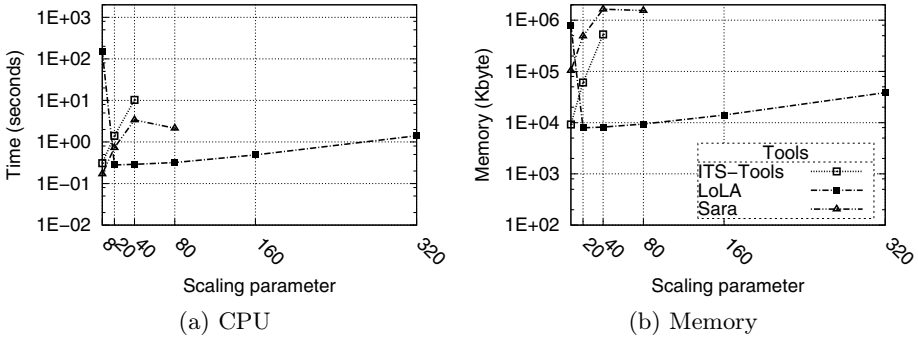


Fig. 8. Evolution of CPU consumption satisfiable formulæ (MAPK)

Figure 8 illustrates an interesting fact on partial order with LoLA on the MAPK benchmark (both CPU and memory). For satisfiable formulæ, both memory and CPU performances are much better for MAPK<sub>320</sub> than for MAPK<sub>8</sub>. This is due to the less parallel nature of MAPK for small scaling values, thus degrading performances of partial order techniques. When the scaling parameter grows, parallelism is generated and the technique becomes useful.

The data for Colored net models permit similar observations as for state space generation, because only decision diagram based tools participated, except LoLA for the Philosophers model. We can observe that LoLA has difficulties to scale up with this model. This may be because when the parameter grows, this model has more places, whereas P/T models only have more tokens. However, if we can state that decision diagrams are a good technique for reachability analysis, the collected data are not sufficient to generalize this assertion to reachability properties.

## 8 Discussion

This section proposes a global discussion on the Model Checking Contest'2011 results from several points of view. First, we focus on the user (*i.e.*, an engineer willing to verify software) point of view in Section 8.1. Then, Section 8.2 considers the tool developer point of view. Last, Section 8.3 recalls the lessons learned by the Model Checking Contest organizers for the next edition.

### 8.1 Engineer Point of View: There Is No Silver Bullet

Models, like software, have a lifecycle. It can be roughly decomposed into: its creation, its verification, and its evolution. However, the evolution phase is a sequence of editions and verifications. Thus, the model lifecycle can be simplified in a sequence of edition and of verification. During each phase, properties are checked on the model. But the kind of properties may vary.

We have used two kinds of properties in this Model Checking Contest: deadlock detection and reachability properties. For both, we can also distinguish the case where the property is satisfiable, and the case where it is not.

**For Place/Transition Net Models.** During the model development, we want to debug the model. To do so, the chosen properties must be checked quickly. Except the case when **ITS-Tools** can use recursive folding, all tools spend at least 10 seconds to answer (for very small configurations) ; the answering time grows rapidly. Thus, deadlock detection is currently not efficient enough to debug large models. Instead, tools like **PeTe**, **LoLA** and **Sara** can answer in less than 1 second for reachability properties. **Sara** gives a quick answer whether the property is satisfiable or not, whereas **LoLA** is more efficient when the property is satisfiable, and **PeTe** when it is not. A good idea would be to run both **LoLA** and **PeTe** in parallel and stop them as soon as one of them answers. Reachability properties could even be checked in background, while editing the model. Then, for users, model creation would be very close to source code edition in modern IDEs, that make use of continuous on-the-fly compilation.

During the model verification phase, all tools can be used, as there is time to do longer checks. Deadlock detection is currently adapted to this phase, as it is a rather long process. For Place/Transition nets, there is no added value in state space generation, as reachability properties can be checked during the edition phase.

For deadlocks, **helena** can be impressively efficient, for the FMS model, or not as good as decision diagram based tools. There should be some investigations on why. Also, **ITS-Tools** shows impressive performance when models can be “folded”. Hierarchical extensions of Petri nets are clearly interesting.

**For Colored Net Models.** Decision diagram based tools are very efficient for state space computation. Using this state space they are then able to find deadlocks and check properties. But state space computation is usually quick only for small models. As most tools that do not use decision diagrams did not participate for Colored net models, we cannot conclude yet about which techniques should be used.

---

|| We must provide P/T equivalents for all Colored models in the next Model Checking Contest, so that more tools can compete.

## 8.2 Lessons Learned by Tool Developers

The Model Checking Contest can help tool developers to discover some unexpected behaviors and compare strategies and techniques among the various participating tools in common situations.

As an illustration, **ALPiNA** developers discovered that it is currently mostly adapted to safe Petri nets. The tool got bad results on all the Place/Transition nets of the contest. Analysis revealed that **ALPiNA** inefficiency on non-safe nets is due to the particular decision diagrams used in the tool.

The contest is also a good way to test the integration of model checkers in an “alien” environment. This can be a basis to extend cooperation and exchange of data between model checkers and promote further cooperation.

As an illustration, LoLA follows several UNIX principles. This made an integration to the contest scripts very smooth. First, for each model, there was a dedicated compiled version of LoLA that can exploit any possible optimization the CPU architecture could offer. Second, “UNIX pipelines” made the evaluation of the reachability results very simple (use of `grep` to filter outputs).

### 8.3 Points Risen by the Discussion at MCC’2011

Several points were raised during the discussion held during the MCC’2011 in Newcastle. We present here the most interesting ones.

**A Difficult Model: Peterson.** One point was outlined in the Model Checking Contest: the Peterson model seems reluctant to all the implemented techniques. The best processed scaling values are 5 (`PNXDD`) for the state space generation and 3 (`helena`, `ALPiNA`) for deadlock detection. This corresponds to very small state spaces compared to the ones reached for other models. This exhibits an interesting situation to be handled by tools.

**Need for a “Push-Button” Examination.** As it is organized, the Model Checking Contest is efficient to identify how some model characteristics could be tackled by some model checking techniques. However, this does not cover the use of model checkers by non-specialists. For this kind of users there should be a efficient “push-button” use of such tools. This aspect should be considered in further editions of the Model Checking Contest. An idea should be to find “surprise models” from case studies, that are not known by the competitors when they submit (and only published when results are known).

**Doing CPU and Memory Measures Is Tricky.** Measuring and confining software executions during this first Model Checking Contest was not trivial. Tools are written in several languages, some of which are based on shell scripts, interpreters or virtual machines. Moreover, tools are allowed to create subprocesses and catch signals. To avoid most problems while not being intrusive, we plan to execute tools within a virtual machine monitored to operate time and memory measures.

## 9 Conclusion

This paper reported our experience with the first Model Checking Contest @ Petri nets. This event and its results were welcomed by the Petri Net Community, as the discussion held at a special session of SUMo’2011 showed.

From the tool developers’ point of view, such an event allows to compare tools on a common benchmark that could become a public repository. Also,

some mechanisms established for the contest, such as a language to elaborate the formula to be verified could become, over the years, a common way to provide formulæ to the various tools developed by the community.

Results also provided hints to the tool developers with regards to the optimization of some techniques in their tool. At least, developers of **ALPiNA** and **Crocodile** attest that some development is being currently done to improve the model checking engine from the results of the model checking contest. This will benefit to the entire community.

From the organizer's point of view, numerous lessons were learned on the process, the analysis of results and the selection of benchmark models. Several points will be integrated in further edition of the Model Checking Contest.

As an illustration, the next edition to be held in 2012 comes with a new step in the process: a call for models that will allow us to gather more models, exposing tools to a larger range of situations. Properties will be extended to CTL and LTL formulas, as well as with structural properties, such as bounds or liveness, and their counterpart in temporal logic. Finally, a "blind" set of models will also be proposed to reproduce a situation where tools are used "as is" by non specialists (and thus with default optimization activated only).

**Acknowledgements.** The Model Checking Contest @ Petri nets organizers would like to thank the following people for the help they provided in setting up this event: Clément Démoulin (infrastructure), Nicolas Gibelin (infrastructure and cluster management), Lom Hillah (production of PNML files), Emmanuel Paviot-Adet and Alexis Marechal (description of selected models), and Steve Hostettler (definition of properties).

The Model Checking Contest organizers would also like to thank the tool developers who made possible such a contest. They are:

- **ACTIVITY-LOCAL** and **YASPA**: Kai Lampka;
- **ALPiNA**: Steve Hostettler, Alexis Marechal, and Edmundo Lopez;
- **Crocodile**: Maximilien Colange;
- **ITS-Tools**: Yann Thierry-Mieg;
- **LoLA**: Karsten Wolf;
- **PeTe**: Jonas Finnemann, Thomas Nielsen and Lars Kærlund;
- **PNXDD**: Silien Hong and Emmanuel Paviot-Adet;
- **Sara**: Harro Wimmel and Karsten Wolf;
- **heleNA**: Sami Evangelista and Jean-François Pradat-Peyre.

## References

1. Aloul, F.A., Markov, I.L., Sakallah, K.A.: FORCE: a fast and easy-to-implement variable-ordering heuristic. In: ACM Great Lakes Symposium on VLSI, pp. 116–119. ACM (2003)
2. Berthelot, G.: Transformations and Decompositions of Nets. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 254, pp. 359–376. Springer, Heidelberg (1987)

3. Buchs, D., Hostettler, S.: Sigma Decision Diagrams. In: Preliminary Proceedings of the 5th International Workshop on Computing with Terms and Graphs, pp. 18–32. No. TR-09-05 in TERMGRAPH workshops, Università di Pisa (2009)
4. Buchs, D., Hostettler, S., Marechal, A., Risoldi, M.: AIPiNA: An Algebraic Petri Net Analyzer. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 349–352. Springer, Heidelberg (2010)
5. Burch, J., Clarke, E., Long, D.: Symbolic Model Checking with Partitioned Transition Relations. In: Halaas, A., Denyer, P.B. (eds.) International Conference on Very Large Scale Integration, pp. 49–58. North-Holland, Edinburgh (1991)
6. Chiola, G., Dutheillet, C., Franceschinis, G., Haddad, S.: On well-formed coloured nets and their symbolic reachability graph. In: Jensen, K., Rozenberg, G. (eds.) Proceedings of the 11th International Conference on Application and Theory of Petri Nets, ICATPN 1990. Reprinted in High-Level Petri Nets, Theory and Application. Springer (1991)
7. Chiola, G., Franceschinis, G.: Colored GSPN Models and Automatic Symmetry Detection. In: The Proceedings of the Third International Workshop on Petri Nets and Performance Models, PNPM 1989, pp. 50–60. IEEE Computer Society (1989)
8. Ciardo, G.: Advances in compositional approaches based on kronecker algebra: Application to the study of manufacturing systems. In: 3rd International Workshop on Performability Modeling of Computer and Communication Systems, pp. 61–65 (1996)
9. Ciardo, G., Trivedi, K.: A decomposition approach for stochastic reward net models. *Perf. Eval.* 18, 37–59 (1993)
10. Ciardo, G., Lüttgen, G., Siminiceanu, R.: Efficient Symbolic State-Space Construction for Asynchronous Systems. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 103–122. Springer, Heidelberg (2000)
11. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
12. Colange, M., Baair, S., Kordon, F., Thierry-Mieg, Y.: Crocodile: A Symbolic/Symbolic Tool for the Analysis of Symmetric Nets with Bag. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 338–347. Springer, Heidelberg (2011)
13. Couvreur, J.-M., Thierry-Mieg, Y.: Hierarchical Decision Diagrams to Exploit Model Structure. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 443–457. Springer, Heidelberg (2005)
14. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. *Commun. ACM* 17(11), 643–644 (1974)
15. Esparza, J., Melzer, S.: Verification of safety properties using integer programming: Beyond the state equation. *Formal Methods in System Design* 16, 159–189 (2000)
16. Evangelista, S.: High Level Petri Nets Analysis with Helena. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 455–464. Springer, Heidelberg (2005)
17. Finnemann, J., Nielsen, T., Kærland, L.: Petri nets with discrete variables. Tech. rep. (2011), <http://jopsen.dk/downloads/PetriNetsWithDiscreteVariables.pdf>
18. Haddad, S., Kordon, F., Petrucci, L., Pradat-Peyre, J.F., Trèves, N.: Efficient State-Based Analysis by Introducing Bags in Petri Net Color Domains. In: 28th American Control Conference, ACC 2009, pp. 5018–5025. Omnipress IEEE, St-Louis (2009)



19. Hamez, A., Thierry-Mieg, Y., Kordon, F.: Building efficient model checkers using hierarchical set decision diagrams and automatic saturation. *Fundamenta Informaticae* 94(3-4), 413–437 (2009)
20. Heiner, M., Gilbert, D., Donaldson, R.: Petri Nets for Systems and Synthetic Biology. In: Bernardo, M., Degano, P., Zavattaro, G. (eds.) SFM 2008. LNCS, vol. 5016, pp. 215–264. Springer, Heidelberg (2008)
21. Hong, S., Kordon, F., Paviot-Adet, E., Evangelista, S.: Computing a Hierarchical Static Order for Decision Diagram-Based Representation from P/T Nets. In: Jensen, K., Donatelli, S., Kleijn, J. (eds.) ToPNoC V. LNCS, vol. 6900, pp. 121–140. Springer, Heidelberg (2012)
22. Hostettler, S., Linard, A., Marechal, A., Risoldi, M.: Improving the significance of benchmarks for petri nets model checkers. In: 1st International Workshop on Scalable and Usable Model Checking for Petri Nets and Other Models of Concurrency, pp. 97–111 (2010)
23. Hostettler, S., Marechal, A., Linard, A., Risoldi, M., Buchs, D.: High-Level Petri Net Model Checking with AIPiNA. *Fundamenta Informaticae* 113 (2011)
24. Klebanov, V., Müller, P., Shankar, N., Leavens, G.T., Wüstholtz, V., Alkassar, E., Arthan, R., Bronish, D., Chapman, R., Cohen, E., Hillebrand, M., Jacobs, B., Leino, K.R.M., Monahan, R., Piessens, F., Polikarpova, N., Ridge, T., Smans, J., Tobies, S., Tuerk, T., Ulbrich, M., Weiß, B.: The 1st Verified Software Competition: Experience Report. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 154–168. Springer, Heidelberg (2011)
25. Kordon, F., Linard, A., Paviot-Adet, E.: Optimized Colored Nets Unfolding. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 339–355. Springer, Heidelberg (2006)
26. Kristensen, L.M., Schmidt, K., Valmari, A.: Question-guided Stubborn Set Methods for State Properties. *Formal Methods in System Design* 29(3), 215–251 (2006)
27. Kristensen, L.M., Valmari, A.: Improved Question-Guided Stubborn Set Methods for State Properties. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 282–302. Springer, Heidelberg (2000)
28. Lampka, K.: A new algorithm for partitioned symbolic reachability analysis. In: Workshop on Reachability Problems. ENTCS, vol. 223 (2008)
29. Lampka, K., Siegle, M.: Activity-Local State Graph Generation for High-Level Stochastic Models. In: Measuring, Modelling, and Evaluation of Systems 2006, pp. 245–264 (2006)
30. Lampka, K.: A new algorithm for partitioned symbolic reachability analysis. *Electron. Notes Theor. Comput. Sci.* 223, 137–151 (2008)
31. Lampka, K., Siegle, M., Ossowski, J., Baier, C.: Partially-shared zero-suppressed multi-terminal bdds: concept, algorithms and applications. *Formal Methods in System Design* 36, 198–222 (2010)
32. Linard, A., Paviot-Adet, E., Kordon, F., Buchs, D., Charron, S.: polyDD: Towards a Framework Generalizing Decision Diagrams. In: 10th International Conference on Application of Concurrency to System Design, ACS D 2010, pp. 124–133. IEEE Computer Society, Braga (2010)
33. Minato, S.: Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In: Proc. of the 30th Design Automation Conference, DAC, pp. 272–277. ACM/IEEE, Dallas (Texas), USA (1993)
34. Pastor, E., Roig, O., Cortadella, J.: Symbolic Petri Net Analysis using Boolean Manipulation, Technical Report of Departament Arquitectura de Computadors (UPC) DAC/UPC Report No. 97/8 (1997)

35. Peterson, G.L.: Myths about the mutual exclusion problem. *Inf. Process. Lett.* 12(3), 115–116 (1981)
36. Schmidt, K.: Narrowing petri net state spaces using the state equation. *Fundamenta Informaticae* 47(3-4), 325–335 (2001)
37. Schmidt, K.: Stubborn Sets for Standard Properties. In: Donatelli, S., Kleijn, J. (eds.) ICATPN 1999. LNCS, vol. 1639, pp. 46–65. Springer, Heidelberg (1999)
38. Schmidt, K.: Using Petri Net Invariants in State Space Construction. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 473–488. Springer, Heidelberg (2003)
39. Thierry-Mieg, Y., Ilić, J.-M., Poitrenaud, D.: A Symbolic Symbolic State Space Representation. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 276–291. Springer, Heidelberg (2004)
40. Thierry-Mieg, Y., Poitrenaud, D., Hamez, A., Kordon, F.: Hierarchical Set Decision Diagrams and Regular Models. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 1–15. Springer, Heidelberg (2009)
41. Wikipedia: Dining philosophers problem (2011), [http://en.wikipedia.org/wiki/Dining\\_philosophers\\_problem](http://en.wikipedia.org/wiki/Dining_philosophers_problem)
42. Wimmel, H., Wolf, K.: Applying CEGAR to the Petri Net State Equation. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 224–238. Springer, Heidelberg (2011)
43. Wolf, K.: Generating Petri Net State Spaces. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 29–42. Springer, Heidelberg (2007)

# Modelling and Formal Verification of the NEO Protocol\*

Christine Choppy, Anna Dedova, Sami Evangelista, Kais Klai,  
Laure Petrucci, and Samir Youcef

Université Paris 13, Sorbonne Paris Cité,  
LIPN, CNRS, UMR 7030, F-93430, Villetaneuse, France

**Abstract.** In order to manage very large distributed databases such as those used for banking and e-government applications, and thus to handle sensitive data, an original peer-to-peer transaction protocol, called NEO, was proposed. To ensure its effective operation, it is necessary to check a set of critical properties. The most important ones are related to availability of data that must be guaranteed by the system. Thus, our objective aims at verifying critical properties of the NEO protocol so as to guarantee such properties are satisfied. The model is obtained by reverse-engineering from the source code and then formal verification is performed. We focus in this article on the two phases of the NEO protocol occurring at the initialisation of the system. The first one, the *election phase*, aims at designating a special node that will pilot the overall system. The *bootstrap protocol*, triggered at the end of the election, ensures that the system will enter its operational state in a coherent way. Therefore, the correctness of these two phases is mandatory for the reliability of the system.

## 1 Introduction

Nowadays, several complex software are developed to manage increasingly huge distributed databases like those used for e-government, Internet based information systems or commerce registries applications. The challenge with such software is to guarantee the access to these databases, maintain them and ensure a mandatory high level of reliability. Moreover, the development of such applications is a crucial problem which requires to elaborate reliable and safe distributed database management software. Therefore, it is necessary to use formal methods to specify the behaviour of such applications and to develop tools to automatically check whether this behaviour satisfies the desired properties.

The *Zope Object Database* (ZODB) [3] is a popular object database which is included as part of the Zope web application server. It is best known for its use for a Central Bank, to manage the monetary of 80 million people in 8 countries [8]. It is also known for its use for accounting, ERP (Enterprise Resource Planning), CRM (Customer Relationship Management), ECM (Enterprise Content Management) and knowledge management.

---

\* This work is supported by FEDER Île-de-France/ System@tic—libre software.

However, the current Zope architecture does not apply for huge data collections yet. In order to overcome this limitation, a new peer-to-peer transaction protocol, called NEO, was designed. This protocol also enjoys ensuring both safety and reliability, which is not easy to achieve for distributed systems using traditional testing techniques. For this, the NEO protocol is based on the following principles: a decentralised peer-to-peer architecture, a duplication of data on different storage nodes, and fault tolerance. Thus, the NEO protocol is a complex architecture implementing various protocol mechanisms where the verification becomes a crucial problem. A description of the context and the general functioning of the NEO protocol can be found in [4].

We distinguish two important phases in the NEO protocol execution, namely the election phase and the bootstrap phase. In order to designate a *primary master* that will pilot the overall system, the election phase is first triggered when the cluster is started. After this phase, the bootstrap protocol is initiated. The specification and verification are conducted, focused on the election and bootstrap phases, the master nodes among which the primary master is designated, and the storage nodes the database is distributed on.

The objective of our work is to analyse critical properties of the NEO protocol so as to guarantee that such properties are satisfied. The model construction is achieved by reverse-engineering, extracting coloured Petri net [13] models from the source code, and then verification is performed. In this paper, we focus on the modelling as well as the verification of properties. This specification work requires choices of adequate abstraction levels both for the modelling and the verification stages. We revise the work presented in [6] on the election phase and extend it to the bootstrap protocol. For the election phase, the following properties are studied: (i) one and only one primary master is elected, (ii) all nodes know the primary master's identity and (iii) the election phase eventually terminates. The following critical properties, regarding the bootstrap protocol, are addressed: (i) all storage nodes eventually reach the final state, (ii) for any system partition, there exists at least one storage node which contains the objects of this partition<sup>1</sup> and (iii) at the end of the protocol, there is no empty storage node (i.e. with no associated partition). More details of these properties are given in Sections 4 and 5, respectively. Various tools have been used in this project. For modelling, we used Coloane [1] and CPN Tools [14], and for verification Helena [9] and CPN Tools.

The rest of the paper is organised as follows. Section 2 recalls the general functioning of the NEO protocol. In Section 3, we present the tools we used in order to model and to analyse the election and the bootstrap phases of the protocol. Section 4 presents the modelling and the formal analysis of the election phase. Section 5 presents the bootstrap protocol model for which an analysis of the desired properties is also explained. Finally, Section 6 concludes the paper and gives some perspectives to this work.

---

<sup>1</sup> For the sake of readability, an element of the partition table is called a partition hereafter (by abuse of language).

## 2 The Neo System

This section informally describes the general functioning of the NEO system protocol implemented in Python. We first introduce the different kinds of network nodes involved before detailing the different stages the system can go through. The phases modelled in this article will be explained in greater details in Sections 4 and 5.

### 2.1 Participating Nodes

Different kinds of nodes play dedicated roles in the protocol, as depicted by the architecture in Figure 1:

**storage nodes** handle the database itself. Since the database is distributed, each storage node cares for a part of the database, according to a *partition table*. To avoid data loss in case of a node failure, data is duplicated, and is thus handled by at least two storage nodes.

**master nodes** handle the transactions requested by the client nodes and forward them to the appropriate storage nodes. A distinguished master node, called *primary master*, handles the operations. *Secondary masters* (i.e. the other master nodes) are ready to become primary master in case of a failure of this node. They also inform other nodes of the identity of the primary master (light grey arrows in Figure 1).

**the administration node** is used for manual setup if needed (dashed arrow in Figure 1).

**client nodes** correspond to the machines running applications concerned with the database objects. Thus, they request either read or write operations. They first ask the primary master which storage nodes are concerned with their data, and can then contact them directly.

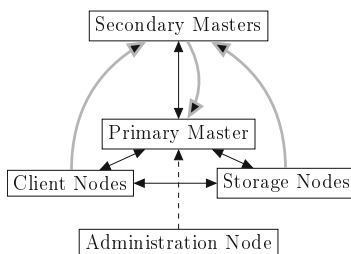


Fig. 1. The NEO-protocol topology

### 2.2 Lifecycle of the NEO System

At the system startup, the primary master is *elected* among all master nodes. The primary master maintains the key information for the protocol to operate. Among these, the *partition table* indicates which parts of the database are assigned to the different storage nodes. This allows for duplication which is vital in case of a crash.

After the election of a primary master, the system goes through various stages with the purpose of checking that all transactions were completely processed, and

thus that the database is consistent across the different storage nodes. We will refer to this second step as the *bootstrap* protocol.

Finally, the system enters its *operational state*. Clients can then access the database through the elected primary master.

This repartition of roles raises several issues. The system topology is in constant evolution: nodes can fail and become unavailable, they can restart, or new nodes can be added. The failure of the primary master has dire consequences since it affects the whole system. A new election must then take place among the remaining secondary masters and the whole process starts again.

We focus in the next sections on the first two stages of this cycle that are vital for the reliability of the system and the consistency of the database.

### 3 Tools

There now exists a profusion of state space analysis tools based on the Petri nets formalism. In March 2011, the Petri net tools database [16] reported about thirty Petri net tools able to perform model checking. Each is characterised by the family of nets it supports (e.g. place-transition nets, coloured nets, algebraic nets), the algorithms it employs (e.g. explicit vs. symbolic states), the state space reduction techniques it implements (e.g. symmetry, partial order reductions), or the kind of properties it can analyse (e.g. safety, liveness). Therefore, choosing the adequate tool in a verification project is a very difficult task that can require some expertise in the underlying principles of the tool. Rather than favouring a single tool we actually picked out several ones during this project. We give below an overview of these tools as well as the reasons that motivated our choices. In Section 3.2 we present the new composition tool we built so as to broaden the interface facilities between the modelling and the verification tools we use.

#### 3.1 Tools Used in the Context of the Neoppod Project

Four Petri net tools have been used so far in this project.

CPN-AMI [11] is a verification platform with structural analysis and model checking facilities provided through different dedicated tools (e.g. Great-SPN [2], Prod [17]).

Helena [9] is a high-level Petri net model checker that provides a high-level language for net description and several state space reduction techniques (e.g. static net reductions, partial order reduction [7]).

Coloane [1] is not stricto sensu a Petri net tool but a generic graphical editor available as an Eclipse plugin. Coloane can produce nets in CPN-AMI and Helena input formats. A composition tool for Coloane has been implemented in the context of this project to facilitate our analysis. This tool will be described in Section 3.2.

CPN Tools [14] is famous for its nice graphical interface, its high-level language, based on SML, and its support of hierarchy [12] allowing the user for creating nets in a modular way and with different abstraction levels.

The first three tools have been used during the analysis of the election protocol while the bootstrap protocol has been modelled with CPN Tools. This choice was mainly motivated by the complexity of the data structures handled during these two steps. The election protocol relies on relatively simple structures (e.g. lists) while the bootstrap protocol makes use of more elaborate ones (e.g. the partition table) that were hard to abstract away without losing too much interesting information. Therefore, we considered relevant to use CPN Tools for the analysis of the bootstrap protocol to benefit from its rich language even though this tool does not offer the same verification capabilities as CPN-AMI or Helena. At last, two reasons drove us for using both CPN-AMI and Helena for the election protocol. First, these two tools do not implement the same algorithms and reduction techniques. Second, the description language of Helena is richer than that of symmetric nets [5] that is the underlying language of the CPN-AMI platform. Therefore, Helena allowed us to keep a model closer to the protocol concepts (as regards data types).

### 3.2 A New XML-Based Composition Tool

Although Coloane has a nice Eclipse based interface it still suffers from a drawback in that it does not support any kind of hierarchy or modularity. Thus, we chose to develop a composition tool that, given a set of XML Coloane files and an XML file (provided by the user) describing a composition scheme, produces the flattened net resulting from the composition and that can be used as input to verification tools, e.g. Helena.

This tool supports various kinds of transformations inspired from [12] such as the *place fusion* merging instances of the same place located in different nets, or the *transition substitution* that replaces an abstract transition with a given subnet describing the actual behaviour of the transition. However, the tool is still at a prototype stage and some issues have not been tackled yet. For instance, places can be fused in a bad way, and no guarantee can be made on the correctness of the output net: this has to be made by the model designer.

To illustrate the essence of the tool we provide in Figure 2 a sample of the composition file written during the analysis of the election protocol. Starting from a set of subnets (declarations in lines ll. 4–11) each describing a module of the final net, the composition tool performs operations written in lines ll. 13–22. The first one (lines ll. 13–15) substitutes abstract transition `poll` by the homonym net in net `electPrimary`. The last operation to be performed (line l. 22) merges all places sharing the same name.

The tool provides some flexibility since some modules or the application of some operations may be conditioned by the definition (or non-definition) of symbols (see e.g. operations at lines l. 16 or l. 19 applied only if symbol `faults` is not defined) at the tool invocation. Thus, the system modeller does not necessarily have to change the net when analysing different configurations, as it may be sufficient to call the tool with the appropriate symbols. Finally, let us point out that this tool is totally independent from the language used for arc inscriptions: we used it for both symmetric nets and Helena nets.

```

1 <netcomposition>
2 <!-- definition of some nets -->
3 ...
4 <subnet id="electPrimary"><fromfile>electPrim.xml</fromfile></subnet>
5 <subnet id="poll"><fromnet>poll</fromnet></subnet>
6 <subnet id="secPoll"><fromnet>secpoll</fromnet></subnet>
7 <subnet id="primPoll"><fromnet>primpoll</fromnet></subnet>
8 <subnet id="sendAnnPs"><fromfile>sendAnnPs.xml</fromfile></subnet>
9 <subnet id="sendAskPs"><fromfile>sendAskPs.xml</fromfile></subnet>
10 <subnet id="crash" ifdef="faults"><fromnet>crash</fromnet></subnet>
11 <subnet id="reboot" ifdef="faults"><fromnet>reboot</fromnet></subnet>
12 <!-- operations to perform -->
13 <substitutetrans>
14 <net>electPrimary</net><trans>poll</trans><subnet>poll</subnet>
15 </substitutetrans>
16 <deletetrans ifdef="faults">
17 <net>electPrimary</net><trans>crash</trans>
18 </deletetrans>
19 <deletetrans ifdef="faults">
20 <net>electPrimary</net><trans>primCrash</trans>
21 </deletetrans>
22 <fusehomonymplaces/>
23 </netcomposition>

```

**Fig. 2.** A sample of the composition file used for the election protocol model

## 4 Formal Analysis of the Election Protocol

Due to the critical aspect of the election protocol, we developed a detailed model of this phase to be able to simulate it and perform state space analysis. Since the protocol is designed to be (to some extent) fault tolerant, we proceeded by injecting faults in a model designed on the basis of the ideal scenario where no fault (e.g. a master node failure, a connection loss) can occur. We describe in this section the modelling and analysis process we followed. As mentioned in Section 3, we extracted both symmetric and Helena nets from the election protocol. However, due to space constraints we focus in this section on the symmetric net, and we provide a sample of the Helena net in Section 4.5.

### 4.1 Overview of the Election Protocol and Its Implementation

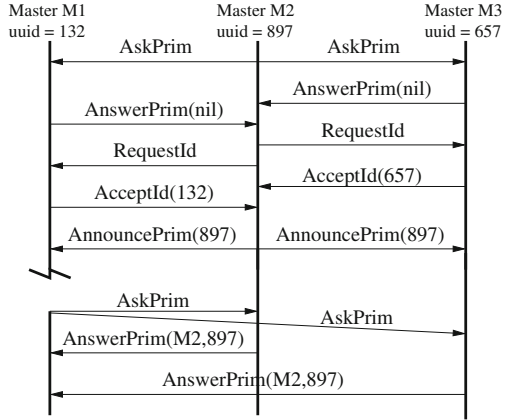
The goal of the election is to select among all alive masters the one with the greatest *uuid*, a unique identifier chosen randomly by each node at its startup.

The election proceeds in two steps: a **negotiation** step performed by a master node to discover if it is the primary master or not; followed by an **announcement** step during which all masters discover the identity of the primary master and check for its liveness.

The functioning of this protocol is illustrated by the message sequence chart of Figure 3 that describes a typical election scenario between three master nodes M1, M2 and M3. We only depict the message exchanges from the perspective of master M2 which is elected as the primary master. Of course masters M1 and M3 also have to ask for the same information. Initially, a master node only knows the network addresses (IP address + port number) of its peers provided to it through



a configuration file. It learns the uuids of all its peers during the negotiation step. First it asks the other nodes if they know a primary master by broadcasting an `AskPrim` message. Other masters answer with an `AnswerPrim` message possibly containing the uuid and the network address of the elected primary master. In our example, these messages are empty since the election is still taking place. The purpose of this first exchange is mainly related to fault tolerance as explained below. Upon reception of the `AnswerPrim` message, the master asks its peer its uuid by sending a `RequestId` message to it. The answer to this message is an `AcceptId` containing the uuid of the contacted node. This process ends when the master has negotiated with all other master nodes, i.e. it knows the uuid of all its peers. A master node which did not receive any `AcceptId` message with a uuid greater than its own knows it is itself the primary master.



**Fig. 3.** Message sequence chart describing an election scenario followed by a crash and reboot of master M1

During the announcement step, the primary master announces to its peers that it is actually the primary master by broadcasting an `AnnouncePrim` message containing its uuid. Secondary masters wait for this message that they interpret as a confirmation of the existence of an alive primary master. All masters can then exit the election protocol.

In case a master experiences a fault (e.g. master M1 in our example) it asks, after its reboot, the identity of the primary master by broadcasting an `askPrim` message. The `answerPrim` messages it will then receive will contain the identity of this master and the awakened node will enter the secondary master state.

A message of type `ReelectPrim` may also be sent by a master if it detects a problem during the election, e.g. two primary masters have been designated. Upon its reception, a master will cancel its current work, and restart the election process from the beginning. In a faultless scenario this situation should however not occur.

This example also highlights the fact that entering the election phase is a local decision made by a master at its startup or if it considers the primary as crashed. Thus some master(s) may be in the election mode while others are executing the normal protocol.

The implementation of the election protocol relies on a few data structures. The most important ones are two sets belonging to the `Master` thread class identifying, for a specific master `m`, all its peers it is not connected to and has to

do so (`m.unconnected`), or is negotiating with (`m.negotiating`). The termination of the negotiation step is conditioned by the emptiness of these two sets: at that point, the node has contacted all its peers and received all their uuids. To have a better understanding of the contents of these sets, it seems necessary to mention the different events that have an impact on these two sets:

- initially, `m` puts in the `m.unconnected` set all masters it considers as alive ;
- when a connection of `m` is accepted by `n`, `m` moves `n` from `m.unconnected` to `m.negotiating` ;
- as `m` receives an `AcceptId` message from `n` it discards `n` from `m.negotiating` ;
- finally if `m` detects the crash of master `n` it deletes `n` from both sets.

## 4.2 Model Architecture

The model consists of 18 modules, each of them modelling a specific part of the code. Among them, the most important ones are the three modules listed below.

**electPrimary** models the main method implementing the election protocol.

**poll** models the polling method used to wait for and handle incoming packets.

**electionFailure** models the handling of an exception raised when some synchronisation fault is detected. The master raising this exception stops the current election process and triggers a new one.

In some figures, there are abstract (or meta-) transitions (transitions `secPoll`, `poll`, `sendAskPs`, `sendAnnPs`, `primaryPoll` in Figure 6, all transitions except die in Figure 12(a), and all transitions in Figure 13(b)) that are then substituted by our composition tool with the appropriate concrete net (or subnet). Such subnets always have two transitions `start` and `end` corresponding to the start and the end of the activity. Guards are put in small notes linked to the corresponding transition (see Figure 9). Some arc labels, markings or guards depend on the parameters of our model although they are automatically generated by a pre-processing of the net. The number of masters was set to 2 in the configuration used for this paper. As usual, all instances of places with the same name are merged.

## 4.3 Detailed Specification of Some Key Elements

**General Declarations.** Fig. 4 gives the main colour classes we use for modelling the election protocol. Class `M` ranging from 0 to `MN` (the number of master nodes) is used to identify masters, with constant 0 specifying a null value<sup>2</sup>.

The message values (e.g. `AskP`, `RI`) of the `MSG_TYPE` class correspond to the messages (e.g. `AskPrim`, `RequestId`) introduced in Section 4.1 and Figure 3. Finally, items of class `NEG` specify the state of a negotiation between master `m` and one of its peers `p`:

<sup>2</sup> Note that we do not distinguish in our model the uuid from the network address.

It may however be worth modelling, in a future version, situations where a master reboots and is assigned a greater new uuid, as it may impact the current election process.

```

1 parameter
2   MN = 2;
3 class
4   BOOL      is [F, T];
5   M         is 0 .. MN;
6   MSG_TYPE is [AskP, AnsP, RI, AI, AnnP, RP];
7   NEG       is [NONE, CO, DONE];

```

Fig. 4. Colour classes of the election model

NONE means that  $p$  has not been contacted:  $p \in m.unconnected$ .  
 CO means that  $m$  has contacted  $p$  and is waiting for its uuid:  $p \in m.negotiating$ .  
 DONE means that  $m$  knows the uuid of  $p$ :  $p \notin m.negotiating \cup m.unconnected$ .

Figure 5 represents some places shared by all modules of our net together with their colour class and initial marking. Place `masterState` models the current knowledge that any master  $m$  has of the primary master. An invariant property states that for any  $m \in 1..MN$  there is a unique token  $\langle m, iam, pm \rangle$  in this place, where  $iam=F$  means “I am not the primary master” and  $iam=T$  means “I am the primary master or I do not know a primary master yet”, and  $pm$  is the uuid of the primary master (or 0 if it is not known yet).







	Color domain	Initial marking
 <code>masterState</code>	$\langle M, BOOL, M \rangle$	$\langle 1, T, 0 \rangle + \langle 2, T, 0 \rangle$
 <code>negotiation</code>	$\langle M, M, NEG \rangle$	$\langle 1, 2, NONE \rangle + \langle 2, 1, NONE \rangle$
 <code>network</code>	$\langle M, M, MSG\_TYPE, M \rangle$	
 <code>electionInit</code>	$\langle M \rangle$	$\langle 1 \rangle + \langle 2 \rangle$
 <code>electedPrimary</code>	$\langle M \rangle$	
 <code>electedSecondary</code>	$\langle M \rangle$	

Fig. 5. Global places shared by all modules of the election model

Tokens in place `negotiation` specify the content of sets `unconnected` and `negotiating` for all masters. For any pair of masters  $(m, n)$  with  $m \neq n$ , there is always a unique token  $\langle m, n, neg \rangle$  that specifies the current status of the negotiation between  $m$  and  $n$  as specified above in the description of class `NEG`.

For each message sent and not processed yet there is a token  $\langle r, s, t, d \rangle$  in place `network` where  $r$  is the receiver,  $s$  the sender,  $t \in MSG\_TYPE$  the type of the message, and  $d$  the uuid encapsulated in the message (meaningful only if  $t = AnsP$ , i.e. `AnswerPrim`).

Last, places `electionInit` (marked with  $\sum_{m \in \{1..MN\}} \langle m \rangle$ ), `electedPrimary`, and `electedSecondary` model different stages of the main election method: start of the negotiation, start of the election in “primary mode” or in “secondary mode”.

**Net Modelling the Main Election Method.** The net of Figure 6 is a high-level view of the election method. The subnet on the left-hand side of the figure

models the negotiation process with the broadcast of AskPrim messages (transition `sendAskPs`) and the network polling (transition `poll`). As soon as a master `m` knows it is a secondary master, a token  $\langle m \rangle$  is present in place `electedSecondary`. It then keeps polling the network (transition `secPoll`) until it knows the identity of the primary master. The subnet on the right-hand side models the behaviour of the primary master. Message `announcePrim` is broadcasted (transition `sendAnnPs`) and then the primary master keeps processing the messages received (transition `primaryPoll`).

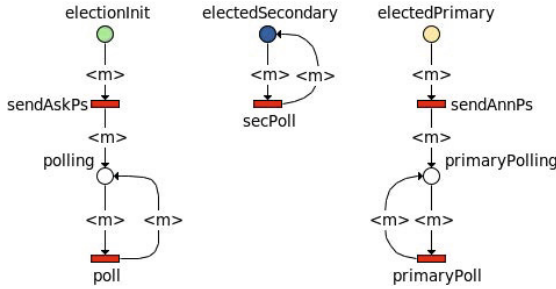


Fig. 6. Net modelling the main election method

**The Polling Mechanism.** A key element of the protocol algorithm is method `poll` that is called by nodes to handle messages received from the network (the polling mechanism is also used in the bootstrap phase presented in Section 5). This method is called by an event manager to which several handlers — one for each message type — are attached, and it only handles a single packet at each call by invoking the appropriate handler. In Figure 7 it is modelled by an input transition `start` putting a token in place `pollStart`. After processing a message, a token  $\langle m \rangle$  is present in place `pollEnd` and the master can then exit method `poll` (transition `end`). This is realised through the merging of these two

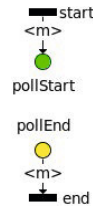
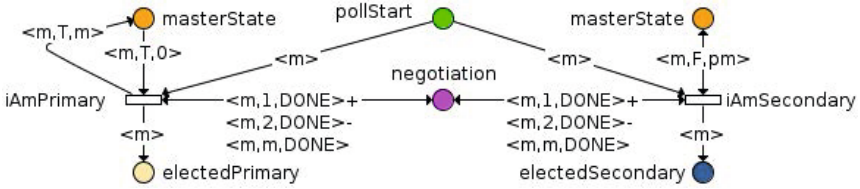


Fig. 7. Net modelling the `poll` method

places with their homonyms in the message handler nets (e.g., Fig. 9) as detailed below. Each transition modelling a message processing also moves a token from `pollStart` to `pollEnd`. Specifically for the case of meta-transition `poll`, we also include in its subnet the nodes of Figure 8. These model the exit condition of the negotiation step. The negotiation is over for master `m` if it is not negotiating with any other master anymore: there must not be any token  $\langle m, n, neg \rangle$  with `neg`  $\neq$  `DONE` in place `negotiation`. Depending on the content of place `masterState`, the token  $\langle m \rangle$  in `pollStart` will move to place `electedPrimary` or `electedSecondary` — both fused with their homonym places of net `electPrimary` (Figure 6).

If  $m$  has not received an `AcceptId` with a `uuid` greater than its own (see the corresponding handler in Figure 10), then a token  $\langle m, T, 0 \rangle$  is still present in place `masterState` and changed to  $\langle m, T, m \rangle$  since  $m$  learns it is the primary master (transition `iAmPrimary`). Otherwise, `masterState` is marked with token  $\langle m, F, pm \rangle$  and  $m$  knows it is a secondary master (transition `iAmSecondary`).



**Fig. 8.** Net modelling the decision process: master  $m$  has negotiated with all other masters and can decide of its role

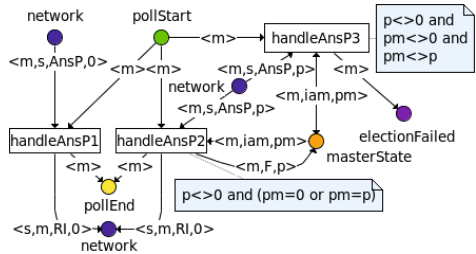
**Message Handlers.** Nets modelling message handlers are presented in Fig. 9, Fig. 10 and Fig. 11 along with the corresponding Python code. These nets follow the same pattern. Their transitions model the handling of a received message by removing one token from place `network` (the message received) and moving token  $\langle m \rangle$  (the receiving master) from place `pollStart` to place `pollEnd`, hence specifying the message has been processed and the master can exit the `poll` function (see the net of Figure 7). The variable  $s$  of each transition identifies the sender of the message. Alternatively, the master token can be put in place `electionFailed` if the processing of the message raises the `ElectionFailure` exception.

As handlers of message types `RequestId` and `AskPrim` are rather straightforward we have chosen to focus on types `AnswerPrim`, `AcceptId` and `AnnouncePrim`.

For messages of type `AnswerPrim` (Figure 9) we distinguish three cases:

```

1 def answerPrimary(self,
2     conn, packet, prim_uuid,
3     known_master_list):
4     app = self.app
5     if prim_uuid is not None:
6         if app.prim is not None and \
7             app.prim_master.getUUID() != \
8                 prim_uuid:
9             raise ElectionFailure
10    app.prim = False
11    app.prim_master = \
12        app.nm.getByUUID(prim_uuid)
13    conn.ask(RequestIdentification(
14        NodeTypes.MASTER, app.uuid,
15        app.server, app.name))
    
```



**Fig. 9.** Handler for message type `AnswerPrim`

- the peer  $s$  does not know any primary master (transition `handleAnsP1`). Local data are not changed by master  $m$  that replies to master  $s$  with a `RequestId` message (arc from `handleAnsP1` to `network`);
- transition `handleAnsP2` is fired if  $s$  knows a primary master ( $p <> 0$ ) and  $m$  does not know any or knows the same one ( $pm=0$  or  $pm=p$ ). The local data of  $m$  held in place `masterState` is updated and, once again,  $m$  replies to  $s$  with a `RequestId` message;
- last, an `ElectionFailure` exception (ll. 6–9) is raised if  $m$  and  $s$  both know a different primary master. This is modelled by transition `handleAnsP3`.

```

1 def acceptIdentification(self ,
2   conn , packet , node_type ,
3   uuid , address , num_partitions ,
4   num_replicas , your_uuid):
5   app = self.app
6   # error management
7   # ...
8   if app.uuid < uuid:
9     app.prim = False
10  app.negotiating \
11    .discard(conn.getAddress())

```

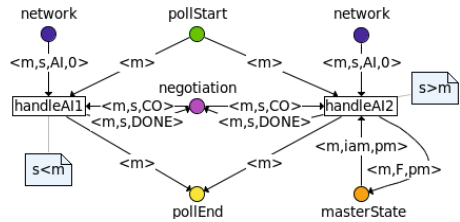


Fig. 10. Handler for message type `AcceptId`

At the reception of an `AcceptId` message (Figure 10), master  $m$  ignores the message if the enclosed `uuid s` is smaller than its `uuid` (transition `handleAI1`) or, if  $s > m$  (transition `handleAI2`), updates its local data by setting its `primary` field to `False` (ll. 8–9). In both cases, the content of place `negotiation` is changed to specify that  $m$  has finished negotiating with  $s$ :  $s$  is removed from the negotiating set of  $m$  (ll. 10–11). This will possibly trigger the exit by master  $m$  from the negotiation phase and enable one of the two transitions of the net of Figure 8.

Finally, a message of type `AnnouncePrim` can be handled in two ways (Figure 11) depending on the local data of the receiver  $m$ :

- $m$  does not think it is the primary master. It thus accepts the sender  $s$  as the primary master and updates its local data: the token `<m,iam,pm>` becomes `<m,F,s>`.
- $m$  also considers itself as the primary master (ll. 7–8 modelled by transition `handleAnnP2`) and thus raises exception `ElectionFailure`.

We mentioned that some synchronisation problems trigger the raise of exception `ElectionFailure` caught in the body of the main method of the election. One of the requirements of the protocol is that, in the absence of faults, this exception is not raised. Therefore, in that first modelling step, we left out the handling of this exception and verified through state space analysis that this exception may not be raised.

```

1 def announcePrimary (
2     self, conn, packet):
3     uuid = conn.getUUID()
4     # error management
5     # ...
6     app = self.app
7     if app.prim:
8         raise ElectionFailure
9     node = app.nm.getByUUID(uuid)
10    app.prim = False
11    app.prim_master = node

```

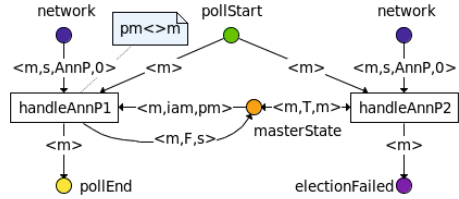


Fig. 11. Handler for message type AnnouncePrim

### 4.4 Injecting Faults in the Model

In sections 4.2 and 4.3, we only considered in our models the ideal situation where no malfunction may occur. Since the NEO system is intended to tolerate faults, it is a primary concern to enhance our models in order to analyse such scenarios. This injection of faults in the model raises several issues. First, we have to define the nature of the faults we are interested in. Both for modelling and state explosion issues we need to focus on some specific kinds of faults. Second, we must — for the same reasons — abstract the way these faults may occur. If we choose, for instance, to model packet losses, this means focusing on the loss of some specific “strategic” packets, even if any packet may be lost. Last, according to the faults we choose to model we need to reinvestigate the election program in order to determine which pieces of code that were abstracted away in our first modelling step (because they dealt with this kind of faults, e.g. the raise or handling of an exception) now need to be considered.

It appeared, during several meetings with the system designers, that the system should be able to recover from the crash of a master. The election protocol should also tolerate other types of faults, e.g., the loss of message, but since most of these are directly handled by lower level layers, they were not considered here. We then decided to restrain the occurrence of such events to two specific situations: the beginning of the election (when any master may be “allowed” to crash), and when a master learns it is the primary master, i.e. when transition *iAmPrimary* of the net of Figure 8 is fired. The first scenario is the most realistic one: in most cases, the election begins precisely because of a primary master failure. The second one is due to the specific role of the primary master: it announces its existence to other masters, announcement that will cause the exit from the election protocol. Therefore, its failure is a critical event compared to the crash of a secondary master that has few consequences. As previously mentioned, a look at the election code reveals that these events would typically raise *ElectionFailure*, exception caught in the main method of the election algorithm. Other exceptional cases are managed in the election code, but most of these deal with errors that are out of the scope of our study, or are defensive programming issues. Therefore these were left out.

**Modelling the Crash of a Master Node.** The net of Figure 12(a) is the main net of Figure 6 modified to include the crash of a master. A fault is simply modelled by transitions `crash` and `primCrash` putting a token  $\langle m \rangle$  in place `crashed`. After its crash, a master may reboot and join again the election (transition `reboot`) or be considered as permanently dead (transition `die`) — at least during the election process. The details of the meta-transition `reboot` are not given due to lack of space. It consists of reinitialising all the internal data of the master, i.e. the content of places `masterState` and `negotiation`, and setting back the token  $\langle m, F \rangle$  in place `live` (described below) to  $\langle m, T \rangle$ .

Transitions `crash` and `primCrash` are substituted by the net of Figure 12(b) modelling the effect of a crash on the global system. In order to be visible by other masters, a crash must have two side effects. First, the token  $\langle m, T \rangle$  in place `live` modelling the fact that master  $m$  is alive (and considered as such by other masters) is changed to  $\langle m, F \rangle$ . Second, the network must be purged from all the messages sent to (or by) master  $m$ . Otherwise, if  $m$  recovers from its crash, it may handle a message received prior to its crash, an impossible scenario that we should not model. Also, a message is automatically ignored by the receiving master if it detects the crash of the sender. So, rather than changing the message handlers nets we decided to also purge the network from messages sent by  $m$ . This is the purpose of transitions `removeRec` and `removeSent`<sup>3</sup>. If transition `end` becomes enabled, the network does not contain any message with the identity of master  $m$ . To guarantee that no message that has to be removed from place `network` is received meanwhile by another master we ensure this treatment is atomic by protecting it with place `lock`. The meta-net of the `poll` function has naturally been changed in such a way that this lock has to be grabbed before a message is handled.

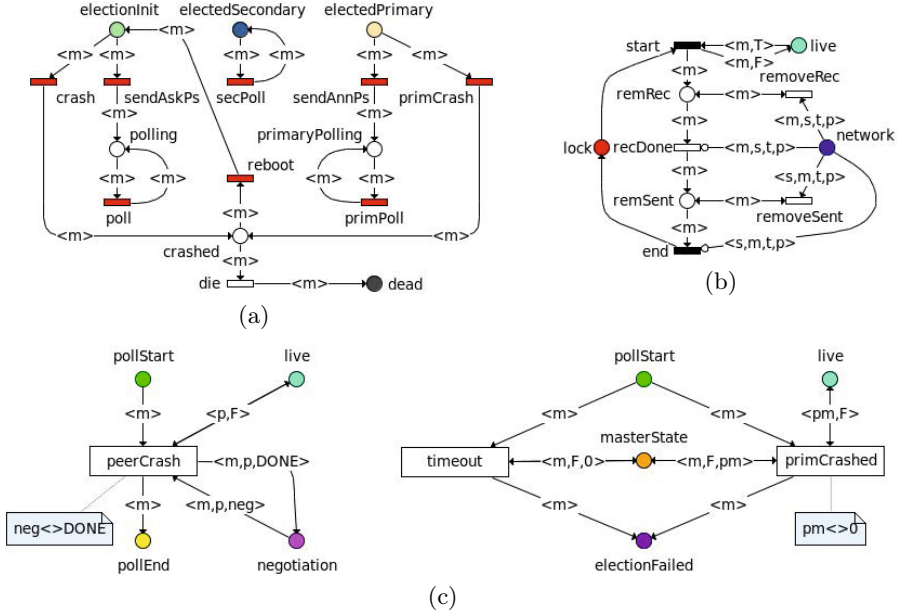
**Faults Detection.** The detection by a master  $m$  of the crash of one of its peers  $p$  is modelled by the net of Figure 12(c). Depending on the state of  $m$  this detection has different consequences.

If  $m$  initiated a negotiation with  $p$  and is still waiting for its `uuid`, it aborts the negotiation as soon as it detects its failure. From the model perspective this consists of removing  $p$  from both `s.unconnected` and `s.negotiating`. This first situation is modelled by transition `peerCrashed` that replaces token  $\langle m, p, \text{neg} \rangle$  by  $\langle m, p, \text{DONE} \rangle$  if master  $p$  is dead, i.e.  $\langle p, F \rangle \in \text{live}$ .

Alternatively, if  $m$  is a secondary master waiting for the announcement of the primary master election it can consider this one as dead if it does not receive an `AnnouncePrim` message after some amount of time. The expiration of this timeout is followed by the raise of exception `ElectionFailure`. The transition `timeout` models

<sup>3</sup> In order to ease the readability we have used inhibitor arcs to check the completion of the network purge. Since the verification tools we use do not support inhibitor arcs, the actual model includes a place counting the number of messages sent by (or to) any master. Zero-test is made via this place. Moreover, note that, due to the additional combinatorics this would generate, we do not model the possibility that a packet is received and handled between a sender crash and this crash detection by the receiver.





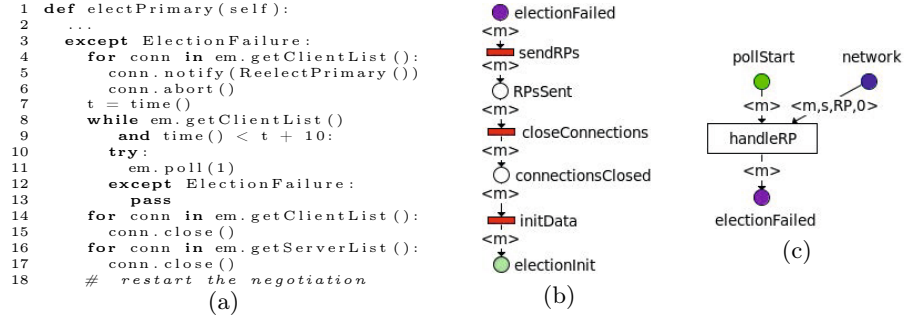
**Fig. 12.** Insertion of master node crashes in the model. Fig. 12(a): The top level net extended with crashes. Fig. 12(b): Side effects of a crash. Fig. 12(c): Detection of the crash of a master node.

this second scenario. One of its pre-conditions is the token  $\langle m, F, 0 \rangle$  to be in place `masterState` to specify that  $m$  is a secondary master not aware of the identity of the primary master.

Last, a secondary master  $m$  will raise exception `ElectionFailure` if it detects the failure of the primary master. This is the purpose of transition `primCrashed`. The master must be aware of the identity of the primary master to raise this exception, i.e.  $\langle pm, F \rangle \in \text{live}$  (with  $pm \neq 0$ ).

All these transitions are waiting for a token to be in place `pollStart` to become firable. Hence, they will be included in the appropriate meta-transition of the main net: transition `peerCrashed` will be put in the subnet of the meta-transition `poll` while transitions `primCrashed` and `timeout` will appear in the subnet of transition `secPoll`.

**Handling of Exception `ElectionFailure`.** Modelling the handling of this exception is essential if one wants to analyse the election protocol in the presence of faults since most synchronisation issues or fault detections will be followed by this exception raise. The code for handling this exception that we had voluntarily put aside in our first modelling phase can be seen on Figure 13(a). It consists of three parts: the broadcast of a `ReelectPrim` message intended to ask all peers to stop the current election process and start a new one (ll. 4–6); the processing



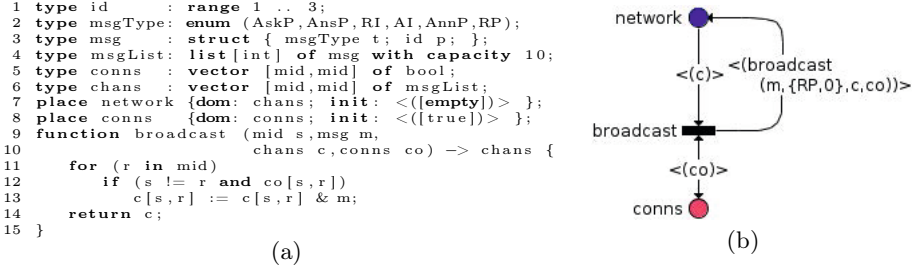
**Fig. 13.** Modelling the handler of exception ElectionFailure. Fig. 13(a): Handler of exception ElectionFailure. Fig. 13(b): Net modelling the exception handler. Fig. 13(c): Handler of message type ReelectPrim.

of incoming messages for some amount of time (ll. 7–13); and the closing of all connections (ll. 14–17). After that, the master restarts the election process.

The corresponding net is in Figure 13(b). Its structure reflects roughly the code. The transition `sendRPs` (of which we do not show the details here) puts a token  $\langle n, m, RP, 0 \rangle$  in place `network` for each alive master  $n \neq m$ . We then close connections (transition `closeConnections`). The subnet implementing this transition is exactly the one corresponding to the crash of a master (see Figure 12(b)). Indeed, from the viewpoint of another master, closing connections is equivalent to consider the master as crashed. This has the consequence of removing all messages of master  $m$  from the network. At last, the firing of transition `initData` reinitialises the internal data of the master and makes it alive to other masters in order to restart the negotiation. The subnet implementing this transition is the same as the subnet of transition `reboot` of the net of Figure 12(a). We see that the handler of this exception is quite equivalent to the crash and reboot of a master. We have left out the call to the `poll` method at l. 11. Indeed, its purpose is mainly to ensure that all peers have received the `ReelectPrim` message before closing the connections, and to ignore other `ReelectPrim` messages that could be received meanwhile (see ll. 12–13). Handling other messages is useless insofar as the election will be triggered again. This kind of timing issues needs not to be modelled. At last, Figure 13(c) depicts the net of the handler of `ReelectPrim` messages. At the reception of this message a master simply raises the `electionFailure` exception.

#### 4.5 Alternative Modelling with Helena

The Helena model has exactly the same module structure but is written in the language of the Helena tool [9]. Figure 14 presents a sample of the final model. The place `network` always contains a single token `c` of type `chans`. For each pair of masters  $(s, r)$ , `c[s, r]` is the list of messages sent by  $s$  to  $r$ . The broadcast by master  $s$  of a message  $m$  is achieved by function `broadcast`. One of its parameters



**Fig. 14.** Sample of the Helena model. Fig. 14(a): Some type and function declarations. Fig. 14(b): Model of the broadcast of a ReelectPrim message.

is the matrix *co* specifying which masters *s* is connected to. The broadcast of ReelectPrim messages can then be modelled with a single transition (Figure 14(b)), instead of performing a loop.

This language allowed us to model some features more concisely and to relax some constraints we had with symmetric nets that prevented us from modelling some parts of the protocol. For instance, the connection loss between masters is another type of faults that could be easily modelled in this new model. Although the system is not expected to tolerate such faults, the system designers were still interested to have some feedback on how the system could behave in the presence of disconnections and to which extent it could tolerate such faults. Broadcast of messages can also be modelled as shown by Figure 14(b). List types can also be used to model FIFO channels.

Note that this additional modelling effort was relatively small since the modelling tools we use (Coloane and our composition tool) are largely independent of the type of high-level net. Therefore, in many cases, we only had to rewrite arc labels from one language to another, an easy task, although a bit tedious.

### 4.6 Analysis

State space analysis has been conducted on the election model. Symmetric net modules were first assembled to produce a single net describing the protocol. In order to use symbolic tools of the CPN-AMI platform [11], this net was then unfolded in a low-level one using optimised techniques [15] and finally reduced [10] to produce a smaller net (but equivalent with respect to the specified properties).

The Helena model briefly described in Section 4.5 was also analysed using a slightly different procedure: since Helena can directly analyse high-level nets, the unfolding step was not performed, and the reduction was directly applied to the high-level net.

For the election protocol we formulated four requirements R0–R3 given below. First, we have seen that, if we do not consider faults, it is important that no exception is ever raised (R0). Two requirements are also logically required for the election protocol (R1 and R2). At last, we want to be sure the cluster can

**Table 1.** State space analysis of some configurations

Configuration			Nodes	Arcs	Terminal nodes	Analysis Results			
Masters	Crashes	Disconnections				R0	R1	R2	R3
2	no	0	78	116	1	✓	✓	✓	✓
		1	102	165	6	×	×	×	
	yes	0	329	650	6	✓	✓	×	
		1	434	968	10	×	×	×	
3	no	0	49,963	169,395	1	✓	✓	✓	✓
		1	57,526	206,525	52	×	×	×	
	yes	0	1,656,002	6,446,764	31	✓	✓	×	
		1	2,615,825	10,313,162	84	×	×	×	

enter its operational state (R3).

**R0** — The `ElectionFailure` exception is not raised.

**R1** — A single primary master is elected.

**R2** — All masters are aware of the identity of the elected primary master.

**R3** — The election eventually terminates.

Properties R0, R1 and R2 can be expressed as safety properties while R3 reduces to the absence of cycles in the reachability graph.

Next, we give some elements on the analysis of different configurations we experimented with, and present two suspicious election scenarios encountered.

*Analysis of Some Instances.* State space analysis has been performed on some instances of the election model listed in Table 1. It also gives statistics we have gathered on their reachability graphs. A configuration is characterised by the number of masters (column **Masters**) joining the election, the possibility of observing master crashes (column **Crashes**), and the number of disconnections that may occur (column **Disconnections**). The table gives for each configuration the number of nodes, arcs and terminal nodes of its state space and indicates for each of the three requirements we have checked whether it is matched (✓) or not (×) for this configuration. Requirement R0 was only checked for faultless configurations as the raise of an exception is naturally expected in the presence of faults. Our observations are the following ones:

- in the faultless configurations (N,no,0), the election behaves as expected;
- the possibility of a master crash does not break requirements R1 and R2 but does not guarantee the termination of the protocol even if we put aside trivial infinite scenarios during which a master keeps crashing and rebooting;
- connection loss between two masters is a severe kind of fault in the sense that the protocol does not show any guarantee in their presence. We actually found out very few situations where requirements R1–R3 are still verified despite a disconnection.

*Faulty Scenarios.* The first scenario is quite straightforward and could be discovered by simulating any configuration that includes a disconnection possibility.

Let us assume that the protocol is executed by two masters. If they get disconnected, then two elections will take place. Each master is isolated and thus declares itself as the primary master. Some storage nodes will then connect to one master and others will connect to the other master. Hence, there will really be two NEO clusters running separately and the data on the storage nodes will progressively diverge. This scenario is actually not unrealistic if we remember that nodes can be distributed worldwide.

A second suspicious scenario is due to lower level implementation details related to the handling of exception `ElectionFailure`. It can be reproduced with 3 master nodes M1, M2 and M3. Let us assume that M3 gets elected but crashes immediately after being elected. M2 (or M1) then detects this crash, raises exception `ElectionFailure` and sends a `ReelectPrim` message to M1. M1 receives this message and automatically proceeds the same way. Now let us assume that meanwhile M2 closes all its connections and restarts the election before M1 sends its `ReelectPrim` message. The `ReelectPrim` message is therefore not received in the handling of exception `ElectionFailure` (in which case it would be ignored) but after the restart of the election process. This will again cause M2 to raise an `ElectionFailure` exception, send a `ReelectPrim` message to M1. If M1 receives its message after it restarts the election (as M2 did), it will proceed exactly the same way. Hence, we can observe situations where M1 and M2 keep exchanging `ReelectPrim` messages that cancel the current election and restart a new one, thus constituting a *livelock*. The election will never terminate.

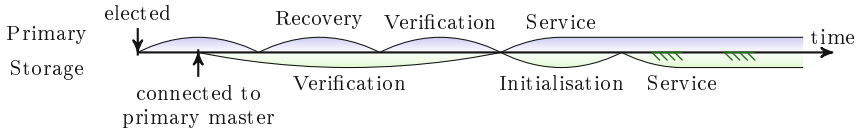
Both problems have been reported to the system designers. They are considering some extensions that could prevent the first scenario. It was not clear whether the second scenario is an actual bug or if it is a spurious error due to an over-abstraction in our modelling. Tests were carried out in order to reproduce this situation. Actually, a programming language side effect avoids this problem. The engineers will work on the code to remove this ambiguity.

## 5 Formal Analysis of the Bootstrap Protocol

The general goal of the bootstrap protocol initiated after the election of the primary master is to ensure that the database is in a consistent state before the system enters its operational state. To reach this consistent state, the following points must be checked:

- All expected data are stored on the storage nodes that are responsible of it.
- All transactions (in its database meaning) have been completed.
- All nodes have the same partition table and are aware of the identifiers (IDs) of the last transaction performed and of the last object updated.

In Figure 15, the different phases of primary master and storage nodes are displayed along a time axis. This graph corresponds to the normal work of the protocol. An error occurring in one phase may cause the recall of preceding phases.



**Fig. 15.** Primary master and storage nodes phases in time

Each phase (e.g. verification, recovery) in the lifecycle of a node is characterised by a *handler* specifying how the different messages expected during this phase are handled. From the code perspective, a handler is a Python class containing one method per type of packet expected. This method is triggered upon the reception of such a packet. In the absence of the appropriate method, the packet is rejected and an exception raised. These handlers are called by the `poll` method that has been described in more details in Section 4.

Right after its election the primary master first does some preliminary work: it announces itself, and checks the list of known storage nodes. Normally, during this period of time a storage node should connect to the primary master and set up the verification handler. It means that from the storage side a verification phase begins. Meanwhile, the primary master starts a recovery manager. After the recovery manager finishes its work, the primary master starts the verification manager, which verifies all the pending transactions on storage nodes. Verification phases of storage and primary nodes finish at the same time. Then the primary master sets up its service handler, and the storage node sets up an initialisation one, moving to an initialisation phase. When the initialisation phase is completed, the storage node goes to service state, performing replication (hatching on Figure 15) of data from time to time (a first time at the very beginning since some data might be missed while the storage node was down).

To be in the operational state, a storage node must be connected to the primary master, have an up-to-date partition table, the last identifiers (last transaction ID and last object ID) and the list of available nodes (regardless of their type). All this is obtained during the first phases of the storage cycle. The cluster state reached is then sound since data is consistent across storage nodes before the cluster becomes operational.

If an exception is caught while the system is operational, it may lead to restart the execution from one of the preliminary phases, according to the exception handled. For example, in case the primary master crashes, a new master node is elected and storage nodes must receive from this new primary master all the information listed above.

## 5.1 Model Architecture

The graphical conventions are the same as in Section 4.2. For instance, `poll_ver` (see Figure 16) is an abstract transition which is “implemented” by the net of Figure 7. Places from initial to operate in Figure 16 model the control flow of

storage nodes. The model of the `poll` function introduced in the previous section has been reused during the modelling of the bootstrap protocol. Therefore, we only describe in this section the packet handlers of the bootstrap steps.

**Shared Places.** Places `network` (described in Section 4.3), `has_pt_ni_lid`, `LastOID` and `LastTID` model important resources of the protocol and are shared by several modules.

- Place `has_pt_ni_lid` allows to check when storage nodes can move from the verification phase to the operational mode. It always contains, for each storage node  $s$ , a single token  $\langle s, pt, ni, lid \rangle$  such that  $pt = T$  iff  $s$  received the partition table;  $ni = T$  iff  $s$  received the list of nodes belonging to the cluster (i.e. the node information); and  $lid = T$  iff  $s$  received the last identifiers regarding the partition table.
- Places `LastOID` and `LastTID` contain, for each (storage or master) node  $n$ , the last object (resp. transaction) identifier  $id$  the node is aware of. This information is modelled by a token  $\langle n, id \rangle$  in the corresponding place.

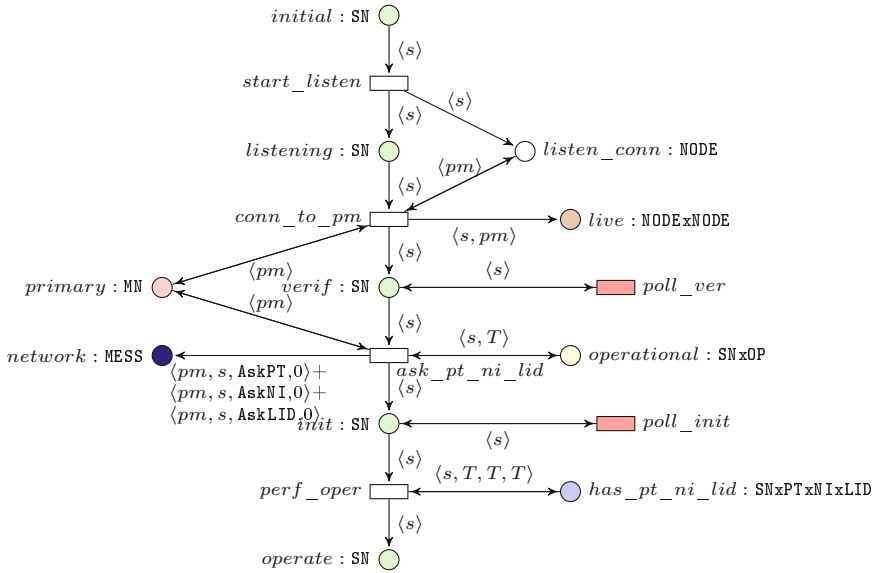
**Global Level Storage Node Model.** The model in Fig. 16 together with the declarations in Table 2 represent the functioning of a storage node from a global perspective. Every storage node starts its life cycle in place `initial`. It then listens to connections with some identification handler (transition `start_listen` puts a storage token into place `listen_conn`), and handles all the attempts of other nodes to connect. This is typically used during the replication phase, when some storage node has detected that it is out-of-date. It connects to another storage node that currently has the up-to-date copy of the required data.

The primary master also starts listening to connections at some moment. From then on, other nodes can connect to it. Note that no node can connect to another one if the second one is not listening to connections.

The next important step in the storage node life cycle is the connection to the primary master. Formally, it waits until place `primary` contains a token (in the current version of the model it means that the primary master has announced itself; later it should be modelled by exchange of messages) and a place `listen_conn` has the same token, meaning that the primary master has opened

**Table 2.** Declarations for the nets in Figures 16-19

<b>class</b>	<b>domain</b>
<code>SN</code> is 0..10;	<code>SNxOPER</code> is $\langle \text{SN}, \text{OP} \rangle$ ;
<code>MN</code> is 1..3;	<code>SNxPTxNIxLID</code> is $\langle \text{SN}, \text{PT}, \text{NI}, \text{LID} \rangle$ ;
<code>MTYPE</code> is $[\text{AskNI}, \text{AskPT}, \text{AskLID}]$ ;	<code>NODE</code> is $[\text{SN}, \text{MN}]$ ;
<code>OP, PT, NI, LID, REP</code> is $[\text{T}, \text{F}]$ ;	<code>NODExNODE</code> is $\langle \text{NODE}, \text{NODE} \rangle$ ;
<code>PART</code> is 1..20;	<code>MESS</code> is $\langle \text{NODE}, \text{NODE}, \text{MTYPE}, \text{INT} \rangle$ ;
<code>PSTATE</code> is $[\text{UP}, \text{OUT}]$ ;	<code>SNxID</code> is $\langle \text{SN}, \text{ID} \rangle$
<code>ID</code> is 1..100;	<code>SNxPART</code> is $\langle \text{SN}, \text{PART} \rangle$
<code>NSTATE</code> is $[\text{RN}, \text{TD}, \text{DW}, \text{BR}]$ ;	<code>NODExNSTATE</code> is $\langle \text{NODE}, \text{NSTATE} \rangle$
	<code>PARTxINT</code> is $\langle \text{PART}, \text{INT} \rangle$



**Fig. 16.** Storage nodes global level

a listening connection. Transition `conn_to_pm` checks the presence of these two tokens via test-arcs, puts one token containing a pair of storage and primary nodes into place `live`, which means that henceforth the connection between these two nodes is established. A storage token is put into place `verif`, saying that the storage node has started its verification phase by setting up a verification handler on its primary master connection.

The verification phase is supervised by the primary master, i.e. a storage node only receives messages and handles them (transition `poll_ver`) until one of the handlers changes the value of the variable `operational` to true. In the net, place `operational` contains as many tokens as there are storage nodes in the system. Each of these tokens consists of a storage node identifier and the current value of its internal variable `operational`. As soon as it becomes true, the storage node sends a message asking for the actual version of the partition table, the last identifiers and the node information to the network (transition `ask_pt_ni_lid`) and proceeds to the initialisation phase (place `init`).

Similarly, the storage node stays in place `init` listening to incoming messages (transition `poll_init`) until it receives the partition table, the last IDs (last transaction identifiers) and the node information.

Finally, a storage token arrives in place `operate`. Hence, the storage node has reached its operational state and starts providing service. If everything goes correctly, it remains in this state forever. If an operation failure occurs, the life cycle continues from connection to primary, but the current version of the model does not cope with errors yet.



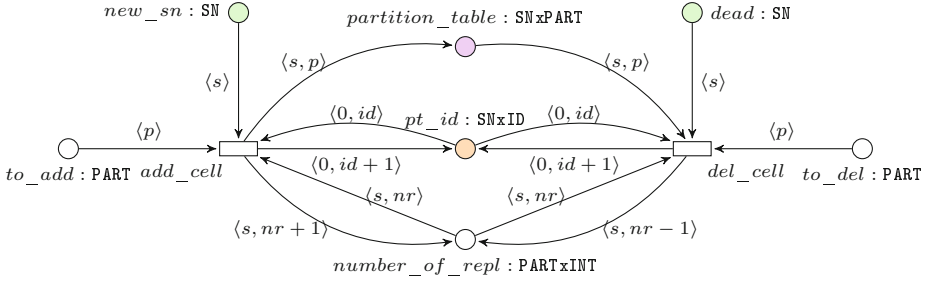


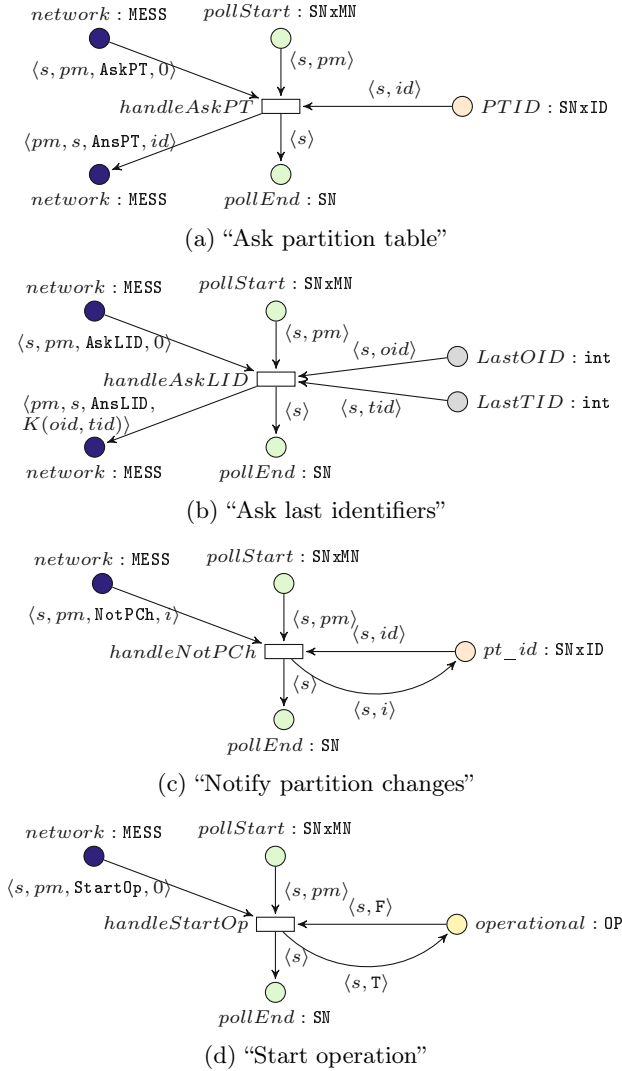
Fig. 17. Model of the partition table

**The Partition Table.** The partition table is one of the key elements of the protocol implementation. It allows for making a correspondence between pieces of data and the storage nodes where they (or their copies) are saved. The overall data is divided into partitions. The number of partitions is defined by the system administrator before the cluster is started, according to physical parameters of the system such as the expected volume of data, the number of available storage nodes, and the degree of data safety (how many replicas should be saved). The number of partitions cannot be changed during the cluster life cycle. For each object, the number of partitions to which it belongs is defined by the simple formula: (Object Identifier) *modulo* (Number of Partitions). So, the partition number is equal to the remainder of the division of Object Identifier by the Number of Partitions.

In the partition table, each row corresponds to one partition and contains the IDs of the storage nodes where the partition is located. Figure 17 represents the basic model of the partition table that contains two transitions: `add_cell` allowing to add a partition and `del_cell` allowing to delete a partition. The internal structure of the table is represented by two places: `partition_table` and `number_of_repl`. The first one contains pairs  $\langle \text{storageID}, \text{partitionID} \rangle$  establishing a correspondence between partitions and storage nodes. If the protocol operates correctly, there should be no duplicates. Place `number_of_repl` contains exactly one token per partition that includes its ID and the number of tokens in the place `partition_table` corresponding to it. The other places model the input arguments for the functions `add` and `delete`, and are shared with other subnets.

## 5.2 Formal Modelling of the Verification Phase

Figure 18 presents the models handling messages, that are called when a storage node is in the verification phase. The goal of this phase is to check that all expected information persists and there is no pending transaction. The process of verification is managed by the primary master. All transitions have two common input places: `pollStart` (corresponding to a storage node that is handling a message) and `network` (corresponding to the message that is being handled). There



**Fig. 18.** Verification phase handlers of storage nodes for different message types

is also one common output place `pollEnd` where a storage token is placed after handling a message. Handling ask messages (with `ask` in the name) finishes with a response that is put into place `network`, to notify it is sent:

1. `handleAskPT` (Ask Partition Table, Figure 18(a)) — Since it is not possible to model the complete partition table without facing the state space explosion problem, only the ID number of the table is sent. Thus the response from a

storage node to a `AskPT` message is of type `AnsPT` with the current partition table ID as parameter.

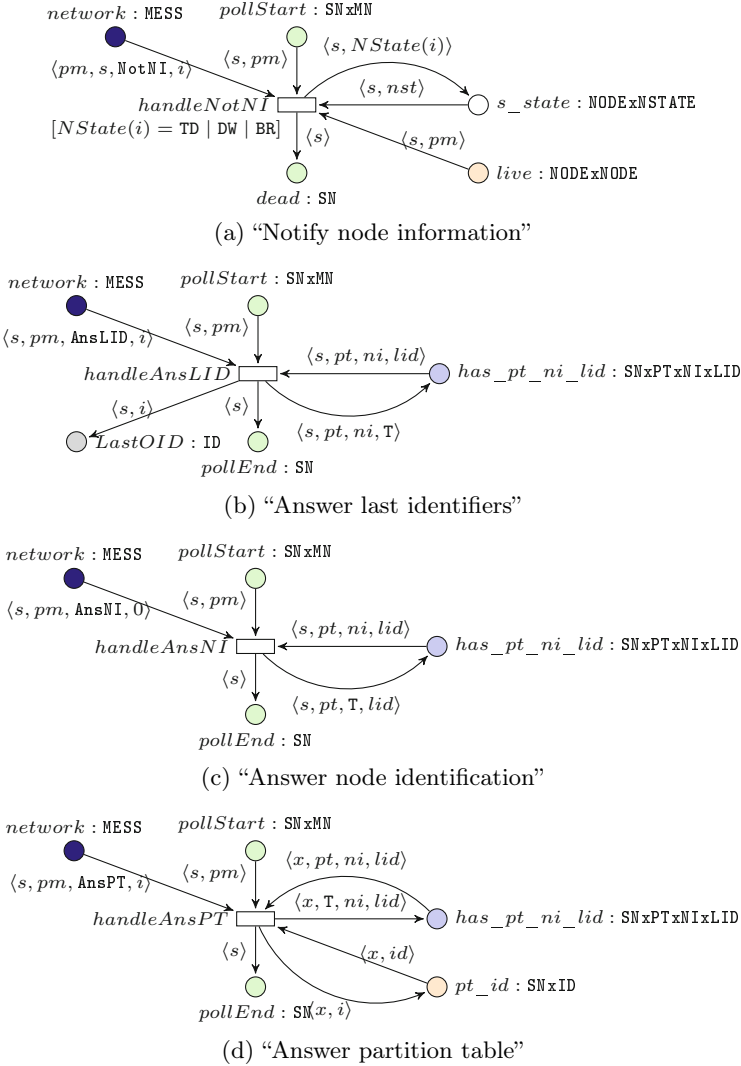
2. `handleAskLID` (Ask Last IDs, Figure 18(b)) — When a storage node receives an `AskLID` message, the primary master requests the last transaction ID and last modified object ID. If any of these numbers is greater than the one currently known by the primary master, the latter saves those IDs obtained from the storage node and considers them as last. Hence, transition `handleAskLID` takes two tokens from places `LastOID` and `LastTID` respectively (corresponding to the storage node that is currently handling this message) and replies with an answer message to the `network` place of type `AnsLID`: receiver `pm`, sender `s`, information  $K(\text{oid}, \text{tid})$  (where  $K$  is a one-to-one mapping from naturals to pairs of natural numbers).
3. `handleNotPCh` (Notify Partition Changes, Figure 18(c)) — This message is sent by the primary master in case the structure of the partition table is changed. This may happen for different reasons: some storage nodes crashed, new storage nodes are added or the distribution of the partitions is not uniform. This message actually contains all the rows of the table that have been changed, but for the modelling purposes only the partition table ID is sent. Hence transition `handleNotPCh` replaces a token corresponding to the current storage node in place `pt_id` with a new one with the ID just received. From then on, this storage node contains this partition table.
4. `handleStartOp` (Start Operation, Figure 18(d)) — This message is sent when the primary master considers the verification phase finished and allows to proceed to the next stage. Transition `handleStartOp` sets the value of the token in place `operational` to true. In the global level model, this activates transition `ask_pt_ni_lid` and storage nodes can move from place `verif` to `init`.

### 5.3 Formal Modelling of the Initialisation Phase

The goal of the initialisation phase (Figure 19) is that every node has the same partition table, and the IDs of last transaction and last modified object. The handlers for receiving last IDs and nodes information from the primary master are very simple. Storage nodes just save the values received in the appropriate places and change the corresponding boolean.

Similar to the verification phase, all transitions have two common input places (`pollStart` and `network`) and one common output place `pollEnd`.

1. `handleNotNI` (Notify Node Information, Figure 19(a)) — The primary master uses `NotNI` messages to announce to the storage nodes their new status. If the new status of a storage node is `DW` (down), `TD` (temporarily down) or `BR` (broken), it closes its connection with the primary master and shuts down. Formally, transition `handleNotNI` replaces a token corresponding to current storage node from `s_state` with one corresponding to the new status (function  $NState(i)$  is a simple mapping of integer  $i$  to the colour domain of storage states). It also removes a token with current storage and primary



**Fig. 19.** Initialisation phase handlers of storage nodes for different message types

master nodes from place *live*, signifying that the connection between them is down and adds the storage node to place *dead*.

2. *handleAnsLID* (Answer Last ID, Figure 19(b)) — The storage node saves its last object ID in its database (place *LastOID*) and changes the value of *has\_pt\_ni\_lid* from false to true, meaning that it now knows last IDs.
3. *handleAnsNI* (Answer Node Information, Figure 19(c)) — Similar to the previous message, only the boolean value of *has\_pt\_ni\_lid* is changed.

4. `handleAnsPT` (Answer Partition Table, Figure 19(d)) — The corresponding boolean is also changed, and a token in `pt_id` is replaced with the one containing the new partition table ID received in the message.

## 5.4 Desired Properties

The engineers working on the NEO protocol provided us with more than 70 properties (related to the whole system and not only to the election and bootstrap phases) they would like to check on our models. These descriptions had first to be refined for several reasons: the terms used to express some properties had different meanings according to the context, several properties were poorly expressed and had to be refined, others were really trivial to check and only required a careful look at the implementation, ... Therefore, the statements were rewritten and we (with the engineers of the NEO protocol) retained three main requirements, namely **R4**, **R5** and **R6**, that concern the bootstrap protocol.

**R4** - The first requirement for the bootstrap phase states that all storage nodes eventually reach the operational mode. It actually implies that the following two conditions hold:

- All storage nodes have reached the operational state in every terminal node.
- The reachability graph is acyclic.

**R5** - According to this requirement, there is at least one storage node, for each partition, that will be responsible for storing that partition.

**R6** - Similarly, the third requirement, implying that any storage node will store at least one partition, can also be expressed as a reachability property.

## 5.5 Analysis Results

We analysed several configurations of our CPN model through simulation and then through state space analysis. Table 3 provides statistics on the reachability graphs of these various instances. Each configuration is defined by a number of storage nodes (column N), and a replication factor (column Repl.), i.e. the number of storage nodes a partition is kept on.

For all configurations we analysed, the reachability graph is acyclic. Hence, the termination of the bootstrap protocol is guaranteed. Moreover, all terminal markings respect our three requirements **R4**, **R5** and **R6** presented in Section 5.4 and describe only acceptable termination states of the bootstrap protocol:

**Table 3.** State space results for several configurations

Configuration	Markings	Terminal	Transitions
N	Repl.	markings	
1	1	537	9 905
2	1	22,952	106 57,059
2	2	76,590	106 217,897

- all storage nodes and the primary master have reached the operational state;
- the database is consistent: a partition is owned by at least one storage node and no storage node is empty.

Terminal markings only differ on the content of place `network` containing messages in communication channels. This could mean that some messages (those still present in the `network` in the termination states) are not required for the system to be operational since they are not consumed. Therefore, the protocol could possibly be optimised by avoiding some messages in specific situations.

It is noteworthy that the state space grows considerably with the number of storage nodes. Actually, more than the number of nodes involved, the number of messages exchanged is the major bottleneck in our analysis. Indeed, each time a node invokes function `poll` it can treat numerous different packets received, hence generating a comparable number of transitions. We believe that the use of partial order reductions [7] could efficiently tackle this issue. Since in most cases the order in which incoming packets are treated is irrelevant, the use of this technique should naturally leverage this source of combinatorial explosion.

To conclude this analysis section we will stress the fact that although we did not find any actual problem in the implementation, we plan to perform further model checking with larger configurations and other analysis tools, e.g. using partial order or symmetry based reductions [5].

## 6 Conclusion and Perspectives

In this paper, we have presented our work on the modelling and analysis of the first two (and essential) steps of the NEO system, a protocol developed to manage very large distributed databases. The correctness of these steps is vital since it implies a coherent system state and database consistency when clients start querying the database. Checking this correctness is also probably the most difficult point since, once the system is functioning, synchronisations ensuring data consistency seem simpler. Modelling is achieved using a *reverse-engineering* approach from the code. It required to devise appropriate choices to work on relevant and useful *levels of abstraction at different steps*. To this end, we used several tools motivated by the different complexities of the objects to be modelled: namely Coloane, CPN-AMI, CPN tools and Helena. We also identified a lack in the modeling tools we used in that, besides CPN tools, they do not easily support the creation of nets modularly and hierarchically structured. This observation led us to define an XML-based composition language to ease our task. This is, to our best knowledge, the first attempt to define such a language.

The outcome of this analysis was profitable to the system designers in several ways. First, we could discover several suspicious election scenarios that led them to make their code more robust. Second, our analysis confirmed that a connection loss between two masters is a severe fault from which the system will not recover in most situations. Last, we increased their confidence in the bootstrap protocol by checking several configurations in which all expected properties are verified.

In the future, three extensions of the work presented here will be considered. First, we intend to take into account the storage nodes failure during bootstrap. Several mechanisms are implemented by the protocol to manage this kind of issue and it is worthwhile analysing them through model checking. Second, we plan to verify the considered properties when the number of storage nodes is not known in advance, and thus take into account the fact that the number of storage nodes can change dynamically during execution. Finally, we plan to use other analysis techniques, especially through Coloane which can interface with several verification tools such as Great-SPN [2] and Prod [17].

## References

1. The Coloane tool Homepage, <https://coloane.lip6.fr/>
2. The GreatSPN tool Homepage, <http://www.di.unito.it/~greatspn>
3. The ZODB Homepage, <http://wiki.zope.org/ZODB/FrontPage>
4. Bertrand, O., Calonne, A., Choppy, C., Hong, S., Klai, K., Kordon, F., Okuji, Y., Paviot-Adet, E., Petrucci, L., Smets, J.-P.: Verification of Large-Scale Distributed Database Systems in the NEOPPOD Project. In: PNSE 2009, pp. 315–317 (2009)
5. Chiola, G., Dutheillet, C., Franceschinis, G., Haddad, S.: A Symbolic Reachability Graph for Coloured Petri Nets. TCS 176(1-2), 39–65 (1997)
6. Choppy, C., Dedova, A., Evangelista, S., Hong, S., Klai, K., Petrucci, L.: The NEO Protocol for Large-Scale Distributed Database Systems: Modelling and Initial Verification. In: Lilius, J., Penczek, W. (eds.) PETRI NETS 2010. LNCS, vol. 6128, pp. 145–164. Springer, Heidelberg (2010)
7. Clarke, E.M., Grumberg, O., Minea, M., Peled, D.: State Space Reduction Using Partial Order Techniques. STTT 2(3), 279–287 (1999)
8. ERP5. Central Bank Implements Open Source ERP5 in Eight Countries after Proprietary System Failed, <http://www.erp5.com/news-central.bank>
9. Evangelista, S.: High Level Petri Nets Analysis with Helena. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 455–464. Springer, Heidelberg (2005)
10. Haddad, S., Pradat-Peyre, J.-F.: New Efficient Petri Nets Reductions for Parallel Programs Verification. Parallel Processing Letters 1, 16 (2006)
11. Hamez, A., Hillah, L., Kordon, F., Linard, A., Paviot-Adet, E., Renault, X., Thierry-Mieg, Y.: New Features in CPN-AMI 3: Focusing on the Analysis of Complex Distributed Systems. In: ACSD 2006, pp. 273–275. IEEE Computer Society (2006), <http://move.lip6.fr/software/CPNAMI/>
12. Huber, P., Jensen, K., Shapiro, R.M.: Hierarchies in Coloured Petri Nets. In: Rozenberg, G. (ed.) APN 1990. LNCS, vol. 483, pp. 313–341. Springer, Heidelberg (1991)
13. Jensen, K., Kristensen, L.M.: Coloured Petri Nets, Modelling and Validation of Concurrent Systems. Springer Verlag Monograph (2009)
14. Jensen, K., Kristensen, L.M., Wells, L.: Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. STTT 9(3-4), 213–254 (2007)
15. Kordon, F., Linard, A., Paviot-Adet, E.: Optimized Colored Nets Unfolding. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 339–355. Springer, Heidelberg (2006)
16. University of Hamburg. The Petri Nets Tool Database, <http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html>
17. Varpaaniemi, K., Heljanko, K., Lilius, J.: Prod 3.2: An Advanced Tool for Efficient Reachability Analysis. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 472–475. Springer, Heidelberg (1997)

# An Initial Coloured Petri Net Model of the Hypertext Transfer Protocol Operating over the Transmission Control Protocol

Sonya Arnold and Jonathan Billington

Distributed Systems Laboratory

University of South Australia

Mawson Lakes Campus, SA 5095, Australia

klos1001@mymail.unisa.edu.au, j.billington@unisa.edu.au

<http://www.unisa.edu.au/csec>

**Abstract.** Transfer of resources in the World Wide Web is achieved by using the Hypertext Transfer Protocol (HTTP). The most widely used version of HTTP in the Internet is HTTP/1.1 published as Request for Comments 2616. HTTP/1.1 is currently undergoing revision and is being restructured into a 7 part document by the Internet Engineering Task Force. Part 1 includes a description of the operation of HTTP over reliable transport connections, and is currently a relatively mature Internet Draft. It is therefore timely to subject these revisions to a rigorous analysis using formal techniques. This paper presents an initial Coloured Petri Net model of HTTP operating over transport connections provided by the Transmission Control Protocol (TCP). This requires modelling HTTP entities, the transport service provided by TCP, and their interactions. The design of the model, including its underlying assumptions and validation, is discussed and reachability analysis results are reported. Insights are gained into: the size of the state space as a function of web requests; and the dimensioning of buffers required in HTTP entities.

**Keywords:** HTTP, persistent and non-persistent connections, reliable transport service, Coloured Petri Nets, reachability analysis.

## 1 Introduction

Our tendency to obtain information on-line is steadily increasing with advances in Internet technologies. Information is most commonly stored on the World Wide Web and accessed with a web browser. The transfer of information between a web browser (the *client*) and a web server (the *server*) is achieved using request-response transactions governed by the Hypertext Transfer Protocol (HTTP) [6].

HTTP/1.1 is specified in *Request for Comments* (RFC) 2616 [6], published by the Internet Engineering Task Force (IETF) [15]. Over the last few years, a new working group known as *HTTPbis* [10] has been formed within the IETF to refine RFC 2616. The HTTPbis charter states that the working group will



incorporate errata and updates, and remove ambiguities while resisting the introduction of new functionality. However, where existing procedures have proved to be faulty, new elements of procedure may be introduced to fix the fault. What has eventuated is a 7 part specification which redefines HTTP/1.1. HTTP/1.1 is a relatively complex protocol, with just Part one [7] spanning 82 pages (when the change log is removed), with a body of 65 pages (compared with the Transmission Control Protocol (TCP) [14] with 85 pages).

The standard behaviour of HTTP/1.0 [1] was to establish and release a TCP connection for every request-response pair. Research on the performance of this design demonstrated how inefficiently HTTP operated over TCP [9, 13]. While HTTP/1.0 made no provision for keeping connections open for multiple request-response pairs (known as *persistent connections*) in its documented form, some experimental implementations used the Keep-Alive header to request a TCP connection remain open for multiple request-response pairs. This design proved faulty as it was unable to interoperate with intermediate HTTP/1.0 proxies [12]. It was not until RFC 2068 HTTP/1.1 [5] (later replaced by RFC 2616 [6]) was standardised that persistent connections became the default behaviour. To manage these connections, HTTP/1.1 introduced a mechanism by which a client and a server can signal the close of the connection. HTTP/1.1-part1 [7] clarifies the management of persistent connections. The removal of ambiguities and clarification surrounding persistent connection management is a current and ongoing task for HTTPbis. HTTP/1.1-part 1 is currently at version 17, and in 2011, 5 different versions were produced, demonstrating the high level of activity within the group. It is therefore timely to develop a formal and hence unambiguous specification of HTTP procedures and their management of persistent connections, particularly TCP connections. Further, section 2.3 of HTTP/1.1-part1 states that it does not address the mapping of request and response messages to transport service data units (SDUs). Hence this paper focuses on providing a formal model of HTTP and its management of persistent connections (according to Part 1), including the mapping of messages to transport SDUs.

Previous research on persistent connections has concentrated on demonstrating their performance advantages. Heidemann et al. [9] discuss the performance of HTTP operating over several transport protocols, including TCP and UDP. Their article analyses TCP connection start-up costs but does not consider the detail of the closing mechanisms. Wu et al [17] model persistent connections using Stochastic Petri Nets and consider how they perform under various workloads. However, their model does not consider HTTP procedures operating over a transport connection in any detail. Gvozdanovic et al. [8] discusses traffic generation using Petri nets and HTTP, but in this case the transport service is not considered. Wells et al. [16] provide a timed hierarchical Coloured Petri Net model of HTTP/1.0 procedures, but do not consider HTTP/1.1 persistent connections. They do consider detailed models of HTTP's operation over TCP, but the TCP modules are not provided in [16]. Their model is aimed at simulation-based performance analysis of web servers, and although the modelling formalism supports functional analysis

using state spaces, no state space analysis is reported due to the size and complexity of their model.

To overcome some of these limitations, we present a hierarchical Coloured Petri Net (CPN) model of HTTP/1.1 operating over the service provided by TCP. We firstly list the assumptions made and then provide an abstract model suited to functional analysis using state spaces. All modules are described so that HTTP's interaction with the transport service is readily apparent. We include the ability of HTTP/1.1 to signal connection closure to a peer and to open 2 successive connections to support the delivery of a request, when faced with premature closure of the first connection by the server. Further, we analyse the model using state spaces and provide some initial results.

Compared with previous work, this paper makes several contributions. Firstly it provides the first Coloured Petri Net model of the handling of persistent connections in HTTP/1.1. This includes the ability of both the client and server to signal connection closure and for the client to be able to reopen connections to a server that has failed to deliver the response, during a single HTTP session. Secondly, the details of the transport service are included and it is shown how requests and responses are mapped to transport service data units. Thirdly, the model is shown to be amenable to state space analysis. This has allowed us to firstly validate the behaviour of the model, and then to prove basic properties of the protocol including correct termination and absence of livelock. We also provide some insight into the maximum number of messages that can reside in HTTP buffers and the transport service during execution of the protocol.

The rest of the paper is organised as follows. Section 2 describes HTTP/1.1's procedures [7]. Our model is presented in section 3 with analysis results in section 4. We draw some conclusions and discuss future work in section 5. This paper assumes knowledge of Coloured Petri Nets [11].

## 2 HTTP

HTTP has been developed to support distributed and collaborative information systems through the use of hypertext. It is an application protocol residing above the Transmission Control Protocol in the Internet Protocol Architecture. It uses a client/server paradigm where the client establishes a connection to send requests and servers accept connections in order to respond to requests. Clients that support persistent connections may send their requests in lock-step, waiting for each response before issuing the next request, or they may *pipeline* [6] their requests. Pipelining allows a client to make multiple requests on a single connection without waiting for each response. Servers must send their responses to pipelined requests in the same order that the requests were received.

Client requests are typically triggered by user actions such as clicking on a hypertext link in a browser to obtain a web page. This stimulates the HTTP client (browser) to open a transport connection. Once the connection is established the request is sent to the server. If successful, the response is returned to the client. The response may contain references to further resources (such as

images). To obtain these resources, the client must make further requests either on the same transport connection or by opening new and possibly concurrent connections. Once all resources have been obtained, the user will be able to view the completed web page.

HTTP/1.1-part 1 specifies: the general architecture of HTTP, including Uniform Resource Identifier (URI) schemes; message formats, parsing and routing; syntax notation and transfer codings; connection types and their operation; header field definitions; registration requirements; and security considerations. Our paper focuses on section 6.1 *Persistent Connections* and section 8.1 *Connection*, where the header field associated with non-persistent connections is discussed.

## 2.1 HTTP/1.1 Messages

There are 2 HTTP message types: a *request* from the client and a *response* from the server. HTTP messages consist of a *start-line* followed by a series of optional *header fields* and an optional message body. Requests and responses differ only in the syntax of this start-line. A request start-line consists of a method, uniform resource identifier (URI) and the version of HTTP being used by the client. The method specifies the operation to be performed on the resource, such as GET (get the resource located at the URI) or PUT (put the resource at the location specified by the URI). An example request is: *GET http://www.unisa.edu.au HTTP/1.1* which will open the home page of the University of South Australia. A response start-line contains the HTTP version in use by the server, a status-code and a reason-phrase. The status-code is a 3 digit code indicating the result of the request and the reason-phrase is a textual explanation of that code. An example response is *HTTP/1.1 200 OK* which indicates the request was successful.

## 2.2 HTTP/1.1 Connection: Close Header

HTTP/1.1 assumes a persistent connection will be used unless the client or the server signals otherwise. This signaling is achieved through the inclusion of the header field *Connection: close* in either a request or a response. A requirement for HTTP/1.1 clients and servers not supporting persistent connections is that *Connection: close* must be included in every request and every response. The transport connection may be closed by the client, or the server or by both concurrently.

If the client includes the *Connection: close* in a request, it may initiate a graceful close of the transport connection either before it receives the response or after, or it may wait for the server to close the transport connection. When the client sends the *Connection: close* header it is letting the server know that it will not send any further requests on the current transport connection. The transport service supports graceful close whereby, once the client closes the transport connection, it may still receive responses until the server has closed its end of the transport connection. When the client receives a *Connection: close* header

in a response from the server it must not send any further requests. It may close the transport connection or wait for the server to do so.

When the server includes the `Connection: close` header it may close the transport connection after it has sent its response, or it may choose to wait for the client to close the transport connection. By sending the `Connection: close` header the server is signaling to the client that it no longer supports the persistent connection.

If the server receives a `Connection: close` header, it knows that this will be the last request received. After it has sent its response it may close the transport connection or it may wait for the client to initiate transport connection release.

In addition, the client or server may close the transport connection at any time (as stated in section 6.1.4 and 6.1.5 of [7]), regardless of the inclusion of the `Connection: close` header. For example, the user may close the web browser in the middle of a web page request or the server or client may time-out waiting for a request or a response. Time-outs of the transport connection are discussed in section 6.1.4 of [7]. Time-outs are not mandated for persistent connections in HTTP/1.1, however maintaining inactive connections indefinitely is not practical and therefore they are discussed in [7]. When the client sends a request it may wait a pre-determined amount of time before deciding that the response is not forthcoming. This results in a client time-out causing the client to discard the current request, initiate a graceful close of the transport connection and notify the user. Likewise a server may not maintain an idle connection with a client indefinitely. This may result in a server time-out whereby the server initiates a graceful close of the transport connection. If a request has not been responded to after the connection has been closed (from a server time-out for example), section 6.1.5 [7] states “Client software MAY reopen the transport connection and retransmit the aborted sequence of requests without user interaction...”. Section 6.1.5 of [7] also states “The automatic retry SHOULD NOT be repeated if the second sequence of requests fails”. If no response has been received for a request after 2 attempts, the request will be discarded and an error message reported to the user.

### 3 Initial CPN Model of HTTP/1.1

Using an incremental approach [2] our philosophy in modelling HTTP/1.1 is to capture its essential behaviour in an initial model, which we then validate. In particular, the model includes operating over persistent and non-persistent transport connections and the management of transport connections during an HTTP session.

#### 3.1 Modelling Assumptions

In order to handle the complexity of HTTP, we make some simplifying assumptions in our initial model. We also attempt to clarify HTTP’s procedures.

1. Proxies. Although HTTP clients and servers may operate over intermediaries such as proxies, they add considerable complexity and are out of the scope of our initial model. Thus we only consider direct connections between clients and servers.
2. Numbers of clients and servers. A client can interact with many servers during a browsing session, and likewise a server will normally have many clients interacting with it. However, each invocation of HTTP is between a single client and a single server. Thus we only consider one client interacting with one server as a single client/server pair is representative of the logical behaviour of HTTP. This allows us to abstract from location information (both the URI and the Host header field), as discussed below under the message abstraction assumption.
3. Connections. As discussed in section 2 a client may open multiple parallel connections to the server if it needs to obtain several resources. To reduce complexity we consider that the client only establishes one connection with the server at any time. However, we do consider the automatic reopening of the connection by the client if the connection is closed before a response has been received, as this is an important part of managing persistent connections. Further, we also model the timeouts at the client and server (discussed in section 6.1.4 of [7]) which monitor inactive connections. This is achieved by allowing the connection to be released by the server or client at any time after it has been established.
4. Pipelining. Pipelining of requests is optional and adds considerable complexity to the operation of HTTP, especially when taking into account reopening of connections. For this reason it has not been included in our initial model.
5. Responses automatically generating requests. We do not include a mechanism for a response to automatically trigger the sending of further requests. This often results in the opening of multiple connections or using pipelining which are out of scope for our initial model.
6. HTTP message abstraction. In order to model persistent connections we must consider the use of requests and responses which may (or may not) include the Connection: close header. We are not concerned with the type of method used in the request, except that we assume that the method is idempotent<sup>1</sup> (which caters for most methods). In the future we may wish to expand this model to include specific methods (therefore our CPN declarations have provision for this) so for realism we simply use the GET method. We do not include error handling and therefore only consider the server successfully responding to requests. Successful responses use status codes in the 200 range and their corresponding reason-phrases. We abstract from this class of status codes (and reason-phrases) by just using the value “success” to represent any one of them. This is valid because the basic operation of HTTP does not depend on a particular code in the 200 range or

---

<sup>1</sup> Idempotent methods are those that have the same effect if sent multiple times. The automatic reopening of connections excludes the use of non-idempotent methods (such as POST), so they are excluded from the current model.

its reason-phrase. We assume that the client and server both use HTTP/1.1 procedures, and thus are not checking backward compatibility of HTTP/1.1 with earlier versions. This allows us to exclude the version number when modelling requests and responses. Because we only consider a single client and a single server, we do not need to model the URI, nor the mandatory Host header field. We do not model optional header fields, except for the Connection: close, which is important for persistent connections. We also do not include the message body as it does not affect the basic operation of HTTP.

7. HTTP aborting connections. HTTP Part 1 only mentions gracefully closing connections. Thus we do not consider HTTP entities aborting connections. While connections may be aborted in implementations of HTTP, there is no specification dictating how this may occur. In addition, aborts add considerable complexity and are therefore out of scope for our initial model.
8. Persistent and non-persistent connections. Although HTTP Part 1 recommends persistent connections as the default behaviour, it also allows for non-persistent connections using the Connection: close header as discussed in section 2. Because Part 1 uses the words “may” and “should” in this context, we assume that a) the client or server can change its operation from a persistent to a non-persistent connection at anytime, and b) that it does not necessarily need to signal closing the connection with the close header before closing. Thus we model this situation non-deterministically. Part 1 states “Once the close has been signaled, the client MUST NOT send any more requests on that connection.” and also “If either the client or the server sends the close token in the Connection header field, that request becomes the last one for the connection.” We interpret this to mean that once the client has included the Connection: close header in a request, or received a response with the Connection: close header, it will not send any more requests.
9. Transport service. HTTP/1.1 assumes that it will operate over a reliable transport protocol with in-order delivery of messages. We have chosen to model the transport service based on TCP as it fulfils these criteria and is most commonly used in practice. We assume that the transport service always accepts connections and that they are successfully established and released. An abort service is not included in our initial model of the transport service.

### 3.2 Model Structure

Our CPN model of HTTP is created using CPN Tools 4. It is a hierarchical model organised into 3 levels, as shown in Fig. 1. The structure of HTTP and its environment are described in the HTTP architecture module. The second level models the HTTP client and server and transport service. The client and server

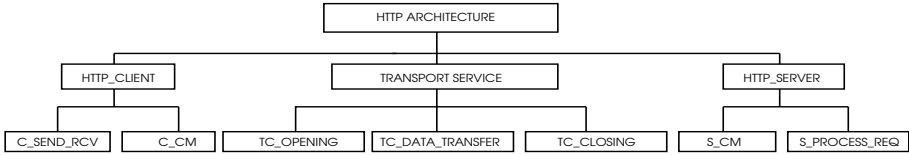


Fig. 1. HTTP CPN Model Hierarchy

have sub-modules involving transport connection management (CM) associated with a HTTP session, and the sending and receiving of HTTP messages. The transport service has 3 sub-modules including opening, closing and data transfer.

The model contains 14 places, 10 substitution transitions and 34 executable transitions.

### 3.3 Overview Model

The CPN module in Fig. 2 describes the HTTP architecture. It consists of 6 places and 3 substitution transitions that together model web browser interaction with a HTTP client, the transport service, and a HTTP server and its storage.

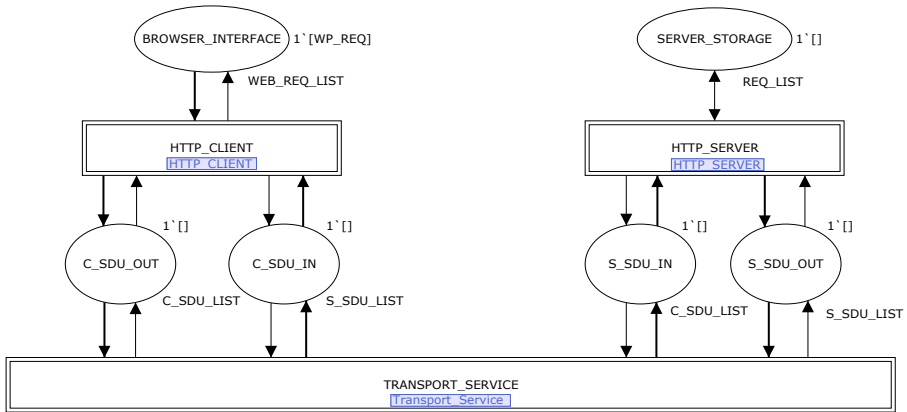


Fig. 2. HTTP Architecture

The BROWSER\_INTERFACE place holds user requests that are issued to the HTTP\_CLIENT. It is typed by WEB\_REQ\_LIST, a list of WEB\_REQUESTS as seen in Listing 1, line 2. In our model, WEB\_REQUESTS is just a singleton set, comprising WP\_REQ, which is a request for a web page.

**Listing 1.** Declarations for the HTTP CPN Model

```

1      (* ——WEB REQUESTS—— *)
2      colset WEB_REQUESTS = with WP_REQ;
3      var wr: WEB_REQUESTS;
4      colset WEB_REQ_LIST = list WEB_REQUESTS;
5      var wr_list: WEB_REQ_LIST;
6
7      (* ——HTTP MESSAGE—— *)
8      colset METHOD = with GET;
9      colset STATUS = with success;
10     colset HEADERFIELD = with Connection_close | none;
11     colset REQUEST = record method: METHOD * header: HEADERFIELD;
12     var rq: REQUEST;
13     colset RESPONSE = record scode: STATUS * header: HEADERFIELD;
14     var rs: RESPONSE;
15     colset REQ_LIST = list REQUEST;
16     var rq_list : REQ_LIST;
17
18     (* ——HTTP CLIENT—— *)
19     colset C_SESS_STATE = with C_TC_CLOSED | C_OPENING_TC | REQUESTING
20     | REQUEST_CLOSE | C_CLOSING_TC;
21     var cstate: C_SESS_STATE;
22     colset C_SDU = union ESTABLISH + CONFIRM + cdata: REQUEST
23     + C_RELEASE;
24     colset C_SDU_LIST = list C_SDU;
25     var cq, cq1: C_SDU_LIST;
26     colset NUMBER_OF_TRIES = int with 0 .. 2;
27     var num_tries: NUMBER_OF_TRIES;
28
29     (* ——HTTP SERVER—— *)
30     colset S_SESS_STATE = with S_TC_CLOSED | WFR | RESPONDING
31     | RESPONDING_RELEASE | S_CLOSING_TC;
32     var sstate: S_SESS_STATE;
33     colset S_SDU = union S_OPEN + CHECK + ACCEPT + sdata: RESPONSE
34     + S_RELEASE;
35     colset S_SDU_LIST = list S_SDU;
36     var sq, sq1: S_SDU_LIST;
37
38     (* ——TRANSPORT SERVICE—— *)
39     colset C_SAP_STATE = with C_CLOSED | OCP | C_DTR | C_NDS | C_NDR;
40     var sc: C_SAP_STATE;
41     colset S_SAP_STATE = with S_IDLE | ICP | S_DTR | S_NDS | S_NDR
42     | S_CLOSED;
43     var ss: S_SAP_STATE;

```

The `SERVER_STORAGE` place receives client requests for in-order processing. It is typed by `REQ_LIST` which is a list of `REQUEST`s as seen in Listing 1, line 11. Although it is not necessary to use lists, because when pipelining is not included there will only be one request at a time in `SERVER_STORAGE`, it is convenient for detecting that the place is empty (using the empty list), and preparing the model for pipelining. A `REQUEST` is modelled as a record containing a `METHOD` (line 8) and a `HEADERFIELD` (line 10). `METHOD` is an enumerated type containing the single HTTP/1.1 request method `GET`. Likewise `HEADERFIELD` is an enumerated type containing 2 entries: `Connection_close` and `none`. All header fields apart from the Host header are optional in any HTTP message. The Host header specifies the host and port number of



the resource being requested. As discussed in section 3.1, we only need to consider one client and one server, and hence we do not need to include the Host header. The inclusion of *none* (representing no header) allows us to include the Connection: close header or not, non-deterministically, when sending a message (as discussed in the section 3.4).

The HTTP\_CLIENT and HTTP\_SERVER interact with the transport service via the interface places C\_SDU\_OUT, C\_SDU\_IN, S\_SDU\_IN and S\_SDU\_OUT. C\_SDU\_OUT and S\_SDU\_IN are typed by C\_SDU\_LIST which is a list over C\_SDU as seen in Listing 1, lines 22 and 23. C\_SDU is a union of request messages (REQUEST) and the Service Data Units (SDU): ESTABLISH, CONFIRM and C\_RELEASE. The client issues an ESTABLISH to the transport service when it wishes to initiate a connection, likewise it issues a C\_RELEASE when it wishes to close its end of the connection. CONFIRM is used by the transport service when opening a connection. The other 2 places, C\_SDU\_IN and S\_SDU\_OUT are typed by S\_SDU\_LIST which is a list over S\_SDU as seen in Listing 1, lines 33 and 34. S\_SDU is a union of response messages (RESPONSE) and SDUs: ACCEPT, S\_RELEASE, S\_OPEN and CHECK. RESPONSE (Listing 1, line 13) is a record consisting of a STATUS and a HEADERFIELD. STATUS (Listing 1, line 9) is a singleton colour set consisting of success (meaning that the request was successful). ACCEPT is sent by the transport service to the client to indicate that it has accepted the connection. S\_RELEASE is sent by the server when it wishes to close its end of the connection. S\_OPEN is issued when the server is ready for connection requests, and CHECK is used by the transport service when establishing connections.

### 3.4 HTTP Client

Fig. 3 depicts the HTTP client. It consists of 2 substitution transitions and 3 additional places: PENDING\_REQUESTS, NUM\_TRIES and CLIENT\_STATE. PENDING\_REQUESTS, typed by REQ\_LIST (Listing 1, line 15), models HTTP requests that need to be sent to the server. It is required for retrying requests. Thus a request is only deleted when a response is received. Lists are used for the same reason they are used to type SERVER\_STORAGE. NUM\_TRIES stores the number of times a particular request is sent. CLIENT\_STATE models the state of the HTTP client during a session. While HTTP is generally considered a stateless protocol, state must be defined (see Listing 1, lines 19 and 20) in order to manage persistent connections as we now explain.

- C\_TC\_CLOSED. A transport connection does not exist. The client will open a connection if it receives a web request from its user.
- C\_OPENING\_TC. An open request (ESTABLISH) has been sent to the transport service and the client is waiting to receive an ACCEPT.
- REQUESTING. The transport connection has been established and the first request sent. The client may send requests and receive responses and initiate connection closure.
- REQUEST\_CLOSE. The client has sent its last request on the connection. It may still receive responses, or initiate connection closure.

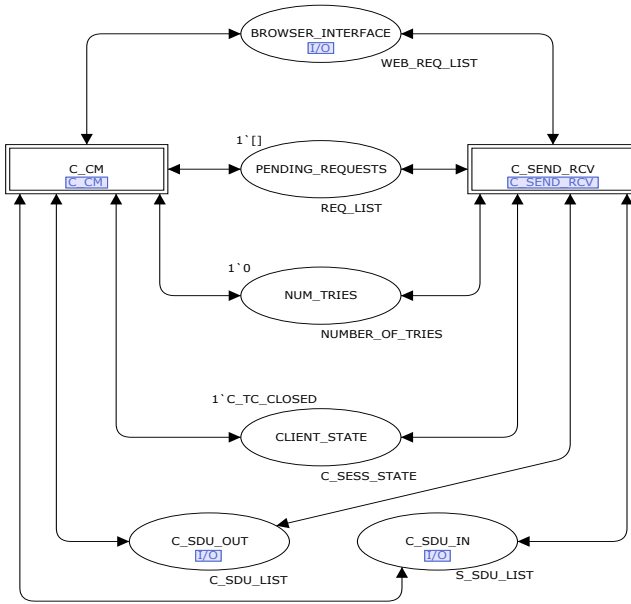


Fig. 3. HTTP Client

- C\_CLOSING\_TC. The client has initiated connection closure by sending a C\_RELEASE SDU to the server. It is unable to send requests but may still receive and process responses.

The C\_CM substitution transition is the client’s connection manager and is responsible for opening and closing connections. C\_SEND\_RCV sends and receives HTTP messages.

**C\_CM**, depicted in Fig. 4, has 6 executable transitions. Initially, the HTTP CLIENT is in the C\_TC\_CLOSED state (shown in Fig. 3) and a web request has been received by the BROWSER\_INTERFACE (see Fig. 2). Before a request can be sent a transport connection must be established. This is achieved by firing REQUEST\_TC, sending an ESTABLISH SDU to the transport service. REQUEST\_TC removes the web request and creates a HTTP request that contains a method (GET) and a header and stores it in PENDING\_REQUESTS. The header can be either Connection\_close or none to represent the use of non-persistent or persistent connections respectively. Firing this transition changes the state of the client to C\_OPENING\_TC. REQUEST\_TC will not fire if there is a request pending and hence a new connection will not be established for the next request until the current request has been processed (either by receiving a response or by declaring it failed).

Once the connection has been established by the transport service an ACCEPT SDU is received causing TC\_ESTABLISH\_AND\_SEND\_1st\_REQUEST

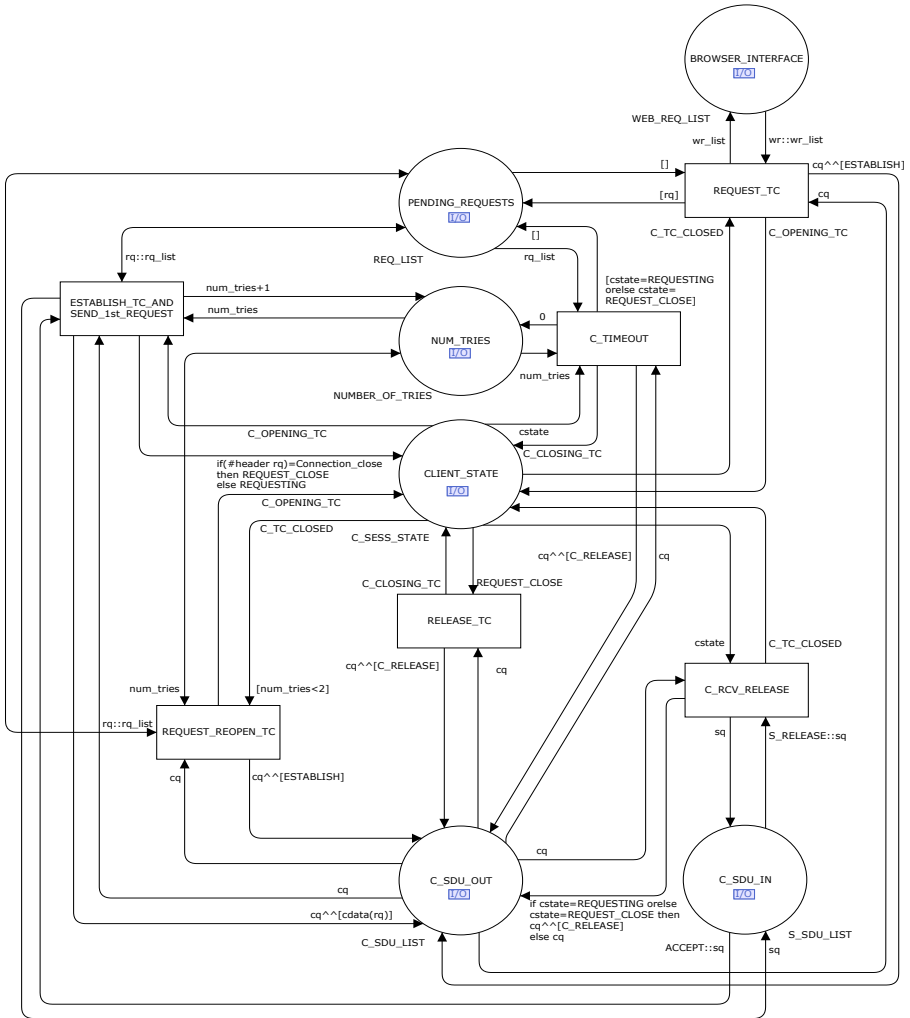


Fig. 4. HTTP Client Connection Manager

to fire. The firing of this transition causes 3 changes. Firstly a copy of the request in PENDING\_REQUESTS is sent to the transport service for delivery to the server. Secondly the number of attempts to send the request is incremented by 1. Thirdly the client changes state to REQUESTING (if the header was none) or REQUEST\_CLOSE (if the header was Connection:close).

The two transitions involved in the client initiating closing a connection are RELEASE\_TC and C\_TIMEOUT. RELEASE\_TC fires when the client is in REQUEST\_CLOSE, that is, after the client has sent a Connection: close header. RELEASE\_TC sends a C\_RELEASE and changes state to C\_CLOSING\_TC. C\_TIMEOUT occurs when the client wishes to issue a time-out on the

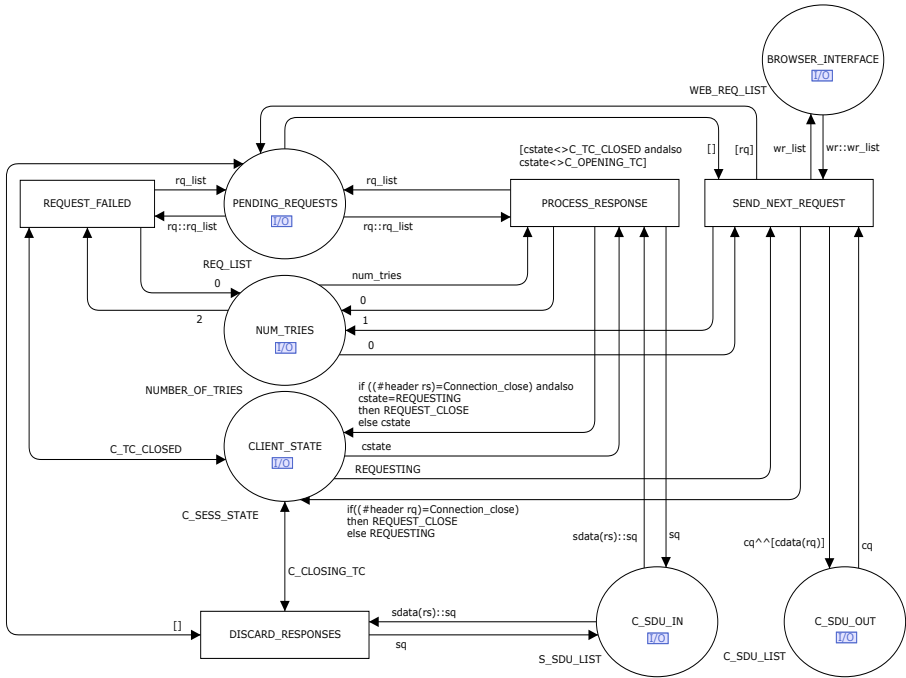


Fig. 5. HTTP Client Send and Receive

connection and discard any pending requests. This may occur in either the `REQUESTING` or `REQUEST_CLOSE` state. The occurrence of `C_TIMEOUT` sends a `C_RELEASE`, removes any requests in `PENDING_REQUESTS`, changes state to `C_CLOSING_TC` and resets `NUM_TRIES` to zero. Note that neither of these two transitions will occur if the client has already received a release from the server.

`CRCV_RELEASE` models the receipt of a `S_RELEASE` SDU. When in `REQUESTING` or `REQUEST_CLOSE`, the client has yet to close its end of the connection, so responds with a `C_RELEASE`. This is not required in the other states. On firing, the client's state becomes `C_TC_CLOSED`.

`REQUEST_REOPEN_TC` occurs when the connection has been released before a response has been received. It attempts to re-establish the connection so that the request can be resent. The guard on this transition ensures the request has only been sent once before, thus complying with [7]. On firing, this transition sends an `ESTABLISH` SDU to the transport service and changes state from `C_TC_CLOSED` to `C_OPENING_TC`.

`C_SEND_RCV` includes 4 executable transitions as shown in Fig. 5. Transition, `SEND_NEXT_REQUEST`, models the operation of persistent connections by sending a new request on an already established connection. This transition

is enabled when there is a web page request in the BROWSER\_INTERFACE, the client is in the REQUESTING state and there are no requests in PENDING\_REQUESTS. When the transition occurs, NUM\_TRIES is set to 1, the header for the request is chosen non-deterministically, a copy of the request is stored in PENDING\_REQUESTS, and the state changes to REQUEST\_CLOSE only if it is the last request to be sent on the connection.

When a request has been sent twice but the server has responded by closing the connection, the request is discarded and an error message reported to the user. This is modelled by REQUEST\_FAILED. PROCESS\_RESPONSE models the successful receipt of a response. It removes the initiating request from PENDING\_REQUESTS, sets NUM\_TRIES to zero ready for the sending of the next request, and remains in the same state, unless it receives the Connection: close header in the REQUESTING state, in which case it moves to REQUEST\_CLOSE to prevent any further requests being made.

DISCARD\_RESPONSES will fire when a response is received by the client that it is no longer interested in. This may occur after the client has timed-out (C\_TIMEOUT occurs) waiting for the response. DISCARD\_RESPONSES fires when the client is in C\_CLOSING\_TC, PENDING\_REQUESTS is empty and a response is received in S\_SDU\_IN.

### 3.5 The HTTP Server

The HTTP server model consists of 2 substitution transitions and 1 additional place, SERVER\_STATE, as shown in Fig. 6.

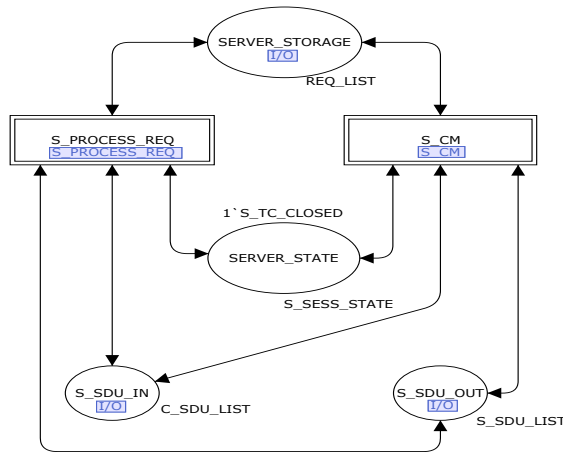


Fig. 6. HTTP Server

SERVER\_STATE is typed by S\_SESS\_STATE (Listing 11, lines 30-31). Like C\_SESS\_STATE, this place models the state of the server's HTTP session:

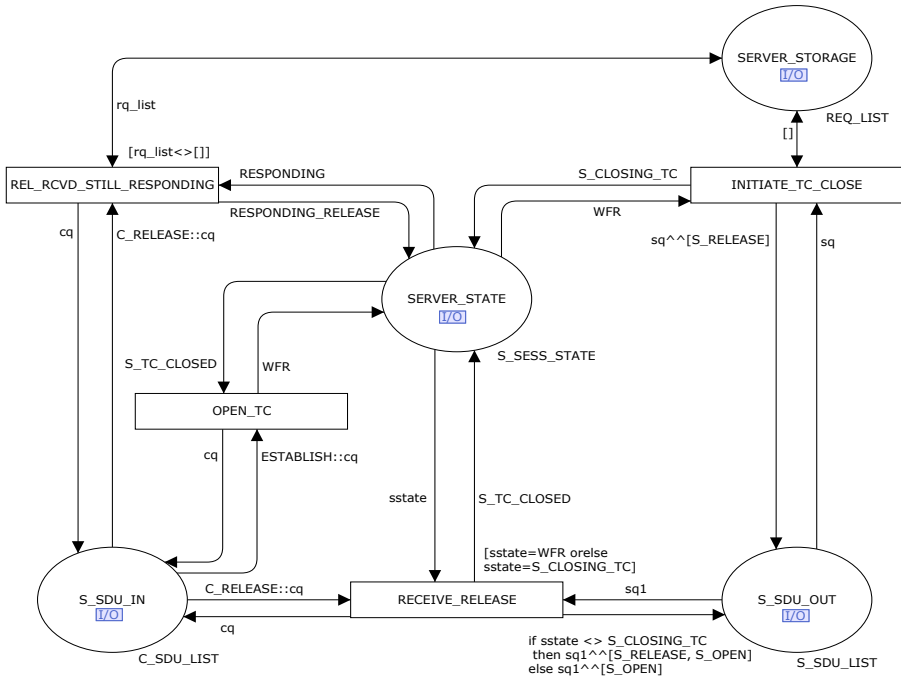


Fig. 7. Server Connection Manager

- S\_TC\_CLOSED. A transport connection does not exist. The HTTP server is ready to receive an ESTABLISH SDU from the transport service.
- WFR. In this state the server is waiting for a request, after a connection has been established with the client. It may initiate connection closure from this state or wait for the client to close first.
- RESPONDING. The server has received a request and is in the process of formulating the response.
- RESPONDING\_RELEASE. The server has received a C\_RELEASE while processing a request. This state is required to ensure that the server sends the requested response before closing its end of the connection.
- S\_CLOSING\_TC. In this state the server has closed its end of the transport connection. Hence any request that is received will not be responded to and is thus discarded.

S\_PROCESS\_REQ models the receipt of a client request and the sending of an appropriate response, while S\_CM models connection management.

S\_CM is shown in Fig. 7. It consists of 4 additional transitions. Initially the server is in the S\_TC\_CLOSED state, as specified in Fig. 6. When an ESTABLISH SDU is received, OPEN\_TC fires changing SERVER\_STATE to WFR

(waiting for request). The server may close the transport connection (INITIATE\_TC\_CLOSE) when it is in the WFR state. The firing of this transition changes SERVER\_STATE to S\_CLOSING\_TC. The server will not gracefully close while it is processing a request and hence the list in SERVER\_STORAGE must be empty. It may abort the session, but this is currently outside the scope of our initial model.

When the server receives a C\_RELEASE SDU, one of two transitions will fire depending on the state of the server. If in RESPONDING, the server is currently processing a request. Therefore the REL\_RCVD\_STILL\_RESPONDING transition will fire changing the server state to RESPONDING\_RELEASE. This will ensure the client's release request is responded to immediately after sending the last response (see transition SEND\_LAST\_RESPONSE in Fig. 8). If the server is in a state where it is immediately able to respond to the release (WFR or S\_CLOSING\_TC) RECEIVE\_RELEASE will occur changing the server state to S\_TC\_CLOSED and sending a S\_RELEASE SDU to the transport service, if one has not already been sent. The server will also indicate to the transport service that it is ready to receive further requests by issuing a S\_OPEN, corresponding to a passive open command in TCP.

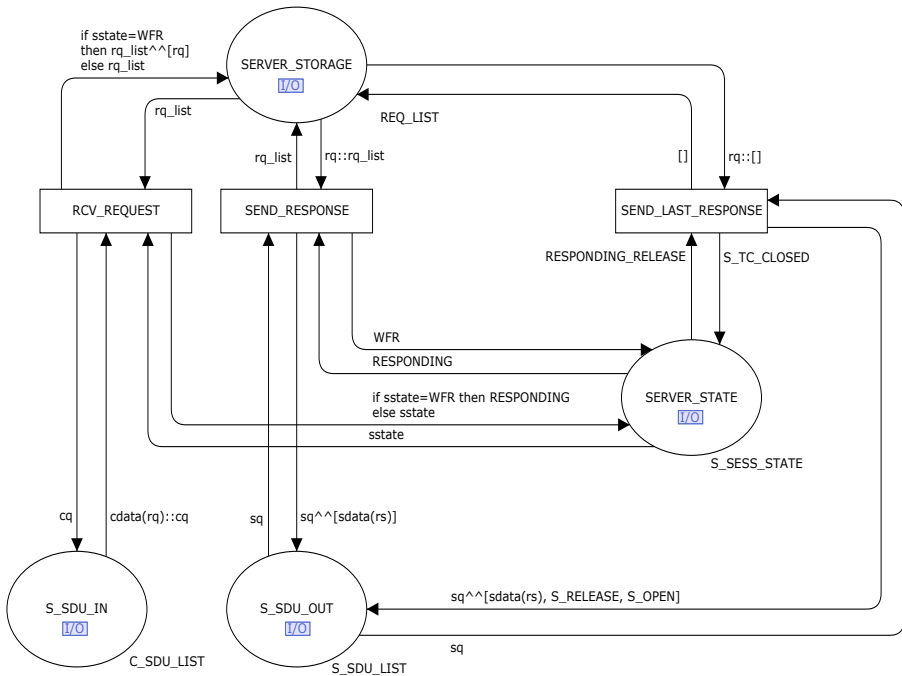


Fig. 8. Server Processing Requests

**S\_PROCESS\_REQUEST** is shown in Fig. 8. It has 3 executable transitions: `RCV_REQUEST`; `SEND_RESPONSE`; and `SEND_LAST_RESPONSE`. When a request is received (`S_SDU_IN`), `RCV_REQUEST` fires sending the request to place `SERVER_STORAGE` for processing when in state `WFR` and changing state to `RESPONDING`. Requests are discarded in any other state.

While in the `RESPONDING` state, the server may send its response to the client. This is modelled by `SEND_RESPONSE`. On firing, this transition removes the request from `SERVER_STORAGE`, modelling the finalisation of request processing, and sends the response, which may include the Connection: close header. The server moves back to `WFR`, waiting for another request.

`SEND_LAST_RESPONSE` occurs when the server is in state `RESPONDING_RELEASE` and there is a request in `SERVER_STORAGE`. This transition removes the last request from `SERVER_STORAGE` and sends the response followed immediately by `S_RELEASE` and `S_OPEN` (to indicate that it is ready to receive further requests) and changes the state of the server to `S_TC_CLOSED`.

### 3.6 Transport Service

The transport service is based on [3] which defines some of the terminology used in this section. Our transport service models the essential capabilities provided by TCP including connection establishment, reliable data transfer and graceful closure. It differs from [3] in several important ways. Firstly, we do not include the abort service, as HTTP [7] only mentions the use of gracefully closing connections. Secondly, we significantly modify the establishment service to make it better reflect TCP's operation. This involves including internal `CHECK` and `CONFIRM` messages to obtain TCP's sequencing of service primitives, i.e. that the client considers the connection to be established before the server does, which is not the case in [3]. Thirdly we enhance the service to allow for multiple serial connections, which requires the inclusion of a `TC_IDLE` interface transition, corresponding to the receipt of TCP's passive open command. Finally, we include explicit communication with the application (HTTP) via the `SDU` interface places.

The top level page for the transport service (TS) is shown in Fig. 9. It comprises 3 substitution transitions connected to the 4 `SDU` interface places and 2 new places, `SAPc` and `SAPs`, that record the states of the Transport Service Access Points (T-SAPs) for the client and server respectively. `TC_OPENING`, `TC_DATA_TRANSFER` and `TC_CLOSING` are the 3 substitution transitions that model the connection establishment, data transfer and connection release services and are described below.

**TC\_OPENING** is shown in Fig. 10. It includes 2 places, `C_S` and `S_C`, which represent order preserving channels between the client and the server and are typed by the colour sets `C_SDU_LIST` and `S_SDU_LIST` respectively. These channel places also occur in the other 2 substitution transition pages (see Figs. 11 and 12) using place fusion. Connection management is controlled by 2 state places:



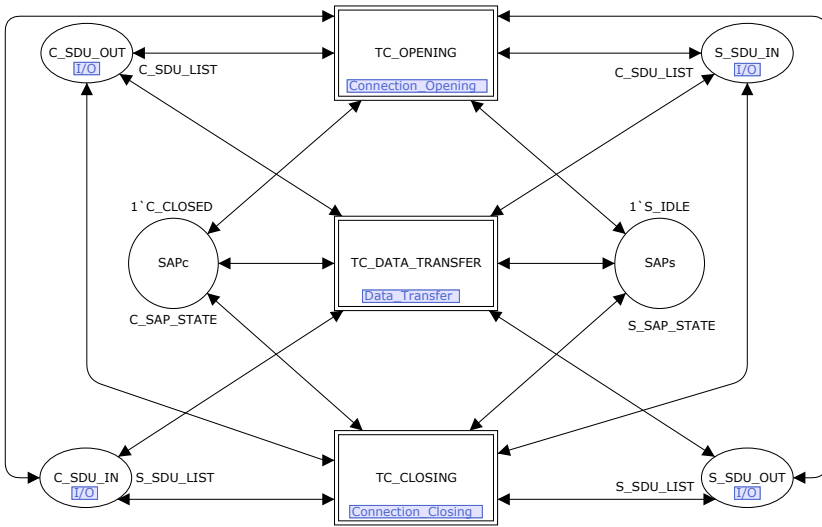


Fig. 9. Transport Service

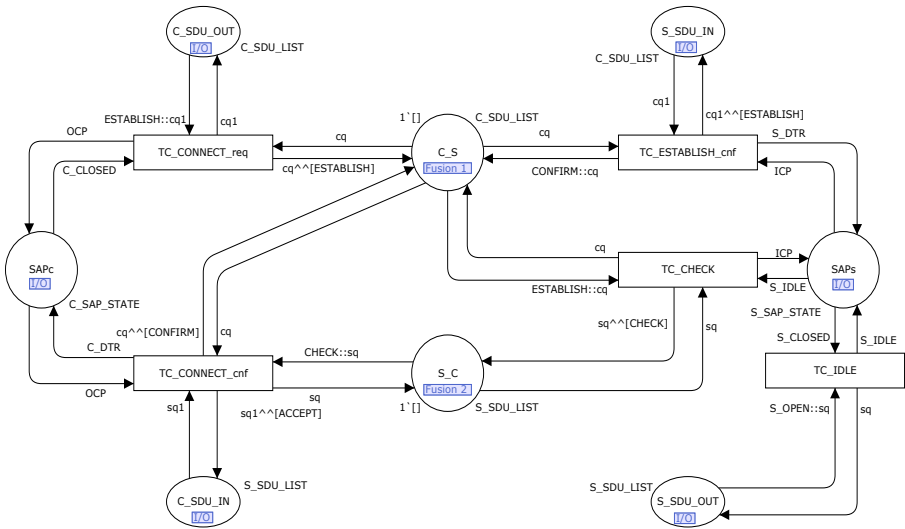


Fig. 10. Connection Establishment Service

SAPc, typed by C-SAP\_STATE (see Listing II, line 39), and SAPs, typed by S-SAP\_STATE (Listing II, lines 41 and 42). SAPc is initially in the CLOSED state, while SAPs is in IDLE, indicating that the TCP entity in the server is ready to establish connections. The HTTP client initiates connection establishment with an ESTABLISH SDU. On the occurrence of TC\_CONNECT\_req, the client

SAP changes state to Outgoing Connection Pending (OCP) and sends the ESTABLISH SDU (implemented in TCP by a SYN segment) over the channel. The server checks if this ESTABLISH SDU is legitimate (it may be an old duplicate) by returning a CHECK message to the client and moves to the Incoming Connection Pending (ICP) state. Given that the CHECK is OK, the TS client issues an ACCEPT SDU to HTTP, changes state to Data Transfer Ready (C\_DTR) and sends a CONFIRM message back to the server (TC\_CONNECT\_cnf). Once the server receives the CONFIRM (TC\_ESTABLISH\_cnf), it establishes the connection by sending an ESTABLISH SDU to HTTP and moving to S\_DTR.

As mentioned before, after the connection is closed at the server end, the HTTP server can indicate to the TS that it is ready for further requests by issuing a S\_OPEN command. This changes the SAPs state to S\_IDLE (TC\_IDLE), allowing the next connection to be established.

**TC\_DATA\_TRANSFER.** Requests and responses are transported in data SDUs as shown in Fig. 11. This is similar to 3, but does not include an urgent data service and is tuned for HTTP messages. It provides in-order lossless channels in both directions. More details are given in 3.

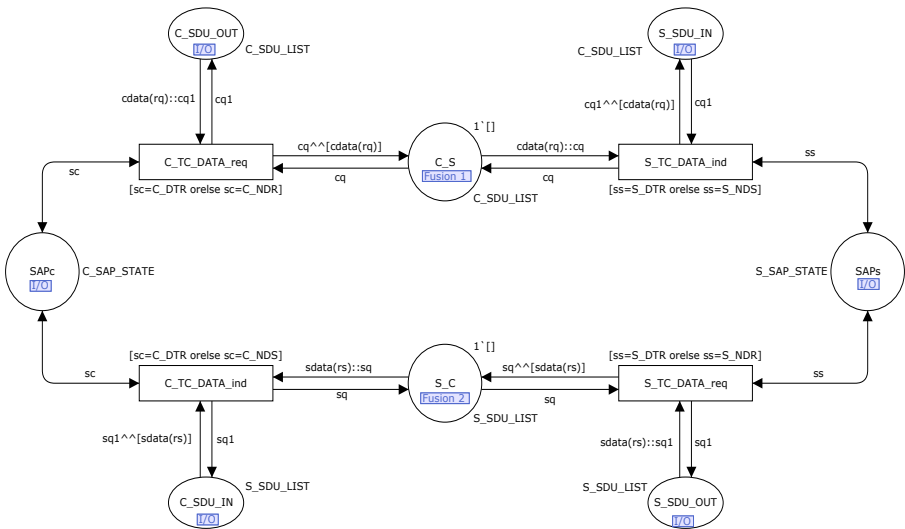


Fig. 11. Data Transfer Service

**TC\_CLOSING.** When either the client or the server wishes to close the transport connection, a RELEASE SDU is issued to the transport service as shown in Fig. 12. This page models graceful release where data in transit is not lost by the TS (see 3 for further details). Note that each side can close the connection independently and hence simultaneous release is included.

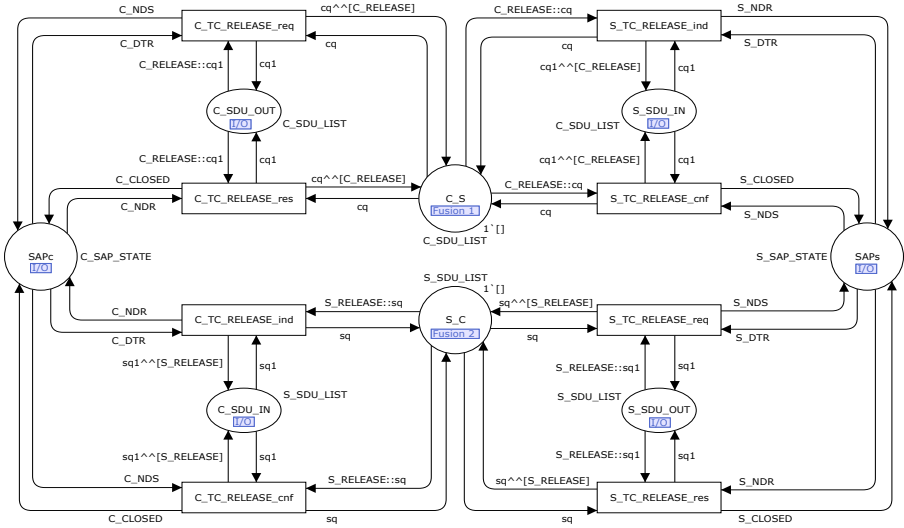


Fig. 12. Connection Release Service

### 3.7 Validation of the Model

We validated the model in a number of ways. Firstly, the model has been developed incrementally [2] and has now been through 7 iterations. At each stage we checked the model using single step simulation, partial state spaces and full state spaces. We also checked the model carefully against the HTTP specification [7], especially sections 6 and 8. The model was also checked independently by each author. When reviewing the 6th iteration we discovered that the client could reopen the connection after it had been released by itself. This was not intended. We therefore revised Fig. 4 to prevent this behaviour from occurring by removing the request if the activity timer expired.

When using full state spaces we ran a series of incremental tests [2]. Firstly we validated that our model was able to correctly establish a connection. Secondly we validated its ability to send requests and receive responses once the connection was established but without including connection closure. Thirdly we validated the closing procedures from the established state, by testing a) closure by the client only, b) closure by the server only and c) concurrent closure by both client and server. Finally we tested the full model. All these tests have allowed us to discover various modelling errors which were subsequently removed. For example, in an earlier version of the model we had included transitions that modelled the discarding of RELEASE SDUs when they were received in unexpected states. For all test cases these transitions were dead and were subsequently removed from the model.

Once these incremental tests were completed, we instrumented the model with an additional place in the HTTP client send and receive module (Fig. 5) (called RESPONSE\_NOTIFICATION) which stored responses and request failures in

the order they were received. This allowed us to validate that for each request there was a corresponding response for different numbers of web page requests. This was the case, except when the server closed prematurely, in which case the response was replaced by a failure notification.

Before analysing the complete model, to be discussed in section 4, we also analysed its behaviour for two special cases: a) non-persistent connections (when the Connection\_close header was included in each request and response) and b) persistent connections only (when the close header was not used). This led to smaller state spaces. Case a) resulted in state spaces that were approximately half the size of those presented in section 4, while for case b) they were about one third the size. This allowed the behaviour to be examined in more detail giving further confidence in the correctness of the model.

### 4 State Space Analysis Results

Once we were satisfied our model was working as expected, we ran a series of test cases investigating standard behavioural properties including liveness, home state and boundedness. In all tests, initially the client and server state places are CLOSED as is SAPc, SAPs is IDLE, NUM\_TRIES is 0 and SERVER\_STORAGE, PENDING\_REQUESTS, all interface SDU places and the transport service channels contain the empty list. We then varied the number of web page requests in BROWSER\_INTERFACE from 1 to 200. Selected results are given in Table 1, which summarises CPN Tool’s state space report.

Table 1. State Space Analysis Test Case Results

Property/Reqs	1	2	3	4	5	10	20	50	100	200
State Space Nodes	703	1455	2207	2959	3711	7471	14991	37551	75151	150351
State Space Arcs	1711	3546	5381	7216	9051	18226	36576	91626	183376	366876
Scc Graph Nodes	703	1455	2207	2959	3711	7471	14991	37551	75151	150351
Scc Graph Arcs	1711	3546	5381	7216	9051	18226	36576	91626	183376	366876
Dead Markings	1	1	1	1	1	1	1	1	1	1
Home Markings	1	1	1	1	1	1	1	1	1	1
Dead Transitions	1	None	None	None	None	None	None	None	None	None
Time (Seconds)	0	0	1	1	1	3	8	27	86	319

**State Space Statistics.** For all tests, and as shown in Table 1, we see that the number of the state space nodes (SSN) and state space arcs (SSA) are linear in the number of requests (r) and given by:

$$SSN(r) = 752r - 49, r \geq 1$$

$$SSA(r) = 1835r - 124, r \geq 1$$

This is expected as the markings of all other places re-occur for each length of the list in BROWSER\_INTERFACE.

**Cyclic Behaviour.** For all tests, the Strongly Connected Component graph is the same as the state space, signifying the absence of cyclic behaviour in the system. Cycles should not be present due to the number of attempts to send the same request on a new connection being different.

**Dead Markings and Home Markings.** As seen in Table 1, all tests reveal a single dead marking which is also a home marking. This verifies that the protocol does not livelock. The dead marking for each test case is the same and has the following properties:

- BROWSER\_INTERFACE contains the empty list, signifying that all requests have been processed by HTTP.
- Both the server and the client are closed as expected when the protocol terminates.
- SAPc is CLOSED and SAPs is IDLE and the channel places contain empty lists demonstrating that the transport connection has been closed correctly.
- SERVER\_STORAGE has an empty list. This shows that there are no requests still being processed in the server when the protocol terminates.
- NUM\_TRIES contains zero. This is expected as after all requests have been sent this place must be set to zero for the next request attempt.
- PENDING\_REQUESTS contains the empty list, therefore no requests have been made that have not received a response or failed.
- All SDU interface places contain empty lists as expected when the protocol terminates.

We thus conclude that the protocol terminates correctly.

**Dead Transitions.** Table 1 shows that test 1 has 1 dead transition, which is SEND\_NEXT\_REQUEST. As mentioned in Section 3, this only fires when there is a new request to send and the connection is open. So we expect it to be dead for test 1. Note that it is not dead for all other tests.

**Boundedness Properties.** The following boundedness properties hold for all test cases.

All places have best upper and lower integer bounds of 1 as expected because they are either state places, queueing places comprising a single list, or a counter (NUM\_TRIES). The best upper multi-set bounds for SERVER\_STORAGE and PENDING\_REQUESTS confirm that only 1 request may be pending (no pipelining occurs) as these places hold either a single request or an empty list. The upper multiset bounds for NUM\_TRIES reveal that it may hold the values 0, 1 or 2. This implies that only 2 attempts can be made to send a HTTP request, as required.

The best upper multi-set bounds reveal that C\_SDU\_OUT can have a maximum of 3 SDUs in its list, i.e. [cdata(rq), C\_RELEASE, ESTABLISH]. When the client sends an ESTABLISH to C\_SDU\_OUT, it cannot send any further SDUs until the ESTABLISH has been consumed and an ACCEPT received. Therefore

C\_SDU\_OUT is limited to a maximum of 3 SDUs. The C\_S channel can also hold a maximum of 3 messages given by [cdata(rq), C\_RELEASE, ESTABLISH] and [CONFIRM, cdata(rq), C\_RELEASE]. Once an ESTABLISH or a C\_RELEASE is issued by the HTTP client, it must be consumed and responded to by the transport service before the client can send any further SDUs. This is also the case for S\_SDU\_IN which can hold [ESTABLISH, cdata(rq), C\_RELEASE]. Before any further SDUs may be added, the C\_RELEASE must be consumed so that the HTTP server can issue a S\_OPEN to allow its transport service entity to return to IDLE, ready for the next connection.

S\_SDU\_OUT's list also has a maximum length of 3, which occurs in [cdata(rs), S\_RELEASE, S\_OPEN]. The server is unable to issue further SDUs until the S\_OPEN has been consumed by the transport service and it has received an ESTABLISH. The S\_C channel can hold up to 2 messages as seen in the list [cdata(rs), S\_RELEASE]. The S\_RELEASE must be consumed by the client and the connection re-established before any further SDUs can be added. The maximum length of the list in C\_SDU\_IN is 2 given in the lists [ACCEPT, S\_RELEASE] and [cdata(rs), S\_RELEASE]. The S\_RELEASE must be consumed by the client, the connection re-established and a request sent before any further SDUs can be added to this list. It is not possible to have a cdata(rs) and an ACCEPT in the list at the same time as the client must process the ACCEPT before a request is sent and responded to.

## 5 Conclusion

This paper has presented an initial CPN model of HTTP/1.1 [7] with emphasis on its essential features, including the use of persistent transport connections. In particular, the model includes the signalling of connection closure between client and server, client and server time-outs due to the connection being idle and the ability to re-open a connection when the server closes it before sending a response. The model includes a detailed transport service that allows for multiple serial connections to be established and released, using the graceful close mechanism. The full model is provided in the paper to allow others to experiment with it. The model is extensible, allowing different methods to be added if required. HTTP requires that after 2 attempts to obtain a response have failed due to premature closing of the connection by the server, then no further automatic attempts to re-open the connection should be made. Thus the value 2 is 'hard coded' into the model. However, the model can be easily extended to allow for any number of serial connections to be attempted, by using a symbolic constant 'MaxTries' instead, and initialising it in a 'val' statement in the declarations.

Our state space analyses have shown that the size of the state space is linear in the number of requests, and we provide expressions for the numbers of nodes and arcs in the state space as a function of the number of requests. All state spaces generated have just 1 dead marking that is also a home state. This shows that the protocol does not include any livelocks. We also conclude that the dead marking is an expected state corresponding to correct termination. The

state spaces demonstrate that when more than 1 request is made on the same connection, there are no dead transitions. We expect there to be 1 dead transition (SEND\_NEXT\_REQUEST) when just 1 request is made which is corroborated by the analysis. We have also investigated bounds on the interface buffers and transport service channels and discovered that they will not need to hold any more than 3 messages.

The model is useful because it captures the essential behaviour of HTTP/1.1, and is amenable to analysis which allows us to confirm important properties (e.g. liveness and termination) and to provide insights into buffer and channel bounds. However, the model also has limitations which relate to the assumptions made when creating the model. We would firstly like to relax the assumption regarding the server always accepting connections, and also to allow the check on ESTABLISH SDUs to fail. This requires enhancing the transport service. Secondly, we would like to include pipelining of requests. Future work will also consider relaxing the other assumptions made in section 3.1 and may include quantitative analysis similar to Wells et al [16].

**Acknowledgements.** The authors are grateful to the anonymous reviewers and the guest editor, Lars Kristensen, for constructive comments that have helped to improve the paper. We also gratefully acknowledge useful discussions with our colleagues Guy Gallasch and Lin Liu regarding this work.

## References

1. Berners-Lee, T., Fielding, R., Frystyk, H.: Hypertext Transfer Protocol – HTTP/1.0. RFC 1945 (May 1996)
2. Billington, J., Gallasch, G.E., Han, B.: A Coloured Petri Net Approach to Protocol Verification. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 210–290. Springer, Heidelberg (2004)
3. Billington, J., Han, B.: On Defining the Service Provided by TCP. In: Proc. 26th Australasian Computer Science Conference, Adelaide, Australia. Conferences in Research and Practice in Information Technology, vol. 16, pp. 129–138 (2003)
4. CPN Tools Online, <http://cpntools.org/>
5. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2068 (January 1997), replaced by RFC 2616
6. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (June 1999)
7. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T., Lafon, Y., Reschke, J.: HTTP/1.1, Part 1: URIs, Connections, and Message Parsing, draft-ietf-httpbis-p1-messaging-17 (October 2011)
8. Gvozdanovic, D., Simic, D., Vizek, U., Matijasevic, M., Valavanis, K., Huljenic, D.: Petri Net Based Modeling of Application Layer Traffic Characteristics. In: EUROCON 2001, International Conference on Trends in Communications, vol. 2, pp. 424–427 (2001)
9. Heidemann, J., Obraczka, K., Touch, J.: Modeling the Performance of HTTP Over Several Transport Protocols. IEEE/ACM Transactions on Networking 5(5), 616–630 (1997)

10. Hypertext Transfer Protocol Bis, <http://datatracker.ietf.org/wg/httpbis/charter/>
11. Jensen, K., Kristensen, L.: Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer (2009)
12. Krishnamurthy, B., Mogul, J., Kristol, D.: Key Differences Between HTTP/1.0 and HTTP/1.1. *Computer Networks* 31(11-16), 1737–1751 (1999)
13. Padmanabhan, V.: Improving HTTP Latency. *Computer Networks and ISDN Systems* 28(1-2), 25–35 (1995)
14. Postel, J.: Transmission Control Protocol. RFC 793 (Standard) (September 1981)
15. The Internet Engineering Task Force, <http://www.ietf.org>
16. Wells, L., Christensen, S., Kristensen, L., Mortensen, K.: Simulation Based Performance Analysis of Web Servers. In: *Proceedings of 9th International Workshop on Petri Nets and Performance Models*, Aachen, Germany, pp. 59–68 (2001)
17. Wu, H., Lin, X., Jiang, D.: A Petri Net Approach to Analyze the Effect of Persistent Connection on the Scalability of Web Services. In: *3rd International Conference on Communication Systems Software and Middleware*, Bangalore, India, pp. 267–270. IEEE (2008)



# Privacy Compliance Verification in Cryptographic Protocols

Suriadi Suriadi<sup>1</sup>, Chun Ouyang<sup>1,2</sup>, and Ernest Foo<sup>1</sup>

<sup>1</sup> Science and Engineering Faculty, Queensland University of Technology, Australia  
{s.suriadi,c.ouyang,e.foo}@qut.edu.au

<sup>2</sup> NICTA, Queensland Research Laboratory, Brisbane, Australia

**Abstract.** To provide privacy protection, cryptographic primitives are frequently applied to communication protocols in an open environment (e.g. the Internet). We call these protocols privacy enhancing protocols (PEPs) which constitute a class of cryptographic protocols. Proof of the security properties, in terms of the privacy compliance, of PEPs is desirable before they can be deployed. However, the traditional provable security approach, though well-established for proving the security of cryptographic primitives, is not applicable to PEPs. We apply the formal language of Coloured Petri Nets (CPNs) to construct an executable specification of a representative PEP, namely the Private Information Escrow Bound to Multiple Conditions Protocol (PIEMCP). Formal semantics of the CPN specification allow us to reason about various privacy properties of PIEMCP using state space analysis techniques. This investigation provides insights into the modelling and analysis of PEPs in general, and demonstrates the benefit of applying a CPN-based formal approach to the privacy compliance verification of PEPs.

## 1 Introduction

To achieve privacy-enhancing features, cryptographic primitives employed in a privacy enhancing protocol (PEP) normally have rich features (e.g. verifiable encryption) which extend the common encryption and signature capabilities often used in other types of *cryptographic protocols* (e.g. authentication protocols). For example, emulating the off-line anonymity afforded by cash transactions, a PEP can ensure that when a user purchases goods on-line, the on-line seller does not learn the identity of the user, but at the same time can be assured that the user's identity has been previously verified by a known trusted entity such that the identity can be revealed when needed. Recently, the Trusted Platform Module (TPM) technology - which provides secure hardware storage of cryptographic keys and implementation of cryptographic primitives - has also been used in PEPs [24].

An important issue in the design of applied cryptographic protocols, such as PEPs, is to ensure that they work correctly and do not contain errors that may weaken the security protection provided by the cryptographic primitives employed. While the *provable security* approach [14] is a widely-accepted method

used to prove the security properties of cryptographic primitives, it is not suitable to verify privacy compliance properties of PEPs. The main reason is that provable security emphasizes on proving the properties of a cryptographic algorithm (as evidenced by the use of ideal cryptographic models, such as the random oracle model - see [5,17]), while the privacy compliance properties of a PEP are *behavioural* and can be more naturally reasoned as properties of communication protocols. For example, one of the privacy properties verified in this paper is the *enforceable conditions* property (detailed in Sect. 4.3). This property is concerned with whether the messages exchanged between protocol entities are such that enough safeguards are included to ensure that a user's PII is *indeed* only revealed when certain conditions are satisfied, even in the presence of malicious behaviours from the entities involved. Consequently, attacks in PEPs normally arise from the existence of multi-party entities who attempt to exploit weaknesses in the *design* of a protocol, not directly at the algorithms of the cryptographic primitives employed. Furthermore, due to the lack of computer-aided tool support, the provable security approach is prone to errors [16].

Formal methods and languages allow the construction of unambiguous and precise models that can be analysed to identify errors and to verify correctness before implementation. Some of them, such as Coloured Petri Nets (CPNs) [13], provide a graphical modelling capability, and have tool support. The application of formal methods has been demonstrated to lead to reliable and trustworthy security protocols [2,8]. However, to the best of our knowledge, verification of PEPs using formal methods is yet to mature.

In this paper, we propose a CPN-based approach to construct a formal specification of a representative PEP, namely the Private Information Escrow Bound to Multiple Conditions Protocol (PIEMCP) [20], and to verify its privacy compliance properties.<sup>1</sup> CPNs are a widely-used formal language for system specification, design, simulation and verification. CPNs provide a graphical modelling language capable of expressing concurrency and system concepts at different levels of abstraction. With the support of CPN Tools, basic constructs of Petri nets are enriched with the functional programming language Standard ML (SML) [12] such that various high-level data type definition and functions can be defined and used in the model.

PIEMCP involves non-trivial multi-party communication (6 or more entities in general) and employs complex cryptographic primitives and TPM functionalities. The hierarchical structuring mechanism of CPNs supports a modular approach in capturing the behaviour of PIEMCP at different levels of abstraction. Using SML, the essential structures and behaviours of a wide variety of privacy-enhancing cryptographic primitives can be captured through a “black-box-style” abstraction such that only the essential features remain. By parameterising the protocol model with different types of attacks, a large number of attack scenarios are captured for analysis. The CPN model of PIEMCP is executable and can be analysed to verify the privacy properties of the protocol using the state spaces generated from the parameterised CPN model.

---

<sup>1</sup> This paper is an extension from our earlier work [22].

The contributions of this paper are two-fold. Firstly, it demonstrates the use of a CPN-based approach to model and verify the privacy properties of a PEP. To our knowledge, our work so far has been the only attempt at the formal verification of PEPs using CPNs. Secondly, the paper proposes several modelling and analysis approaches that have been (or can be) applied to other PEPs [19,24]. These can be used as preliminary guidelines for a general CPN-based approach for modelling and verification of PEPs.

This paper is structured as follows. Sect. 2 briefly explains PIEMCP. Sect. 3 proposes the modelling approach and describes selected parts of the CPN model of PIEMCP. Based on this CPN model, Sect. 4 details the verification of PIEMCP focusing on a set of privacy compliance properties. Related work is discussed in Sect. 5 with conclusions provided in Sect. 6. We assume that the reader has basic knowledge of CPNs. While we endeavour to explain the basic idea of PIEMCP, given the space constraints, prior knowledge in the area of information security and privacy is useful.

## 2 Overview of PIEMCP

PIEMCP [20] is used in a federated single-sign on (FSSO) environment whereby a user only has to authenticate once to an identity provider (IdP) to access services from multiple service providers (SPs). The entities involved are users, IdPs, SPs, and an anonymity revocation manager (ARM) or referees. An IdP assures SPs that although users are anonymous, when certain conditions are fulfilled, the users' identity can be revealed. A user's identity refers to a set of personally identifiable information (PII). Although the services that SPs provide can be delivered without the need of PII, they require the PII to be revealed by an ARM *or* referees when certain *conditions* are satisfied. An example of *conditions* would be "the user  $X$ 's PII should only be revealed to  $SP1$  if the user has posted some inflammatory/illegal messages/pictures on the forum".

PIEMCP consists of four stages, namely PII escrow (PE), key escrow (KE), multiple conditions (MC) binding, and revocation. An execution of the protocol involves two distinct sessions: the *escrow session* which consists of a sequential execution of the PE, KE and MC stages, and the *revocation session* which consists of an execution of the revocation stage. A user can run  $n$  escrow sessions, during which his/her PII is hidden (anonymous). At least one escrow session has to be completed before a revocation session can start. During the revocation session, the user's PII linked to a specific SP in a specific escrow session is revealed. For  $n$  escrow sessions, each with  $m$ -number of SPs, up to  $n \times m$  revocation sessions can be performed.

PIEMCP has two variants: one uses a trusted ARM for anonymity revocation while the other uses a group of referees (no ARM). While these two variants overlap to a certain degree, in this paper, we only consider the second variant of PIEMCP because it involves concurrent behaviours which highlight the relevance of CPN as the modelling language. Figure 1 depicts the *main* message exchanges between the entities of this protocol. For simplicity, a double-headed line is an abstraction of an exchange of one or more messages which collectively achieve a single cryptographic operation (normally a proof-of-knowledge operation).

The *PE stage* begins when a user requests a service from a service provider SP1. This triggers the agreement between user and SP1 of *conditions* (denoted *Cond1*) whose fulfillment allows the PII to be revealed to SP1, and a set of UCHVE parameters (explained in the ensuing paragraphs). SP1 then sends a message NT-PE-1 containing *Cond1* and UCHVE parameters to the IdP to escrow the user’s PII. The IdP contacts the user to obtain his encrypted PII (NT-PE-2). The user then encrypts his PII using a Verifiable Encryption (VE) scheme under a freshly generated public ( $\text{pubk}_{VE}$ ) and private ( $\text{privk}_{VE}$ ) key pair. The output of this VE operation is a ciphertext denoted as  $\text{VE}(\text{PII})_{\text{pub}_{VE}}$ . The user sends to the IdP, NT-PE-3 which comprises of  $\text{VE}(\text{PII})_{\text{pub}_{VE}}$  and  $\text{pubk}_{VE}$ . The user keeps  $\text{privk}_{VE}$  which is needed to decrypt  $\text{VE}(\text{PII})_{\text{pub}_{VE}}$ . Next, the user and the IdP engage in a cryptographic “proof-of-knowledge” (PK) protocol (NT-PE-4). This is to prove to the IdP that the VE ciphertext given correctly hides some *certified* PII without letting the IdP learn the value of the PII itself. We denote this operation as PKVE. The output of PKVE is an acceptance or rejection of  $\text{VE}(\text{PII})_{\text{pub}_{VE}}$ . A successful PKVE operation will lead to the IdP generating and sending a *pseudonym* to the user (NT-PE-5).

The *KE stage* is started after the user receives and stores the *pseudonym*. The IdP and the user now engage in another PK protocol - the Direct Anonymous Attestation (DAA) (NT-KE-1). This is to convince the IdP that the user is using a valid TPM device while concealing the identity of the TPM device. A

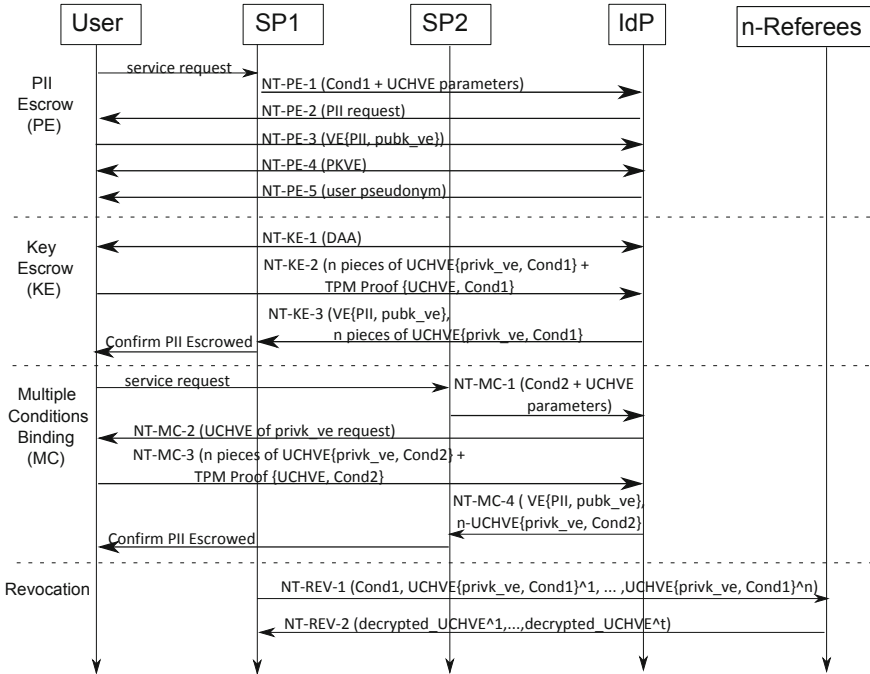


Fig. 1. Message exchanges within the four stages of PIEMCP

successful DAA prompts the user's TPM to generate (1) a universal custodian-hiding verifiable group encryption (UCHVE) of  $\text{priv}_{VE}$  under  $Cond1$  (denoted as  $\text{UCHVE}(\text{priv}_{VE})_{Cond1}^{1..n}$ ) and (2) a TPM proof of a correct UCHVE execution. In the rest of this paper, the generation of such UCHVE ciphertext with the TPM proof are represented by a TPM module, called TPM Module 2. The  $\text{UCHVE}(\text{priv}_{VE})_{Cond1}^{1..n}$  actually is a group of  $n$  distinct ciphertext pieces which can later be given to a group of  $n$  referees among whom there are  $t$  ( $t \leq n$ ) *designated* referees. Only the designated referees can decrypt their respective ciphertext pieces. At least  $k$  ( $k \leq t$ ) decrypted pieces are required to recover the VE private key (i.e.  $k$  is the *threshold* value). Both  $\text{UCHVE}(\text{priv}_{VE})_{Cond1}^{1..n}$  and the corresponding TPM proof are sent to the IdP in NT-KE-2. The IdP then verifies the proof and if correct, prepares a response NT-KE-3 to SP2 which includes the  $\text{VE}(\text{PII})_{pub_{VE}}$  and  $\text{UCHVE}(\text{priv}_{VE})_{Cond1}^{1..n}$ . Having obtained  $\text{VE}(\text{PII})_{pub_{VE}}$  and  $\text{UCHVE}(\text{priv}_{VE})_{Cond1}^{1..n}$ , SP1 now can, with the help of referees, recover the user's PII when  $Cond1$  is fulfilled, but *cannot* do so until that time. SP1 then confirms to the user that his/her PII has been escrowed successfully.

In the *MC stage*, the user goes to another service provider SP2 to request service. This time SP2 requests the IdP to escrow the  $\text{priv}_{VE}$  in NT-MC-1 under *conditions*  $Cond2$  (i.e.  $Cond1 \neq Cond2$ ) and UCHVE parameters that have been agreed between user and SP2. The IdP requests the user's TPM to produce  $\text{UCHVE}(\text{priv}_{VE})_{Cond2}^{1..n}$  (that is, a *new* UCHVE encryption of  $\text{priv}_{VE}$  under  $Cond2$ ) and the associated TPM proof (NT-MC-2). The user then performs the requested operation and sends  $\text{UCHVE}(\text{priv}_{VE})_{Cond2}^{1..n}$  with the corresponding TPM proof in NT-MC-3 (in other words, the TPM Module 2 is executed again). The IdP verifies the TPM proof of  $\text{UCHVE}(\text{priv}_{VE})_{Cond2}^{1..n}$ , and if correct, prepares a response NT-MC-4 to SP2 which includes  $\text{VE}(\text{PII})_{pub_{VE}}$  and  $\text{UCHVE}(\text{priv}_{VE})_{Cond2}^{1..n}$ . Similar to SP1, SP2 now has the necessary ciphertexts which, with referees' help, can reveal the user's PII when  $Cond2$  are satisfied, but *cannot* do so yet at this point. SP2 then confirms to the user that his/her PII has been escrowed successfully.

For any subsequent service providers that the user contacts within an escrow session, the user and the service provider only need to execute the MC stage activities. Therefore, the MC stage activities are specific to the *second and subsequent* service providers visited by the user, while the PE and KE stage activities are specific to only the *first* SP visited by the user.

The *revocation stage* is executed when the agreed conditions are satisfied and when a user has completed at least one escrow session. Assuming that  $Cond1$  is satisfied, SP1 sends a revocation request NT-REV-1<sub>1..n</sub> to each of the  $n$  referees. For each referee  $r_i$ , the message NT-REV-1 <sub>$i$</sub>  consists of  $\text{UCHVE}(\text{priv}_{VE})_{Cond1}^i$  and  $Cond1$ . Each referee then checks if  $Cond1$  is fulfilled, and if so, the referee tries to decrypt the given ciphertext piece. Only the designated referees can decrypt the ciphertext pieces. If decryption is successful, each designated referee sends the decrypted data NT-REV-2<sub>1..t</sub> to SP1. When  $k$  ( $k \leq t$ ) or more

decrypted data are received, SP1 can recover  $\text{priv}_{VE}$ , and subsequently decrypt  $\text{VE}(\text{PII})_{\text{pub}_{VE}}$  to recover the user’s PII.

Above we described the normal execution of PIEMCP (i.e. without attacks). However, each of the parties involved may behave maliciously resulting in various attack scenarios. The design goal of PIEMCP is to achieve the required security behaviour with and without considering the attacks. In the next section, a CPN model of PIEMCP is presented which can be configured to capture both normal scenario and attack scenarios.

### 3 CPN Model of PIEMCP

#### 3.1 Modelling Approach

We introduce two modelling approaches specific to PEPs: the *Cryptographic Primitive Abstraction* and the *Model Parameterization with Attacks*. We have also captured the *TPM Provable Execution* modelling approach in our model but it is not described in this paper (for details, see [18, Section F.2]).

*Cryptographic Primitive Abstraction.* To capture complex cryptographic behaviours, we firstly model the representation of a ciphertext as a CPN colour set, and then capture its operations using SML functions. This approach is flexible and inclusive as virtually any type of cryptographic primitives can be captured. The CPN `record` type can encode the necessary information to represent a primitive properly, and SML can be used to *simulate* the operations. The cryptographic operations captured by SML functions are “symbolic” rather than an actual operation. For example, an encryption function defined in our approach *does not* perform the actual encryption, rather, we impose certain restriction on what the recipient of this ciphertext can do with this message (such as not being able to extract the message without having a correct decryption key).

Our approach of expressing cryptographic operations as functions promotes reuse which leads to a cleaner and more concise model. A disadvantage of this approach is that the modeler has to consciously follow the restriction imposed on cryptographic messages produced by these functions as CPN Tools does not automatically enforce these restrictions. In Sect. 3.2, we demonstrate this approach by modelling a VE ciphertext and a zero-knowledge operation (PKVE). The complexity of UCHVE ciphertext prevents us from describing it due to space constraint; however, it is available in the referenced thesis [18, pp. 196].

We also propose a technique to capture the commonly-used message signing and verification operations. We define a CPN colour set for the message to be signed, followed by a definition of its signature. A signed message is a pair consisting of the message and its signature. The verification of a signed message upon the receipt of the message is *enforced* within a transition guard. If the signature verification fails, the message integrity and/or authenticity are compromised. As a result, the guard returns a false value, thus preventing any further processing on the message – a so-called fail-stop mechanism.

*Model Parameterization with Attacks.* We propose the parameterisation approach to modelling attacks such that one or more attacks can be switched on or off depending on the environmental assumptions. We scope our work to only consider malicious insiders - which we consider to be a greater concern in PEPs. The Dolev-Yao intruder model [11] (which represents an external intruder), while relevant, is not considered in this paper. There are many attacks that a malicious insider could launch. Creating a new model to capture each type of attack (existing or new) scales poorly as the number of attacks grows. Parameterisation allows the re-use of the existing model while allowing it to behave differently according to the attacks being set - virtually allowing thousands of possible attack scenarios to be captured. We have modelled 14 types of attacks using 14 parameters, each with a boolean value of “true” (on) or “false” (off), which theoretically can capture  $2^{14}$  possible attack scenarios. The attack parameters are then referred to in the arc-inscriptions, transition guards, or transition code-regions. Note that although it is not necessary to consider all attack scenarios (see Sect. 4), the ability of our model to capture a comprehensive attack scenario may be exploited in the future to allow other types of analysis.

The advantage of this approach is that we do not have to change the model (e.g. adding/deleting transitions) to obtain different behaviours. The disadvantage however is that it may reduce the readability of the model due to the addition of parameter inscriptions (such as `if/else` statements) and may make model debugging more difficult as the number of attacks increases. This approach risks the introduction of complexity during model validation in comparison to having two separate models (one without attacks and one with attacks). However, this risk is somewhat compensated with an easier model maintenance practice: changes to the basic behaviour of the model only need to be applied once to the model and its effect will apply to all other parameterized behaviours. This is not the case when we have two or more separate models.

### 3.2 Model Description

The PIEMCP model is a hierarchical CPN consisting of 4 levels: 1 main (top-level) page, 5 second-level pages, 13 third-level pages, and 1 fourth-level page. As detailed in Sect. 2, a sequential execution of the PE, the KE, and the MC stage forms one *escrow session*. For simplicity, our model covers a minimum full protocol execution: the PIEMCP CPN model allows sequential execution of a certain number of escrow session (determined by the model parameter `session`) followed by one revocation session. Note that it is possible for both the escrow and revocation session to run in parallel, however, modelling such concurrency does not capture any additional behaviour of the protocol as these two sessions are assumed to be distinct, i.e. they do not interfere with each other [2]. Therefore, our model currently does not capture this parallelism.

---

<sup>2</sup> While it may be interesting to model and analyze the security properties of our protocol in the presence of parallel *escrow* and *revocation* sessions, we reserve this for future work.

The revocation page can be executed after the completion of at least *one* escrow session. Selected parts of the PIEMCP CPN model are described to demonstrate the modelling approaches detailed in Sect. B.1. Relevant CPN colour sets definitions and functions are provided in Table 1.

*Main page.* Figure 2 shows the top-level page which captures the protocol entities (represented as *substitution transitions*) and the communication channels between any two entities (as *places* with thick lines). Since these communication channels represent application-layer communication, we assume the existence of no errors commonly associated with lower-layer communication channels (such as data loss). While it may be possible to fold the three SP1\_REFEREE channels into one, we decided to split them into three to improve readability (i.e. to explicitly separate distinct logical communication channels between entities).

As explained in Sect. 2, the PE and KE stage activities are specific to the *first* service provider (e.g. SP1) visited by the user, while the MC stage activities are

**Table 1.** Colour set definition

---

```

1  CRYPTOGRAPHIC COLOUR SETS DEFINITION
2  =====
3  colset K_PUB_VE = INT;
4  colset K_PRIV_VE = INT;
5  colset K_SIGN_GEN = INT;
6  colset PII = STRING;
7  colset LABEL = STRING;
8  colset PROVABILITY = BOOL;
9  colset COMMITMENT_PII = record message:PII * random:RANDOM;
10 colset CIPHER_VE_PII = record message:PII * key:K_PUB_VE * label:LABEL*provable:PROVABILITY;
11
12 PIEMCP MESSAGES DEFINITION
13 =====
14 colset SP_REQ = record genCond:STRING * conditions1:STRING * <other fields omitted>
15 colset SP_REQ_SIG = record message:SP_REQ * key:K_SIGN_GEN;
16 colset SIGNED_SP_REQ = record message:SP_REQ * signat:SP_REQ_SIG;
17 colset SIGNATURE_GEN = record message:MSG * key:K_SIGN_GEN * provable: PROVABILITY;
18 colset SIGNED_MSG = record message:MSG * signat:SIGNATURE_GEN;
19 colset DEC_REQ = record conditions:LABEL * uchvePiece:CIPHER_UCHVE_KVE_PIECE;
20 colset DEC_REQ_SIGNATURE = record message:DEC_REQ * key:K_SIGN_GEN * provable:BOOL;
21 colset SIGNED_DEC_REQ = record message:DEC_REQ * signat:DEC_REQ_SIGNATURE;
22 colset DECRYPT_OUTPUT = product BOOL * MSG;
23
24 COMMUNICATION CHANNEL DEFINITION
25 =====
26 colset IDP_SP1 = union msgEscrow:SIGNED_SP_REQ + signedSPResponse1:SIGNED_SP_RESPONSE;
27
28 FUNCTIONS and PARAMETERS
29 =====
30 fun veKeysRel(privKey:K_PRIV_VE, pubKey:K_PUB_VE)= if privKey=pubKey then true else false;
31 fun veEnc(msg:PII, pubKey:K_PUB_VE,cond1:LABEL)=
32   {message=msg,key=pubKey,label=cond1,provable=true};
33 fun decVE(key:K_PRIV_VE, cipherVE:CIPHER_VE_PII, cond1:LABEL)=
34   if veKeysRel(key, #key(cipherVE)) andalso cond1 = (#label(cipherVE)) then
35     1'(true, #message(cipherVE)) else 1'(false, "");
36 val condActually = true;
37 val session=2;
38 val threshold=2;
39 val honestRef=1;
40 val toRevoke=1;

```

---



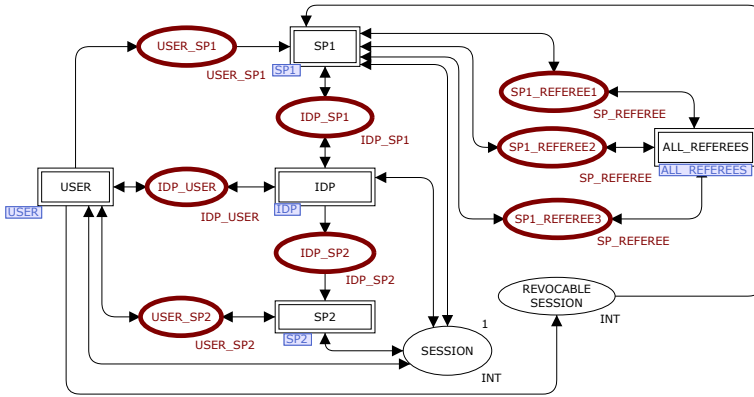


Fig. 2. The PIEMCP CPN – Top-level page

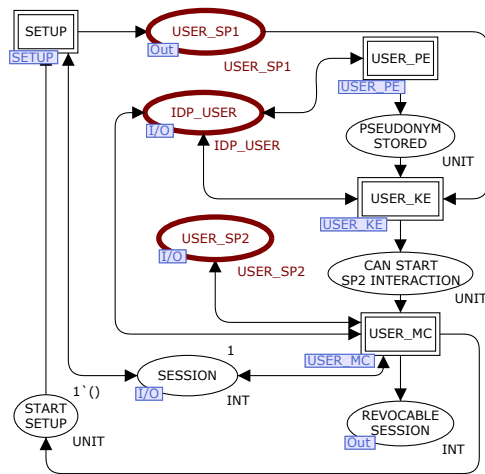


Fig. 3. The PIEMCP CPN – USER second-level page

specific to the *second and subsequent* service providers (e.g. SP2). Therefore, we decided to separate the modelling of SP activities into two substitution transitions (SP1 and SP2) due to their *non-overlapping* activities.

*Second-level Pages.* The multi-stage operation of PIEMCP is detailed on the second-level pages for each of the entities. For example, the second-level page for the user is shown in Figure 3 whereby the sequential execution of the PE, KE (with SP1), and MC (with SP2) stages is modelled. The completion of the MC stage signals the completion of one escrow session and may trigger the execution of another escrow session by marking the place *START\_SETUP*. The determination of whether or not to execute another escrow depends on the model parameter *session* and is explained in detail in the extended version of this paper

[21, Appendix A.2]. Furthermore, the completion of an escrow session also means that, theoretically, the user PII for that particular escrow session is now revocable. However, as explained in the beginning of Sect. 3.2, our model will only allow the execution of the revocation stage after the completion of a certain number of escrow sessions as determined by the parameter `session`. Similar second-level pages for IdP, SP1, and SP2 have also been modelled (detailed in the extended version of this paper [21, Appendix A.1]).

*Third-level Pages.* The details of the IdP's, SPs', and user's activities during the PE, KE, MC, and revocation stages are provided on the third-level pages. Four examples of such pages are provided in Figures 4 to 7. Note that the main transitions in Figures 4 to 6 have been annotated with a number (located at the bottom-right corner of each of the transitions) indicating the normal order (i.e. all attack parameters switched off) in which they occur.

Figure 4 depicts the model of the IdP's activities during the PE stage. This page demonstrates the message signing and verification approach. The input arc to the transition `IDP_VERIFYES_SP1_REQ_AND_STARTS_PII_ESCROW` (Figure 4, top centre) contains a variable `escrowReqSig` of colour set `SIGNED_SP_REQ` within the union colour set `IDP_SP1` (see Table 1 lines 14, 15 and 26). The `escrowReqSig` variable represents an SP1-signed message whose content is the *conditions* string. This message is equivalent to message NT-PE-1 in Figure 1. As the IdP receives this message, it verifies the signature validity which is captured in the transition guard of the same transition. If the guard expression (`verifyEscrowReqSig(escrowReqSig)`) returns true, the signature is valid and the transition is enabled, allowing the IdP to contact the user to proceed with the PE stage.

Figure 5 depicts the details of the user's activities. This page models the generation of necessary cryptographic data by the user. Here, we demonstrate how complex cryptographic primitive behaviours can be modelled. The VE ciphertext is defined as the colour set `CIPHER_VE_PII` (see Table 1 line 10) which is a record consisting of four fields: the message itself, the public encryption key, the label under which the message is encrypted, and the provability property. A provable ciphertext means that the recipient of the ciphertext can validate that the received ciphertext correctly encrypts some claimed value (in this case the user's PII) without the recipient learning the value of either the PII itself or the decryption key. We consider the `message` field inside a colour set that represents a ciphertext to be *unreadable*. The model in Figure 5 captures the generation of a VE ciphertext of PII, the result of which will trigger the placement of a token in the `PII_VE_CIPHER` place (Figure 5, top-right).

The VE operations, including the encryption and decryption operations, are captured as functions (see Table 1 lines 30-35). As stated in Sect. 3.1, our encryption operation does not perform the actual message encryption and decryption operation. Rather, these operations are abstracted into two functions – `veEnc` and `decVE` – and an auxiliary function `veKeysRel`. The function `veEnc` transforms the main inputs for a VE encryption algorithm and outputs a token typed by the colour set `CIPHER_VE_PII`. The decryption operation (1) takes as input a representation of a VE private key and the ciphertext to be decrypted, (2)



the first element indicates the success/failure of the decryption, and the second element contains the decrypted message (in case of success).

Next, the user sends the NT-PE-3 message (containing the VE ciphertext of PII, and the public VE key) to the IdP - represented by the transition U\_SENDS\_PII\_ESCROW\_DATA (Figure 5, middle-right). When the IdP receives this message, the PKVE operation is triggered (NT-PE-4). Here, we demonstrate how a complex zero-knowledge proof protocol like PKVE is modelled in CPN. We break this operation into three transitions across the USER\_PE and the IDP\_PE pages (indicated with grey-filled transitions): the START\_PKVE transition triggered by IdP to signal the user the start of such a protocol (Figure 4, centre), the GENERATE\_PKVE\_PROOF transition executed on the user side to generate the required PKVE proof data (Figure 5, middle-bottom), and the VERIFY\_PKVE\_PROOF transition executed by the IdP to verify the given PKVE proof data (Figure 4, middle-left). The result of PKVE is represented by the place PKVE\_RESULT. The essential processing required by the IdP to verify the correctness of the proof is captured by the function `pkve`<sup>3</sup>, which is invoked in the arc inscription from the transition VERIFY\_PKVE\_PROOF to the place PKVE\_RESULT. Upon a successful PKVE, the IdP generates a pseudonym and sends it to the user to be used for that particular session.

Figure 5 also shows the attack parameterisation approach mentioned in Sect. 3.1. The USER\_ATTACK2 parameter (see the output arc inscription from the transition U\_SENDS\_PII\_ESCROW\_DATA to the place IDP\_USER around the centre of Figure 3) depicts the behaviour of a malicious user who falsifies/gives an incorrect VE public key to the IdP in the NT-PE-2 message. Thus, when USER\_ATTACK2 is set to “true”, the user will send an incorrect VE public key value (represented as “0”), otherwise, a correct value is sent.

Figure 6 shows the model for the revocation stage. The first transition (top) SP1\_RETRIEVES\_FULFILLED\_CONDITIONS is only enabled if the total number of executed escrow sessions (represented by the variable `counter`) is greater than the value of the parameter `session`. The completion of an escrow session increases the value of `counter` by one, and as a result, the completion of `session`-number of escrow sessions will result in the value of `counter` to be `session+1`. Hence, the guard to the above transition essentially ensures that there must be at least `x`-number of escrow sessions completed before the revocation stage can start (whereby `x` is determined by the `session` parameter).

Next, SP1 firstly retrieves the condition string of a completed escrow session which is deemed to have been fulfilled (the model parameter `toRevoke` - Table 1 line 40 - determines the corresponding completed escrow session). This data is retrieved from the stored session data executed through the code segment associated with the transition SP1\_RETRIEVES\_FULFILLED\_CONDITIONS (which is not shown in Figure 6). Note that the session data were previously stored by SP1 at the completion of the KE stage (details available in the extended version of this paper [21, Appendix A.4]). Figure 6 also demonstrates the parameterisation of another attack parameter SP\_ATTACK3 whereby a service provider may

<sup>3</sup> Definition of this function is available in the referenced thesis [18, pp. 305].

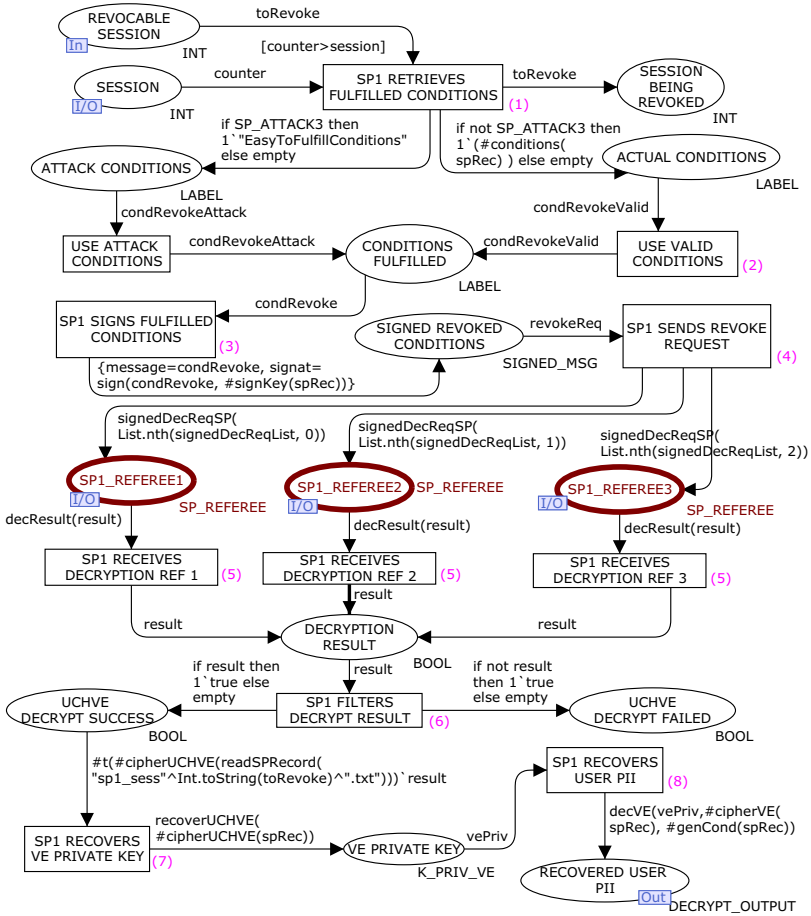


Fig. 6. The PIEMCP CPN – SP1-REV page (revocation stage initiated by SP1)

attempt to revoke a user’s PII by including the trivial “easy to fulfill” condition string (which is different from what was agreed with the user previously during the escrow stage). This is captured in the two output arcs from the transition SP1\_RETRIEVES\_FULFILLED\_CONDITIONS. The setting of SP\_ATTACK3 will mark the place ATTACK\_CONDITIONS (Figure 6, top left) with the “easy to fulfill” condition string and no token will be sent to the place ACTUAL\_CONDITIONS (Figure 6, top right). In the absence of this attack, the place ACTUAL\_CONDITIONS will be marked with the actual condition string as read from the session data file and the place ATTACK\_CONDITIONS will not have any token.

SP1 then sends a PII revocation request to all referees, modelled by the transition SP1\_SENDS\_REVOKE\_REQUEST (Figure 6, middle right), by sending the condition string and the UCHVE pieces which are retrieved by reading the session data stored previously (achieved through code-region not shown in Figure 6). The

details of each referee’s model are described later in this section; at this stage, when the SP1 receives the decrypted UCHVE pieces, it will then attempt to recover the VE private key. This is captured by the input arc to the transition `SP1_RECOVERS_VE_PRIVATE_KEY` (Figure 6, bottom left). This arc inscription requires  $t$  (representing the threshold value) successfully decrypted pieces of the UCHVE ciphertext by referees before the message (i.e. the VE private key) can be decrypted. This page also demonstrates how CPN can be used to capture the concurrent processing required (amongst the referees) during the UCHVE decryption process. The combination of the modelling approach used on this page and the referee pages (described in the ensuing text) therefore demonstrates how we can capture a *threshold decryption* process using CPN.

The details of the referees’ model are described below. Figure 7 (left-hand side) shows the detailed referees’ activities during the revocation stage. Since the operations of each referee are the same, we decided to create one `REFEREE` page which can be instantiated for individual referees. An example of a `REFEREE` page instance is shown in Figure 7 (right-hand side). To capture the different runtime behaviour of individual referees, we parameterise each `REFEREE` page instance (on the `ALL_REFEREE` page) with two main parameters: the referee number ( $ID$ ) and the condition fulfillment decision (the `REFEREE_NUMBER_i` and `COND_FULFILLMENT_i` places respectively, where  $i=\{1,2,3\}$ ). The later parameter is used to capture the (non-)malicious behaviour of a referee and is determined through the setting of its initial value. For example, the initial marking of `COND_FULFILLMENT_1` (Figure 7, top left) states that when all attack parameters which affects the referees’

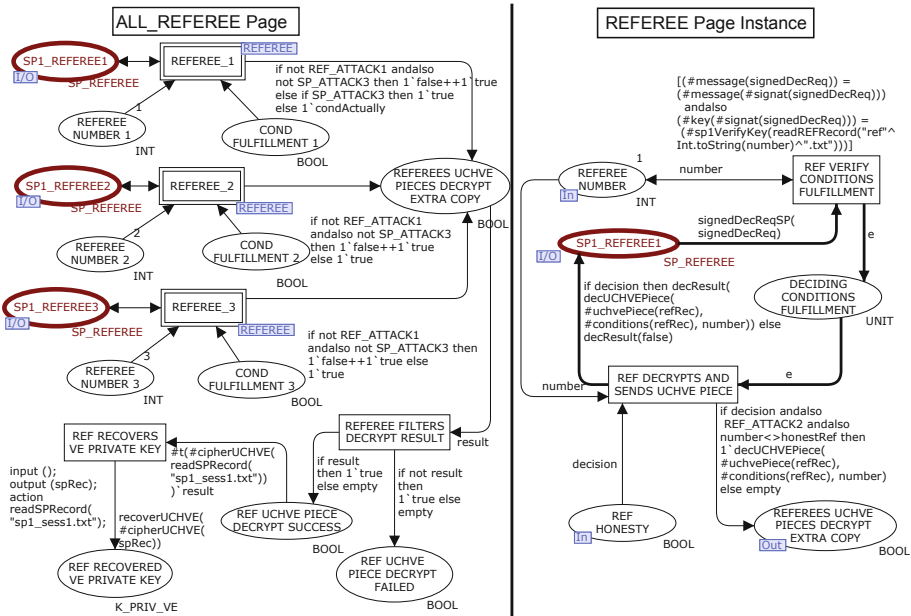


Fig. 7. ALL\_REFEREEES page (left) and a REFEREE page instance (right)

decision of conditions fulfillment (i.e. `REF_ATTACK1` and `SP_ATTACK3`) are false, the model considers both situations whereby the 1st Referee (`REFEREE_1`) agrees and disagrees to the fulfillment of the conditions; hence the place is initialized with both a `true` token and a `false` token.

When any one of the attacks listed above is set to true, we now need to be able to differentiate between “honest” referee and “malicious” referee. The `REF_ATTACK1` parameter captures the behaviour of malicious referees which agree on some conditions fulfillment even when it is not the case. An honest referee will state the *actual* fulfillment of the conditions, hence, for `REFEREE_1` (which represents an honest referee - consistent with our protocol assumption that there exists at least one honest designated referee), the value of the place `COND_FULFILLMENT_1` (Figure 7, top left) is set according to the model parameter `condActually`. For malicious referees (`REFEREE_2` and `REFEREE_3`), the initial value of their corresponding places, `COND_FULFILLMENT_2` and `COND_FULFILLMENT_3` (Figure 7, left middle), will be set to `true` (when `REF_ATTACK1` is set to `true`) to capture the malicious behaviour described before. The attack parameter `SP_ATTACK3` is defined to capture the behaviour of a service provider attempting to launch a revocation session using a set of made-up conditions which will most likely cause the referees to agree to their fulfillment. When this parameter is switched on, the initial marking for the above-mentioned places of all referees is a true token.

Figure 7 (right) also shows the parameterization of malicious referees who attempt to pool all decrypted UCHVE pieces amongst themselves with the hope of being able to recover the VE private key (captured by the parameter `REF_ATTACK2`). By studying the inscription of the arc from the transition `REF_DECRYPTS_AND_SENDS_UCHVE_PIECE` to place `REFEREES_UCHVE_DECRYPT_EXTRA_COPY` (Figure 7, bottom right) and by observing the parameter `honestRef` set to 1 (Table 1 line 39), this malicious behaviour only applies to `REFEREE_2` and `REFEREE_3`.

## 4 Verification of PIEMCP

### 4.1 Analysis Approach

We conduct the verification of PIEMCP using state space analysis. The verification can be complex due to the numerous avenues by which attackers could attempt to break the privacy protection provided by PIEMCP. We propose to scope the verification within a set of plausible known attack scenarios.

The verification of PIEMCP takes into account both the absence and presence of attack behaviours, and is carried out in two stages: the baseline behaviour analysis (Sect. 4.2) and privacy compliance verification (Sect. 4.3). Firstly, the baseline behaviour analysis is performed through standard state space analysis, including the inspection of proper session termination, deadlock/livelock freedom, and absence of unexpected dead transitions. As a result, the analysis informs us about the baseline correctness of PIEMCP. Next, we specify a set of common privacy compliance properties of PIEMCP using ASK-CTL [9], a dialect of Computational Tree Logic (CTL), supported by CPN Tools. These property statements are then interpreted into queries for model-checking the

state spaces generated from the PIEMCP CPN model to prove if the privacy compliance holds for the protocol.

The PIEMCP CPN model has an initial state where the protocol can begin the setup process and the session number is initialised with the first session's identifier and the three referee identifiers are specified. This is captured by the initial marking  $M_0$  where the places `START_SETUP` and `SESSION` on the `USER` page (Figure 3, bottom) and the three `REFEREE_NUMBER_i` ( $i = 1, 2, 3$ ) places on the `ALL_REFEREEES` page (Figure 7, left-hand side) are marked accordingly. Furthermore, our model has a `session` parameter to execute two sequential *escrow sessions* before a revocation is started (see Table 1 line 37).

In the absence of attack behaviour (that is, all attack parameters are set to `false`), the state space generated from the above configuration has 606 nodes and 1374 arcs, and contains no cycles (given the fact that the SCC graph has the same number of nodes and arcs). The PIEMCP CPN model is then configured to include a number of known attacks, and is executed under each of the configurations. A set of state spaces is generated capturing the behaviour of PIEMCP with the corresponding attack scenarios.

We introduce some notations to be used.  $CPN_P^{M_0}$  denotes the PIEMCP CPN model with an initial marking  $M_0$ .  $P_{PlaceName}^{PageName}$  and  $T_{TransitionName}^{PageName}$  refer to a specific place and transition in the CPN model, respectively. The marking of a place is then written as  $M(P_{PlaceName}^{PageName})$ .

## 4.2 Baseline Behaviour Analysis

The standard state space report generated from  $CPN_P^{M_0}$  *without* any attack behaviour (when all the attack parameters are set to `false`) shows that there are 8 dead markings. A close inspection of these markings indicates that they reflect all 8 different protocol termination points based on the dynamic conditions fulfillment decision (boolean decision) by the 3 referees modelled on the `ALL_REFEREEES` page (Figure 7). Also, there are three dead transitions:  $T_{USE\_ATTACK\_COND}^{SPI\_REV}$  (Figure 6),  $T_{REF\_RECOVERED\_VE\_PRIVATE\_KEY}^{ALL\_REFEREEES}$ , and  $T_{REF\_RECOVERED\_VE\_PRIVATE\_KEY}^{ALL\_REFEREEES\_REF\_RECOVERED\_VE\_PRIVATE\_KEY}$  (Figure 7-left). These are expected dead transitions because they reflect attack behaviours.

Moreover, the report shows that both the upper and the lower integer bounds of the place  $P_{SESSION}^{USER}$  is 1 (i.e. a place invariant). This is expected since the place is marked with the identifier of an ongoing escrow session throughout the protocol execution (where sessions are executed one by one without interruption). Also, the place  $P_{REF\_RECOVERED\_VE\_PRIVATE\_KEY}^{ALL\_REFEREEES}$  is always *empty*, which is expected as it reflects the modelling strategy for capturing problems (which then marks this place) with the basic design of the PIEMCP itself. In conclusion, the state space report confirms the expected baseline behaviour of PIEMCP *without* attacks.

For the PIEMCP *with* attacks (when one or more attack parameters are set to `true`), the expected baseline behaviour is to stop the protocol execution as soon as an attack is detected - a *fail-stop* behaviour. We would like to validate that the PIEMCP CPN model exhibits such behaviour when taking into account all possible attack scenarios.



**Table 2.** The set of attack parameters, their effects on the PIEMCP stages, and the number of attack scenarios to consider

Entity	Parameter	Escrow Session			Revoc.	Description
		PE	KE	MC		
User	USER_ATTACK1	T				Incorrect PII and <i>Cond1</i> used to generate $VE(PII)_{pubVE}$
	USER_ATTACK2	T				Incorrect $pubk_{VE}$ sent to IdP
	USER_ATTACK3		T			Incorrect UCHVE parameters used
	USER_ATTACK4		T			Non-agreed “hard to fulfill” conditions used in TPM Module2
Service Provider	SP_ATTACK1	T				Non-agreed “easy to fulfill” conditions forwarded to IdP
	SP_ATTACK11		T			during PE and KE respectively
	SP_ATTACK12		T			Non-agreed UCHVE parameters forwarded to IdP
	SP_ATTACK2			T		Non-agreed “easy to fulfill” conditions forwarded to IdP at MC
	SP_ATTACK22			T		Non-agreed UCHVE parameters forwarded to IdP at MC
	SP_ATTACK3				T	See explanation for Figure 6, pp. 262
	SP_ATTACK6				T	SP2 uses invalid signature key
Referee	SP_ATTACK7			T		SP1 and SP2 use the same condition within an escrow session
	REF_ATTACK1				T	See explanation for Figure 6 and
	REF_ATTACK2				T	Figure 7, pp. 262
Number of attack scenarios to consider		$2^3-1$	$2^4-1$	$2^4-1$	$2^3-1$	

Table 2 details all the 14 attack parameters considered, and the particular stage of PIEMCP where each attack may take place. As mentioned in Sect. 3, the PIEMCP CPN model captures sequential executions of the four stages in the order that PE is followed by KE, then MC and optionally Revocation stage at last. Following this order, we first allow only the attacks to occur in the PE stage. There are  $2^3-1$  attack scenarios resulting from combinations of 3 attack parameters (USER\_ATTACK1, USER\_ATTACK2 and SP\_ATTACK1). These are captured by 7 configurations of  $CPN_P^{M_0}$  which then lead to the generation of 7 state spaces. Analysis of these state spaces shows that the protocol detects the above attacks and terminates within the PE stage. Similarly, we allow only the attacks to occur in the subsequent KE, MC, and Revocation stages, respectively, and the analysis results show that for each of the stages the protocol detects the relevant attacks and terminates within that stage.

From the above analysis, it follows that due to the sequential execution of the four stages of PIEMCP and the fact that the fail-stop mechanism *does* work within each of these stages, once an attack occurs in an earlier stage (e.g. PE) the protocol terminates within that stage, regardless of whether or not the attacks are allowed to happen in a subsequent stage (e.g. KE, MC, or Revocation).

Therefore, the total 44 attack scenarios that pass the above fail-stop behaviour validation cover the behaviour of all possible  $2^{14}$  attack scenarios in PIEMCP based on the list of 14 attack parameters specified in Table 2.

### 4.3 Privacy Compliance Verification

We define four privacy compliance properties for PIEMCP. These are formalised as ASK-CTL statements over  $CPN_P^{M_0}$ . CPN Tools support ASK-CTL [9] as an implementation of a subset of CTL (mainly the “until” operator) over the state spaces of CPN models. ASK-CTL implements two basic path quantification operators to capture this logic:  $\text{EXIST\_UNTIL}(A_1, A_2)$  and  $\text{FORALL\_UNTIL}(A_1, A_2)$ . The  $\text{EXIST\_UNTIL}$  operator means that there must be at least *one* path, from a given state, whereby predicate  $A_1$  holds for every state in the path until the state where predicate  $A_2$  holds. The  $\text{FORALL\_UNTIL}$  operator is similar, except that it requires *all* paths to fulfill  $A_1$  until  $A_2$  is true. From these, two derived path quantification operators are  $\text{POS}(A)=\text{EXIST\_UNTIL}(\text{true}, A)$  and  $\text{EV}(A)=\text{FORALL\_UNTIL}(\text{true}, A)$ , which check the reachability of a state in which predicate  $A$  holds. More specifically,  $\text{POS}(A)$  checks if there is at least *one* path that leads to a state where  $A$  holds (i.e. it is **possible** to reach such a state), while  $\text{EV}(A)$  checks if *all* paths lead to a state where  $A$  holds (i.e. it must **eventually** reach such a state) [4].

Below, we use the above ASK-CTL temporal operators, a dialect of those in CTL, to specify four privacy compliance properties in the context of PIEMCP. We introduce some notations to be used in the property definitions. Firstly, we divide the 14 attack parameters into two groups:  $A_{ses}$  for the set of attack parameters targeting an escrow session, and  $A_{rev}$  for the set of attack parameters targeting a revocation stage. More specifically,

- $A_{ses} = \{\text{USER\_ATTACK1}, \text{USER\_ATTACK2}, \text{USER\_ATTACK3}, \text{USER\_ATTACK4}, \text{SP\_ATTACK1}, \text{SP\_ATTACK11}, \text{SP\_ATTACK12}, \text{SP\_ATTACK2}, \text{SP\_ATTACK22}, \text{SP\_ATTACK6}, \text{SP\_ATTACK7}\}$
- $A_{rev} = \{\text{SP\_ATTACK3}, \text{REF\_ATTACK1}, \text{REF\_ATTACK2}\}$

Next, we define two predicates with respect to an escrow session or a revocation:

- $S$  is the set of escrow sessions,  $\forall s \in S, \text{Session}^s M = (M(P_{\text{SESSION}}^{\text{Main}}) = 1^s)$
- $R$  is the set of revocable sessions,  $\forall r \in R, \text{Revoking}^r M = (M(P_{\text{BEING\_REVOKED}}^{\text{SP1\_REV}}) = 1^r)$

We refer to various places and transitions in the formalization of properties. Given the space constraints, only the formalization of the *enforceable conditions* can be followed using the CPN pages that have been described in Sect. 3.2. Other properties refer to certain places/transitions located within those CPN pages which are described in the Appendix.

**Multiple Conditions.** In PIEMCP, when no attack occurs during an escrow session, the *multiple conditions* property requires that the protocol always reaches the end of the session, and also each SP should receive an escrowed PII

---

<sup>4</sup> ASK-CTL provides many other operators, which we do not use in the compliance property specification.

that is cryptographically bound to conditions which are different from one SP to another. Any attacks which may compromise this property must be detected and caused a premature ending of the protocol. We have configured the CPN model with one attack parameter that may compromise this property, `SP_ATTACK7`, which depicts the scenario of SPs colluding to use the same condition string with the same user in a session. The goal of this attack is to make sure that all SPs involved in an escrow session share the same *condition* string such that when it is satisfied, all SPs within that escrow session are authorized to learn the user's PII. Therefore, in such a scenario, we expect the user to detect it and prematurely end its interaction with the malicious SP.

We formalize the above informal property definition as follows: In  $CPN_P^{M_0}$ , in the absence of attack behaviour, when the protocol runs to the end of an escrow session, the place  $P_{CAM\_START\_NEXT\_SESSION}^{USER\_MC}$  is marked signaling the end of a MC stage (i.e. the end of a session - see [21, Figure 16] for the corresponding CPN model), and the place  $P_{SESSION}^{Main}$  (Figure 2) is marked by the session identifier of that escrow session. The two places,  $P_{UCHVE\_COND}^{SP1\_KE}$  and  $P_{UCHVE\_COND}^{SP2}$  (see [21, Figure 15, Figure 18] for the corresponding CPN models), which are used to store the above conditions regarding an escrowed PII for  $SP_1$  and  $SP_2$  respectively, should be marked by different conditions at the end of an escrow session. Informally, this means that the value of *Cond1* and *Cond2* (referred to in Section 2) must not be the same ( $Cond1 \neq Cond2$ ).

When `SP_ATTACK7` is switched on, the desired behaviour of our protocol (reflecting the non-violation of this property) is captured by those execution paths which lead to a marking where  $P_{CAN\_REQUEST\_SP2\_SERVICE}^{USER\_MC}$  (see [21, Figure 16] for the corresponding model) is marked with a `false` token.

*Property 1 (Multiple Conditions).* With the following predicates:

- `SessionEnd`  $M = (M(P_{CAM\_START\_NEXT\_SESSION}^{USER\_MC}) = 1'e)$
- `DiffCondSP`  $M = (M(P_{UCHVE\_COND}^{SP1\_KE}) \neq \emptyset \wedge M(P_{UCHVE\_COND}^{SP2}) \neq \emptyset \wedge M(P_{UCHVE\_COND}^{SP1\_KE}) \neq M(P_{UCHVE\_COND}^{SP2}))$
- `ReqSP2Fail`  $M = (M(P_{CAN\_REQUEST\_SP2\_SERVICE}^{USER\_MC}) = 1'false)$

PIEMCP has *multiple conditions* property iff  $CPN_P^{M_0}$  has the following behaviour:

- if all the attack parameters  $A_{ses}$  are `false`, then  $\forall s \in S, \forall M \in M_0 >: \text{Ev}(\text{Session}^s M \wedge \text{SessionEnd } M \wedge \text{DiffCondSP } M)$
- otherwise, if `SP_ATTACK7` (and others in  $A_{ses}$  are `false`), then  $\exists s \in S, \exists M \in M_0 >: \text{Ev}(\text{Session}^s M \wedge \text{ReqSP2Fail } M)$  □

**Zero-knowledge.** In PIEMCP, when there are no attacks during an escrow session, and before the revocation of a user's PII for that escrow session, the zero-knowledge property requires that the IdP must validate that the ciphertexts (and the corresponding parameters) it possesses are correct while at the same time does not learn the value of the user's PII. When there are attacks which may compromise this property, we require our protocol to be able to detect

it. A malicious entity (such as user) may falsify the ciphertexts or their related parameters with the hope that the IdP does not detect it and still accepts the ciphertexts and the related parameters. If this situation occurs, then this property is violated due to flaws in the design of our protocol. We have modelled six attacks that may compromise this property with the parameters: `USER_ATTACK1`, `USER_ATTACK2`, `USER_ATTACK3`, `USER_ATTACK4`, `SP_ATTACK12`, and `SP_ATTACK22`. All of these attacks involve either the user or SPs sending to the IdP some incorrect/falsified ciphertexts and/or related parameters. For example, `USER_ATTACK1` involves the user sending to the IdP a ciphertext which encrypts some “garbage” data. The details of how we model these attacks are available in the extended version of this paper [21, Appendix B] (among which `USER_ATTACK2` was described in Sect. 3.2).

We formalize this property as follows: In  $CPN_P^{M_0}$ , three places,  $P_{PKVE\_RESULT}^{IDP\_PE}$  (Figure 4, bottom-left corner),  $P_{TPM\_PROOF\_VERIFICATION\_RESULT}^{IDP\_KE}$  ([21, Figure 14]), and  $P_{TPM\_PROOF\_VERIFICATION\_RESULT}^{IDP\_MC}$  ([21, Figure 17]), capture the correctness of the encryption. When there are no attacks, all three places must be marked by a `true` token; when any of the above-mentioned attacks is switched on, at least one of the three places must be marked by a `false` token.

*Property 2 (Zero-knowledge).* With the following predicates:

- $UsrVE-T \ M = (M(P_{PKVE\_RESULT}^{IDP\_PE}) = 1'true)$
- $UsrVE-F \ M = (M(P_{PKVE\_RESULT}^{IDP\_PE}) = 1'false)$
- $UchveKE-T \ M = (M(P_{TPM\_PROOF\_VERIFICATION\_RESULT}^{IDP\_KE}) = 1'true)$
- $UchveKE-F \ M = (M(P_{TPM\_PROOF\_VERIFICATION\_RESULT}^{IDP\_KE}) = 1'false)$
- $UchveMC-T \ M = (M(P_{TPM\_PROOF\_VERIFICATION\_RESULT}^{IDP\_MC}) = 1'true)$
- $UchveMC-F \ M = (M(P_{TPM\_PROOF\_VERIFICATION\_RESULT}^{IDP\_MC}) = 1'false)$

PIEMCP has *zero-knowledge* property iff  $CPN_P^{M_0}$  has the following behaviour:

- if all the attack parameters in  $A_{ses}$  are `false`, then  $\forall s \in S$ :  

$$Ev(\text{Session}^s \wedge UsrVE-T \wedge UsrTPM-T \wedge UchveKE-T \wedge UchveMC-T) \wedge$$

$$\neg Pos(\text{Session}^s \wedge (UsrVE-F \vee UsrTPM-F \vee UchveKE-F \vee UchveMC-F))$$
- if `USER_ATTACK1`  $\vee$  `USER_ATTACK2` (and others in  $A_{ses}$  are `false`), then  $\exists s \in S$ :  $Ev(\text{Session}^s \wedge UsrVE-F) \wedge \neg Pos(\text{Session}^s \wedge UsrVE-T)$
- if `USER_ATTACK3`  $\vee$  `USER_ATTACK4`  $\vee$  `SP_ATTACK12` (and others in  $A_{ses}$  are `false`), then  $\exists s \in S$ :  $Ev(\text{Session}^s \wedge UchveKE-F) \wedge \neg Pos(\text{Session}^s \wedge UchveKE-T)$
- if `SP_ATTACK22` (and others in  $A_{ses}$  are `false`), then  $\exists s \in S$ :  $Ev(\text{Session}^s \wedge UchveMC-F) \wedge \neg Pos(\text{Session}^s \wedge UchveMC-T)$   $\square$

**Enforceable Conditions.** The *enforceable conditions* property requires that a user’s PII should never be revealed unless all designated referees agree that the cryptographically bound conditions are satisfied and that the referees must not be able to learn the value of the PII themselves (they can only decrypt UCHVE ciphertext pieces which does not allow them to learn the PII - at least  $k$  decrypted UCHVE pieces are needed). This requirement applies regardless of whether there are any attack behaviours or not. Possible attacks that

can be launched to compromise this property include those parameterised by REF\_ATTACK1 and REF\_ATTACK2 (both attacks have been described in Sect. 3.2).

We formalize this property as follows. Note that the fulfilment status of certain revocation conditions for a session is captured by parameter `condActually` (Table 1 line 36). In  $CPN_P^{M_0}$ , if `condActually` does not hold, then: (1) the number of decrypted UCHVE pieces ( $|M(P_{\text{UCHVE\_DECRYPT\_SUCCESS}}^{\text{SP1\_REV}})|$  in Figure 6, bottom left) by the designated referees must be fewer than the minimum number of referees ( $k$ ) needed for a successful PII revocation, and (2) the user PII must not be revealed by checking the marking which indicates the revelation of the user PII ( $M(P_{\text{RECOVERED\_USER\_PII}}^{\text{SP1\_REV}}) \neq \emptyset$  in Figure 6, bottom right corner) in each revoking session must not be reached too. When `condActually` holds, we expect the number of decrypted UCHVE pieces to be greater or equal to  $k$ , and that the user's PII must eventually be revealed. Finally, we must ensure that the marking indicating illegal recovery of private VE key by the referees ( $M(P_{\text{REF\_RECOVERED\_VE\_PRIVATE\_KEY}}^{\text{ALL\_REFEREES}}) \neq \emptyset$  in Figure 7, bottom left corner) is *not reachable*.

*Property 3 (Enforceable Conditions).* With these predicates and notations:

- HasRefVEKey  $M = (M(P_{\text{REF\_RECOVERED\_VE\_PRIVATE\_KEY}}^{\text{ALL\_REFEREES}}) \neq \emptyset)$
- HasRecUsrPII  $M = (M(P_{\text{RECOVERED\_USER\_PII}}^{\text{SP1\_REV}}) \neq \emptyset)$
- HasRevocation  $M = (M(P_{\text{SESSION\_BEING\_REVOKED}}^{\text{SP1\_REV}}) \neq \emptyset)$
- $k = 2, \dots, n$  specifies the minimum number of referees who need to confirm the fulfilment of revocation conditions for a successful PII revocation
- $[M_0 >$  is the set of reachable markings (from the initial marking  $M_0$ )

PIEMCP has *enforceable conditions* property if and only if  $CPN_P^{M_0}$ , with all the parameters in  $A_{ses}$  being `false`, has the following behaviour:

- $\neg \text{Pos}(\text{HasRefVEKey})$
- if  $\neg \text{condActually}$ , then
  - $\forall M \in [M_0 >: \text{HasRevocation}(M) \Rightarrow |M(P_{\text{UCHVE\_DECRYPT\_SUCCESS}}^{\text{SP1\_REV}})| < k$
  - $\forall r \in R: \neg \text{Pos}(\text{Revoking}^r \wedge \text{HasRecUsrPII})$
- otherwise (`condActually`)
  - $\exists M \in [M_0 >: \text{HasRevocation}(M) \Rightarrow |M(P_{\text{UCHVE\_DECRYPT\_SUCCESS}}^{\text{SP1\_REV}})| \geq k$
  - $\forall r \in R: \text{EV}(\text{Revoking}^r \wedge \text{HasRecUsrPII})$  □

**Conditions Abuse Resistant.** The *conditions abuse resistant* property requires that an SP and an IdP must not be able to make the user to encrypt the PII or the VE private key, under a set of conditions different from those originally agreed. Similarly, an SP or IdP must not be able to successfully revoke the user's PII using conditions different from those originally agreed. Various attacks which may compromise this property have been modelled (USER\_ATTACK1, SP\_ATTACK1, USER\_ATTACK4, SP\_ATTACK11, SP\_ATTACK2, and SP\_ATTACK3). From the brief explanation of these attacks shown in Table 2, we can see that these attacks all involve manipulating the condition string at various stages of PIEMCP. The details of how we modelled these attacks are available in the full version of this paper [21, Appendix B] (with the exception of SP\_ATTACK3 which have been explained in detail in Section 3.2).

We formalize this property as follows: When there are no attacks, the cryptographically bound conditions used to produce a VE ciphertext must be the same as the one originally agreed (EqGenVEConds, EqSP1UchveConds, EqSP2UchveConds). When there are attacks targeting the general conditions used in the PE stage (parameterised by USER\_ATTACK1 and SP\_ATTACK1), we expect that the IdP is able to detect such an attempt to use incorrect conditions (different from those originally agreed, which is captured by  $\neg$ EqGenVEConds) thus resulting in the incorrect encryption (UsrVE-F). Similar behaviour applies to the scenarios involving USER\_ATTACK4 and SP\_ATTACK11 as well as those involving SP\_ATTACK2.

For attacks targeting the use of invalid conditions during the revocation stage (parameterised by SP\_ATTACK3), we expect that the transition  $T_{\text{USE\_ATTACK\_COND}}^{\text{SP1\_REV}}$  (Figure 6, middle-left) is not a dead transition anymore, and that the marking which indicates the revelation of user's PII (HasRecUsrPII), or the illegal revelation of VE private key (HasRefVEKey) should *not* be reached.

The following CPN pages (and the corresponding figures in which these pages are shown) are used in the property definition: SETUP [21, Figure 10], USER\_PE (Figure 5), USER\_KE [21, Figure 16], USER\_MC [21, Figure 13], IDP\_PE (Figure 4), IDP\_KE [21, Figure 14], and IDP\_MC [21, Figure 17].

*Property 4 (Conditions Abuse Resistant).* With these predicates and notations:

- HasGenCond  $M = (M(P_{\text{GEN\_COND}}^{\text{SETUP}}) \neq \emptyset)$
- HasVECond  $M = (M(P_{\text{PII\_VE\_CIPHER}}^{\text{USER\_PE}}) \neq \emptyset)$
- HasSP1Cond  $M = (M(P_{\text{SP1\_COND}}^{\text{SETUP}}) \neq \emptyset)$
- HasSP1UchveCond  $M = (M(P_{\text{KVE\_UCHVE\_CIPHER}}^{\text{USER\_KE}}) \neq \emptyset)$
- HasSP2Cond  $M = (M(P_{\text{SP2\_COND}}^{\text{USER\_MC}}) \neq \emptyset)$
- HasSP2UchveCond  $M = (M(P_{\text{KVE\_UCHVE\_CIPHER}}^{\text{USER\_MC}}) \neq \emptyset)$
- EqGenVEConds( $M, M'$ ) =  $(M(P_{\text{GEN\_COND}}^{\text{SETUP}}) = M'(P_{\text{PII\_VE\_CIPHER}}^{\text{USER\_PE}}))$
- EqSP1UchveConds( $M, M'$ ) =  $(M(P_{\text{SP1\_COND}}^{\text{SETUP}}) = M'(P_{\text{KVE\_UCHVE\_CIPHER}}^{\text{USER\_KE}}))$
- EqSP2UchveConds( $M, M'$ ) =  $(M(P_{\text{SP2\_COND}}^{\text{USER\_MC}}) = M'(P_{\text{KVE\_UCHVE\_CIPHER}}^{\text{USER\_MC}}))$
- EqVECondIDP  $M = (M(P_{\text{GEN\_COND}}^{\text{IDP\_PE}}) \neq \emptyset \wedge M(P_{\text{IDP\_VE\_CIPHER}}^{\text{IDP\_PE}}) \neq \emptyset \wedge M(P_{\text{GEN\_COND}}^{\text{IDP\_PE}}) = M(P_{\text{IDP\_VE\_CIPHER}}^{\text{IDP\_PE}}))$
- EqUchve1CondIDP  $M = (M(P_{\text{CIPHER\_UCHVE\_KVE}}^{\text{IDP\_KE}}) \neq \emptyset \wedge M(P_{\text{AGREED\_COND}}^{\text{IDP\_KE}}) \neq \emptyset \wedge M(P_{\text{CIPHER\_UCHVE\_KVE}}^{\text{IDP\_KE}}) = M(P_{\text{AGREED\_COND}}^{\text{IDP\_KE}}))$
- EqUchve2CondIDP  $M = (M(P_{\text{CIPHER\_UCHVE\_KVE}}^{\text{IDP\_MC}}) \neq \emptyset \wedge M(P_{\text{AGREED\_COND}}^{\text{IDP\_MC}}) \neq \emptyset \wedge M(P_{\text{CIPHER\_UCHVE\_KVE}}^{\text{IDP\_MC}}) = M(P_{\text{AGREED\_COND}}^{\text{IDP\_MC}}))$
- UsrVE-F, UchveKE-T, and UchveKE-F, refer to definitions in Property 2
- HasRefVEKey and HasRecUsrPII, refer to definitions in Property 3
- $BE(T)$  is the set of all binding elements for a transition (instance)  $T$
- $\forall M, M' \in [M_0 \rangle, \forall be \in BE, M \xrightarrow{be} M'$ :  $M'$  is reachable from  $M$  upon firing  $be$

PIEMCP has *conditions abuse resistant* property if and only if  $CPN_P^{M_0}$  has the following behaviour:

- if all the parameters  $A_{ses} \cup A_{rev}$  are **false**, then for each escrow session  $s \in S$ , and for markings  $M, M' \in [M_0 \rangle$  such that  $\text{Session}^s M$  and  $\text{Session}^s M'$ :

- $\text{HasGenCond}M \wedge \text{HasVECond}M' \Rightarrow \text{EqGenVEConds}(M, M')$
- $\text{HasSP1Cond}M \wedge \text{HasSP1UchveCond}M' \Rightarrow \text{EqSP1UchveConds}(M, M')$
- $\text{HasSP2Cond}M \wedge \text{HasSP2UchveCond}M' \Rightarrow \text{EqSP2UchveConds}(M, M')$
- if  $\text{USER\_ATTACK1} \vee \text{SP\_ATTACK1}$  (and others in  $A_{ses} \cup A_{rev}$  are **false**), then  
 $\exists s \in S: \text{EV}(\text{Session}^s \wedge \text{UsrVE-F}) \wedge \neg \text{POS}(\text{Session}^s \wedge \text{UsrVE-T} \wedge \text{EqVECondIDP})$
- if  $\text{USER\_ATTACK4} \vee \text{SP\_ATTACK11}$  (and others in  $A_{ses} \cup A_{rev}$  are **false**), then  
 $\exists s \in S: \text{EV}(\text{Session}^s \wedge \text{UchveKE-F}) \wedge$   
 $\neg \text{POS}(\text{Session}^s \wedge \text{UchveKE-T} \wedge \text{EqUchve1CondIDP})$
- if  $\text{USER\_ATTACK2}$  (and others in  $A_{ses} \cup A_{rev}$  are **false**), then  
 $\exists s \in S: \text{EV}(\text{Session}^s \wedge \text{UchveMC-F}) \wedge$   
 $\neg \text{POS}(\text{Session}^s \wedge \text{UchveMC-T} \wedge \text{EqUchve2CondIDP})$
- if  $\text{SP\_ATTACK3}$  (and others in  $A_{ses} \cup A_{rev}$  are **false**), then
  - $\exists be \in BE(T_{\text{USE\_ATTACK\_COND}}^{\text{SP1\_REV}}): \exists M, M' \in [M_0] \langle M \xrightarrow{be} M' \rangle$   
 (i.e.  $T_{\text{USE\_ATTACK\_COND}}^{\text{SP1\_REV}}$  is not a dead transition)
  - $\neg \text{POS}(\text{HasRefVEKey})$
  - $\exists r \in R: \neg \text{POS}(\text{Revoking}^r \wedge \text{HasRecUsrPII})$  □

The above four property specifications have been implemented into ASK-CTL queries (based on the full syntax of ASK-CTL) in CPN Tools for model-checking the state spaces of  $CPN_P^{M_0}$ . The results of the execution of these queries over the 45 state spaces in total (capturing the protocol without attack or with various attacks, refer to Table 2) demonstrate that PIEMCP satisfies these four privacy compliance properties.

## 5 Related Work

Formal methods based on process algebra have been used to model and verify security protocols (such as LySa [8]). Process algebra allows the modelling of a system's behaviour as a set of algebraic statements. Common verification techniques used with process algebra include equational reasoning and model checking [4]. For example, the Pi-Calculus [15] supports *labeled transition* semantics in modelling a system. This allows the verification of protocols through state exploration techniques such as model checking. However, we choose not to use process algebra approach because of its complexity which tends to (unnecessarily) complicate even simple things [1]. In comparison to the graphical-based modelling approach in CPNs, the pi-calculus approach is a less intuitive approach to model a large distributed system such as PEPs. Model validation can only be performed by users who are experts in both the protocol itself *and* the process algebra syntax. Nevertheless, pi-calculus-based approach has been used to verify privacy-related technologies, such as the DAA protocol [3].

State exploration techniques (such as state space analysis and model checking) have also been widely used for security protocol analysis. Examples belonging to this category are Scyther [10], and ProVerif [7]. These are state-of-the-art tools capable of automatically detecting attacks in many security protocols. The main reason we do not use these tools is because the types of security properties verifiable by these tools are not relevant to PEPs. Instead, they are mostly relevant to



authentication and key agreement protocols, i.e. *secrecy*, *authenticity*, and their variants. When protocols related to privacy are verified using these tools, the privacy property is reduced to confidentiality and authenticity. We argue that this is a simplistic approach to verifying privacy and that privacy does not simply equate to confidentiality and/or authenticity. The *behaviour* of a protocol in preserving or violating a user’s privacy is just as important. These tools also lack the rich graphical and simulation support of CPNs<sup>5</sup>. Therefore, we do not find these tools to be suitable for our purpose. Although CPNs have been widely used to analyze industrial communication protocols (such as Transmission Control Protocol (TCP) [6]), its use in the area of security protocols is still new with limited documented cases. For example, Al-Azzoni et al [2] used CPN to model and verify the Tatebayashi, Matsuzaki, and Newman (TMN) key exchange protocol [23]. The main difference between our work and theirs is that they focus on verifying the *secrecy* property of the TMN protocol, while our work focus on verifying the privacy behaviour of PEPs. The work presented in this paper is an extension of our earlier work [22]. The main differences include: (1) the improvement of the PIEMCP CPN model by re-structuring the model in terms of modularisation of individual entities, their communication channels, and different stages of operations; (2) the inclusion of the dynamic referee behaviour, i.e. the `ALL_REFEREEES` page and instantiation of the one `REFEREE` page according to the number of referees involved; (3) a detailed analysis of the attack scenarios, which leads to the finding of a set of necessary configurations of the PIEMCP CPN model capturing all possible attack behaviours; (4) the elaboration of privacy compliance properties in terms of an improved formalisation of property definitions which we believe is more precise and fine-grained (e.g. each property is now defined in terms of a set of relevant attack behaviours, instead of a “blanket” approach used in the previous work [22]); and (5) analysis and verification of PIEMCP based on the updated CPN model and privacy compliance property definitions.

## 6 Conclusion

We have shown that CPNs can be used to model complex PEPs, a class of cryptographic protocols, and support the verification of their privacy compliance properties based on state space analysis. We have also proposed several modelling techniques, notably the cryptographic primitive abstraction (capturing complex primitives and zero-knowledge proof protocol) and parameterised attacks. We have also shown how we can formalise and verify privacy compliance properties using standard state space analysis techniques and ASK-CTL queries.

Future work involves refinement and generalization of the modelling and analysis approaches proposed in this paper such that they can be applied to other PEPs. We also hope to build a better user front-end to simplify and automate the tasks required in the modelling and verification of PEPs. The function of

---

<sup>5</sup> Scyther provides some static graphical support. However, it falls short of interactive protocol simulation and graphically-driven protocol specification.



such a front-end could be as simple as aiding users with the configuration of attack parameters without the need of knowing CPNs. Another long-term goal is to achieve automated attack detections for PEPs using a CPN-based approach.

**Acknowledgements.** This work is partially supported by National ICT Australia (NICTA). NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

## References

1. van der Aalst, W.: Pi calculus versus Petri nets: Let us eat humble pie rather than further inflate the Pi hype. *BPTrends*, 1–11 (May 2005)
2. Al-Azzoni, I., Down, D.G., Khedri, R.: Modeling and verification of cryptographic protocols using Coloured Petri nets and Design/CPN. *Nordic Journal of Computing* 12(3), 201–228 (2005)
3. Backes, M., Maffei, M., Unruh, D.: Zero-knowledge in the applied Pi-calculus and automated verification of the direct anonymous attestation protocol. In: *IEEE Symposium on Security and Privacy*, pp. 202–215 (May 2008)
4. Baeten, J.C.M.: A brief history of process algebra. *Theor. Comput. Sci.* 335(2-3), 131–146 (2005)
5. Bellare, M., Rogaway, P.: Random oracles are practical: A paradigm for designing efficient protocols. In: *ACM CCS*, pp. 62–73 (1993)
6. Billington, J., Han, B.: Modelling and analysing the functional behaviour of TCP's connection management procedures. *STTT* 9(3-4), 269–304 (2007)
7. Blanchet, B.: An efficient cryptographic protocol verifier based on Prolog rules. In: *14th IEEE CSFW*, pp. 82–96. *IEEE Computer Society* (2001)
8. Bodei, C., Buchholtz, M., Degano, P., Nielson, F., Nielson, H.R.: Static validation of security protocols. *J. Comput. Secur.* 13(3), 347–390 (2005)
9. Christensen, S., Mortensen, K.H.: *Design/CPN ASK-CTL Manual - Version 0.9*. University of Aarhus, Aarhus C, Denmark (1996)
10. Cremers, C.J.F.: The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols. In: Gupta, A., Malik, S. (eds.) *CAV 2008*. LNCS, vol. 5123, pp. 414–418. Springer, Heidelberg (2008)
11. Dolev, D., Yao, A.C.C.: On the security of public key protocols. *IEEE Transactions on Information Theory* 29(2), 198–207 (1983)
12. Gilmore, S.: *Programming in standard ML '97: A tutorial introduction*. Tech. rep., The University of Edinburgh (1997)
13. Jensen, K., Kristensen, L.M.: *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer (2009)
14. Koblitz, N., Menezes, A.: Another look at "provable security". *J. Cryptology* 20(1), 3–37 (2007)
15. Milner, R.: *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press (June 1999)
16. Ngo, L., Boyd, C., Nieto, J.G.: Automating Computational Proofs for Public-Key-Based Key Exchange. In: Heng, S.-H., Kurosawa, K. (eds.) *ProvSec 2010*. LNCS, vol. 6402, pp. 53–69. Springer, Heidelberg (2010)

17. Pointcheval, D.: Contemporary cryptography - Provable security for public key schemes. *Advanced Courses in Mathematics*, pp. 133–189. Birkhäuser (2005)
18. Suriadi, S.: Strengthening and Formally Verifying Privacy in Identity Management Systems. Ph.D. thesis, Queensland University of Technology (September 2010)
19. Suriadi, S., Foo, E., Josang, A.: A user-centric federated single sign-on system. *Journal of Network and Computer Applications* 32(2), 388–401 (2009)
20. Suriadi, S., Foo, E., Smith, J.: Private information escrow bound to multiple conditions. Tech. rep., Information Security Institute - Queensland University of Technology (2008), <http://eprints.qut.edu.au/17763/1/c17763.pdf>
21. Suriadi, S., Ouyang, C., Foo, E.: Privacy compliance verification in cryptographic protocols. Tech. Rep. 48484, Queensland University of Technology, Brisbane, Australia (2012), <http://eprints.qut.edu.au/48484/>
22. Suriadi, S., Ouyang, C., Smith, J., Foo, E.: Modeling and Verification of Privacy Enhancing Protocols. In: Breitman, K., Cavalcanti, A. (eds.) *ICFEM 2009*. LNCS, vol. 5885, pp. 127–146. Springer, Heidelberg (2009)
23. Tatebayashi, M., Matsuzaki, N., Newman Jr., D.B.: Key Distribution Protocol for Digital Mobile Communication Systems. In: Brassard, G. (ed.) *CRYPTO 1989*. LNCS, vol. 435, pp. 324–334. Springer, Heidelberg (1990)
24. WP 14.1: PRIME (Privacy and Identity Management for Europe) - Framework V3 (March 2008)

# Modeling Energy-Aware Cloud Federations with SRNs

Dario Bruneo, Francesco Longo, and Antonio Puliafito

Dipartimento di Matematica, Università degli Studi di Messina  
Viale F. Stagno d'Alcontres, 31 - 98166 Messina (ME), Italia  
{dbruneo,flongo,apuliafito}@unime.it

**Abstract.** Cloud computing is a challenging technology that promises to strongly modify the way computing and storage resources will be accessed in the near future. However, it may demand huge amount of energy if adequate management policies are not put in place. In particular, in the context of Infrastructure as a Service (IaaS) Cloud, optimization strategies are needed in order to allocate, migrate, consolidate virtual machines, and manage the switch on/switch off period of a data centre. In this paper, we present a methodology based on stochastic reward nets (SRNs) to investigate the more convenient strategies to manage a federation of two or more private or public IaaS Clouds. Several policies are presented and their impact is evaluated, thus contributing to a rational and efficient adoption of the Cloud computing paradigm.

**Keywords:** Cloud computing, Energy saving, Quality of Service, Performance evaluation, Stochastic reward nets.

## 1 Introduction

Energy saving is one of the most critical challenges of the 21th century, calling for efforts in a wide range of research fields, e.g., economics, engineering, chemistry. In the Information and Communications Technology (ICT) area, energy aware initiatives are recently classified under the term *Green computing* [1]. In this context, the *Cloud paradigm* is emerging as a promising technology able to rationalize the use of hardware resources by providing ubiquitous on demand access to virtual resources available on the Internet [2]. By moving applications on the Cloud, it is possible to manage load peaks reducing data center inefficiencies due to low resources utilization.

Clouds provide services at three different levels: *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS), and *Software as a Service* (SaaS). In the present work, we focus on IaaS Clouds that provide users with computational resources in the form of virtual machine (VM) instances deployed in a remote data center. In the literature, a distinction is recently being made among *public* and *private* IaaS Clouds [3]. A public Cloud is offered as a service, usually over an Internet connection, by an off-site third-party provider who bills on a fine-grained utility computing basis. Relationship between customers and providers

are regulated by Service Level Agreements (SLA). Such contracts characterize a service in terms of quality requirements, performance, and availability constraints, security, responsibility penalties, and methods of payment. Examples of public IaaS Clouds are Amazon EC2 [4] and IBM Smart Business Cloud [5,6]. Nowadays, public Clouds are widely known by the general public, while private Clouds are advancing as a promising technique able to allow a single organization to manage its own infrastructure in a scalable, reliable, and energy-aware way. We are witnessing the recent spreading of numerous software projects aiming at providing Cloud middleware that can be used to create private Clouds on top of classical hardware infrastructures with the aim of optimizing resources utilization and energy consumption. Examples of Cloud middleware are Eucalyptus [7] and openQRM [8].

Federation [9] is the next frontier in this context. Being deployed inside the internal firewall, a private Cloud is usually locally accessed by the organization users and it cannot be exploited by external entities. Throughout federation, different small and medium private Clouds, belonging to the same or to different organizations, can join each other to achieve a common goal, usually represented by the optimization of resources utilization. A single private Cloud can use its own infrastructure to deal with normal load conditions and can ask to its federated Cloud partners more resources to absorb load bursts. On the other hand, in an energy-aware Green computing context, virtual server consolidation policies [10] can be put in place to allow small organizations to completely turn off their infrastructure when the maintenance and energy costs are too high with respect to the load. Relationship among federated private Clouds are also regulated by SLAs. Configurations in which public and private Clouds interact among themselves are usually called *hybrid* Clouds.

The interest in topics related to energy saving in the field of Cloud computing is confirmed by the specific scientific literature and by the proliferation of several research projects, mainly aiming at proposing energy management policies [11] and VM consolidation techniques [12]. The validation of such approaches is usually performed through simulation [13]. Similarly, the problem of assessing the Quality of Service (QoS) offered by Cloud computing is an issue of primary and strategic importance. Powerful strategies are needed to guarantee SLA commitments and the performance analysis and prediction are essential instruments to define such strategies [14]. Several techniques have been used in the QoS and performance evaluation of Cloud infrastructures and services [15] mainly based on real measurement or experimental frameworks. Traditional measurement based performance evaluation requires extensive experimentation with each workload and system configuration and may not be feasible in terms of cost due to sheer scale of Cloud. A simulation model can be developed and solved but in contrast with an analytic model, it might be time consuming as the generation of statistically significant results may require many simulation runs [16]. Stochastic modeling is a more attractive alternative because of lower relative cost of solving the model while covering large parameter space [17].

In this paper, we present an analytical methodology based on stochastic reward nets (SRNs) [18], an extension of generalized stochastic Petri nets, to investigate the cost/benefit of different energy saving strategies to manage a federation of two or more Clouds in the context of IaaS. Public and private Clouds are both taken into consideration. The methodology is based on a compositional approach: the main components characterizing a typical private Cloud infrastructure are identified and represented using basic SRN models able to capture their functional and non-functional behavior. A basic SRN is also associated to each public Cloud infrastructure. Such basic SRNs are composed, according to the structure of the actual Cloud federated environment that has to be analyzed, with the aim of obtaining a complete model. Finally, such model can be solved and useful performance parameters can be obtained considering the final goal to reduce the overall management costs. Several aspects and policies characterizing a Cloud system, such as federation and VM consolidation [19], are taken into consideration and their impact is evaluated, thus contributing to a rational and efficient adoption of the Cloud computing paradigm.

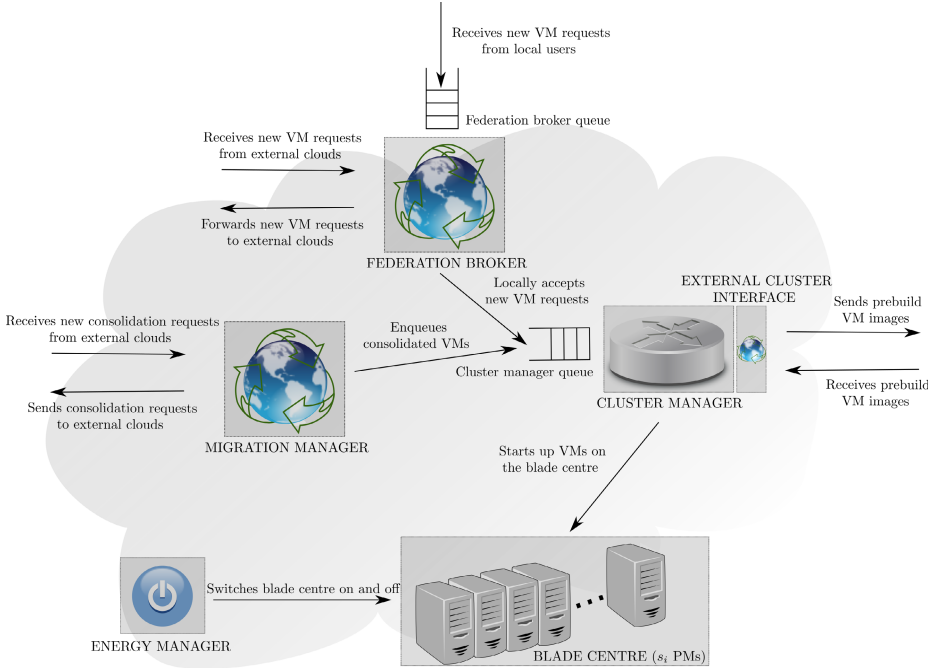
The paper is organized as follows. Section 2 describes the assumptions and formulates the problem. The basic SRN models associated to the Cloud components are illustrated in Section 3 while the criteria for their composition into the final model are described in Section 4. In such a section, a case study and the corresponding model are also shown. Performance assessments are discussed in Section 5. Related works are described in Section 6. Finally, concluding remarks and future works are presented in Section 7.

## 2 Assumptions and Problem Formulation

In the ecosystem of IaaS Clouds we take as a reference, both private and public Clouds are allowed (hybrid federated Cloud environment). We assume a scenario in which  $N$  private federated IaaS Clouds interact among each other and collaborate in order to optimize the resources utilization and the energy consumption, while still fulfilling user requests. In particular, we assume that private Clouds can federate each other by stipulating peer-to-peer business dealings and can back up on public Cloud services in order to deal with specific overload situations.  $M$  public Clouds can be present. Our aim is to represent such a scenario using of a composed SRN model and to quantify the advantages deriving from *energy management*, *federation*, and *VM consolidation* policies, both from the private organization and the user point-of-views.

### 2.1 Private Clouds

Fig. 1 shows our reference architecture for a single private IaaS Cloud. The main components and their interactions are highlighted. All the entities that we take into consideration are usually present in all the most used IaaS Cloud middleware. In the following, we provide a high level description of the behavior of each component in the system.



**Fig. 1.** Architecture of a single private IaaS Cloud

**Cluster Management** - Whenever a user request is processed in a private IaaS Cloud, a set of pre-built disk images is used to deploy one or more VM instances on the private organization data center. We assume that the data center of the  $i^{th}$  private Cloud (with  $1 \leq i \leq N$ ) is composed of  $s_i$  physical machines (PMs) organized into a single blade center. A PM can be usually shared by multiple VMs according to the CPU, RAM, and disk capacity requested by each VM but, as a simplifying assumption, we consider the case in which a single VM can be deployed on each PM and each user request is related to a single VM instantiation. A Cluster Manager (CM) translates the user requests into VM instantiation actions. New requests are accepted until the maximum CM queue capacity is reached; further requests are dropped.  $q_{CQ_i}$  indicates the maximum queue capacity for the  $i^{th}$  CM.

If a free PM exists, a request is taken from the CM queue and a VM instantiation action is performed. We suppose that, in order to execute a VM on a PM belonging to the local blade center, the CM simply needs to send an activation request to the PM hypervisor that can immediately mount the corresponding pre-built VM image from a distributed local file system with negligible time delay. Finally, once a VM ends its task, the corresponding image is stored back to the local distributed file system and the corresponding PM is let free to accept other requests. We assume the VM completion time in the  $i^{th}$  Cloud to be exponentially distributed with rate  $\lambda_{r_i}$ .

**Energy Management** - If an energy management policy is active, the private Cloud blade center and the associated chiller refrigerator (that usually represents the main cause of energy loss) can be switched off in order to reduce the energy consumption and save money. An Energy Manager (EM) monitors the state of the private Cloud and determines if the blade center can be switched on or off. Different policies can be taken into consideration in this context. In particular, the power off and power on actions can be triggered depending on the status of the managed Cloud infrastructure, e.g., the utilization of the blade center, the number of requests in the system queues, the current load. The set of conditions for the power off action to be triggered on the  $i^{th}$  private Cloud need to be verified for a certain amount of time, namely  $t_{off_i}$ . More fine grained energy management policies can be implemented such as activating/deactivating the single PMs. Our methodology is general enough to represent such management strategies. However, to simplify the presentation of the proposed approach, in this paper we assume that the energy management policy operates at the whole blade center level. We assume that the time necessary for the blade center to be switched on or off is negligible.

**Federation** - If a federation policy is active, a Federation Broker (FB) is used to communicate with external private or public Clouds in order to redirect user requests that cannot be fulfilled locally. New requests are accepted until the maximum FB queue capacity is reached; further requests are dropped.  $q_{BQ_i}$  indicates the maximum queue capacity for the FB associated to the  $i^{th}$  private Cloud. If a request stays in the FB queue for a time greater than  $t_{f_i}$  and at least one of the federated Clouds is able to accept it, then the request is forwarded. An External Cluster Interface (ECI) allows the CM to communicate with other CMs and to transfer the pre-built VM images related to the redirected requests. As soon as a VM is transferred from the local Cloud, the corresponding request is inserted in the CM queue of the remote Cloud to be processed. We assume that an exponentially distributed time delay with rate  $\lambda_{t_i}$  is necessary in order for a VM image to be copied to the external Cloud storage. A maximum number of  $T_i \in \mathbb{N}$  simultaneous transfers can be performed.

**VM Consolidation** - Finally, if a VM consolidation policy is set, a Migration Manager (MM) can trigger a consolidation action in order to migrate the VMs executing on the local data center to an available external Cloud. The goal of a consolidation action is usually to force a power off of the local blade center. If a Cloud exists willing to accept the migrating VMs, the ECI starts the transfer of the VM images. As soon as the VM migrations are completed, the corresponding requests are inserted in the remote CM queue to be started up again on the remote Cloud. Also in this case, we assume that an exponentially distributed time delay with rate  $\lambda_{m_i}$  is necessary in order for a VM image to be migrated from the  $i^{th}$  private Cloud. The VM consolidation is an atomic task and it is triggered only if all the running VMs can be transferred to a remote Cloud and if such a condition stays for a time greater than a certain threshold  $t_{m_i}$ . In this

way, at the end of a VM consolidation the blade center will result empty, thus allowing the scheduling of a switch off operation.

**VM Requests Arrival Process** - An important aspect in the scenario that we are taking into consideration is the modeling of the load. In order to take into account the dependency on the hours of a day, we model the VM request arrival process as a Markov modulated Poisson process (MMPP) with two states. We indicate with  $\lambda_{d_i}$  the rate of the arrival process for the  $i^{th}$  private Cloud during the day hours, while  $\lambda_{n_i}$  represents the rate of the arrival process during the night hours.  $\lambda_{dn}$  and  $\lambda_{nd}$  are the rates associated with the transitions between the two states of the MMPP, representing the alternation between day and night.

## 2.2 Public Clouds

We take into consideration a scenario in which  $M$  public IaaS Clouds are available. As already mentioned, we suppose that private Clouds can back up on public Clouds in order to deal with load bursts that cannot be managed by the local infrastructure and by any of the private federated partners. We suppose that a public Cloud does not share information about its current status and about the internal policies that regulates the management of its infrastructure. For such a reason, we prefer to model only the external behavior of the public Cloud with respect to its client. The SLAs that are stipulated to regulate this kind of commercial relationships are usually based on availability and price constraints. A public Cloud client usually asks for a certain level of service availability to be guaranteed. On top of it a corresponding price is agreed on a computation time basis. Another parameter is related to the maximum number of VMs that can be contemporaneously running. For such reasons, in the following we assume that a public Cloud can be easily included in the considered scenario by simply modeling its availability, the maximum number of submitted VMs, and their execution time. We assume that the public Cloud can be in one of two states: available or not available. We indicate with  $a_j$  ( $1 \leq j \leq M$ ) the  $j^{th}$  public Cloud availability, i.e., the portion of time that the public Cloud stays in its available state while we use  $R_j$  to define the maximum number of VMs contemporaneously running. Finally,  $\lambda_{pr_j}$  represents the rate of the exponentially distributed VM completion time.

## 2.3 System Policies

By mean of the proposed methodology, we are able to analyze different management policies. However, for a matter of clarity, in the remainder of the paper we will consider the following specific management policies.

**Energy Management** - The EM triggers a power off when no VMs are instantiated in the blade center, no requests are waiting in the system queues, and no consolidation and/or federation actions are being performed, i.e., no VM images



are being transferred among Clouds. On the other hand, a power on is triggered on the  $i^{th}$  Cloud blade center when the number of total requests waiting in the system queues is greater than a certain threshold  $P_i \in \mathbb{N}$ .

**Federation** - The FB of the  $i^{th}$  Cloud interacts with the corresponding ECI in order to submit the request to an available federated external Cloud if the local blade center is off, the CM queue is full, or a consolidation/power off action is being performed. The ECI of the  $p^{th}$  remote private Cloud accepts an external request if its blade center is on, it is not extremely loaded, i.e., the number of requests in its queues is not greater than a certain threshold  $E_p \in \mathbb{N}$ , and if no consolidation/power off action is currently being performed.

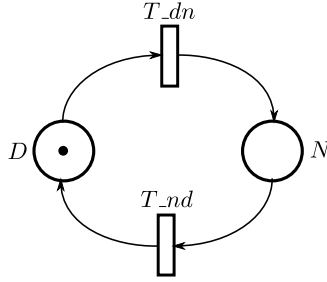
**VM Consolidation** - We assume that a consolidation action is triggered in the  $i^{th}$  Cloud when the number of VM instances currently running in the blade center is less than a threshold  $M_i \in \mathbb{N}$ , and no requests are waiting in the queues. The MM of the  $p^{th}$  remote Cloud accepts an external VM consolidation if its blade center is on, the number of PMs available to accept VM instances is greater than a certain threshold  $C_p \in \mathbb{N}$ , and if no consolidation/power off action is currently being performed.

### 3 Modeling Private and Public Cloud Components

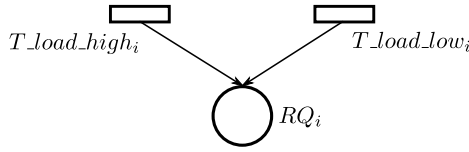
Let us describe the methodology for the performance and energy consumption analysis of the hybrid federated Cloud ecosystem described in Section 2. The final goal is to show how to translate the scenario and the assumptions reported in the previous section into a formal SRN model. SRNs [18] are extensions of generalized stochastic Petri nets (GSPNs) [20]. In SRNs, every tangible marking of the net can be associated with a reward rate thus facilitating the computation of a variety of performance measures. Key features of SRNs are: (1) each transition may have an enabling function (also called a guard) so that a transition is enabled only if its marking-dependent enabling function is true; (2) marking dependent arc multiplicities are allowed; (3) marking dependent firing rates are allowed; (4) transitions can be assigned different priorities; (5) besides traditional output measures obtained from a GSPN, such as throughput of a transition and mean number of tokens in a place, more complex measures can be computed by using reward functions.

In this section, we present the basic SRN models for each of the private Cloud building blocks of Fig. 1 and for a generic public Cloud infrastructure. We also show the evaluation that can be performed on top of each of them. Such evaluation is conducted through the estimation of a set of steady-state measures. In order to define all the measures, the following notation is used:  $E\{\#P\}$  is the expected number of tokens in place  $P$ ,  $Th\{T\}$  is the expected throughput of transition  $T$ , and  $Pr\{e\}$  is the probability that event  $e$  occurs.

Note that the enabling functions described in the following allow to model the policies described in Section 2.3. However, our methodology is general enough



**Fig. 2.** The SRN model for the day/night alternation



**Fig. 3.** The SRN model for the VM request arrival process to the  $i^{th}$  private Cloud

to deal with a variety of different energy and management policies by simply changing some of the enabling functions associated to the sub-model transitions without need to change the model structure.

**VM Requests Arrival Process** - Let us start with the model of the load. Fig. 2 represents the basic SRN that models the two stages of the MMPP, i.e., the alternation between day and night. Moving a token between places  $D$  and  $N$ , transitions  $T\_dn$  and  $T\_nd$  model the switching between such stages. They are exponentially distributed transitions with rate  $\lambda_{dn}$  and  $\lambda_{nd}$ , respectively. Fig. 3 depicts the basic SRN model for the VM request arrival process to the  $i^{th}$  private Cloud. Transitions  $T\_load\_high_i$  and  $T\_load\_low_i$  represent the request arrival during the day and the night hours, respectively. The rates associated to such exponentially distributed transitions are  $\lambda_{hi}$  and  $\lambda_{li}$ , respectively and, as soon as they fire, they insert a token into place  $RQ_i$  modeling the arrival of a new VM request. The enablement of such transitions depends on the MMPP stage alternation and it is regulated by the following enabling functions:

$$f_{load\_high_i} = \begin{cases} 1, & \text{if } \#D = 1 \\ 0, & \text{otherwise} \end{cases}$$

$$f_{load\_low_i} = \begin{cases} 1, & \text{if } \#N = 1 \\ 0, & \text{otherwise} \end{cases}$$

Other configurations can be modeled by changing the enabling functions associating the high load condition to the night stage or increasing the number of stages, e.g., modeling the presence of different timezones.

From the model of Fig. 3, the *expected number of requests per time unit* to the  $i^{th}$  Cloud ( $A^i$ ) can be computed as:

$$A^i = Th\{T_{load\_high_i}\} + Th\{T_{load\_low_i}\}. \tag{1}$$

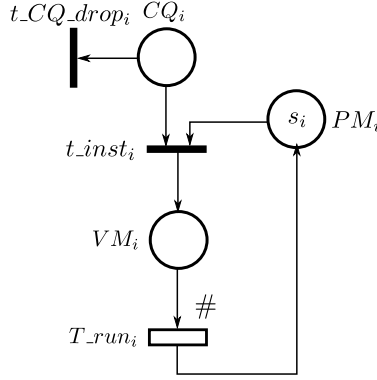


Fig. 4. The SRN model for the Cluster Manager of the  $i^{th}$  private Cloud

**Cluster Manager** - The SRN model for the CM of the  $i^{th}$  private Cloud is depicted in Fig. 4. Place  $CQ_i$  represents the CM queue. A token in this place models a VM request waiting for a free PM in order to be instantiated. Immediate transition  $t_{CQ\_drop_i}$  models the drop of a single request and it is associated with the enabling function  $f_{CQ\_drop_i}$  that allows it to be enabled if the maximum queue capacity has been reached, i.e., as soon as the number of tokens in place  $CQ_i$  is equal to  $q_{CQ_i} + 1$ :

$$f_{CQ\_drop_i} = \begin{cases} 1, & \text{if } \#CQ_i = q_{CQ_i} + 1 \\ 0, & \text{otherwise} \end{cases}$$

The blade center is modeled by place  $PM_i$ . Tokens in such place model PMs that are available to accept VM instances.

If a PM is available and a request is present in the CM queue, immediate transition  $t_{inst_i}$  fires, modeling the VM instantiation process. Such transition extracts a token from places  $CQ_i$  and  $PM_i$  and puts a token in place  $VM_i$ . The number of tokens in such place represents the number of VMs currently in execution in the blade center. The exponentially distributed transition  $T_{run_i}$  models the time necessary for a VM to complete its task. The VM executions are concurrent tasks and, for this reason, transition  $T_{run_i}$  is marking dependent and its rate is proportional to the number of tokens in place  $VM_i$  ( $\#VM_i \cdot \lambda_{r_i}$ ). As soon as transition  $T_{run_i}$  fires, it moves a token from place  $VM_i$  to place  $PM_i$  modeling the presence of a PM newly available to accept requests.

From the model of Fig. 4, several measures can be computed. In particular, the *CM drop probability* ( $P_{CM_d}^i$ ), i.e., the probability for a request to be dropped

due to the  $i^{th}$  CM queue being full, can be computed as:

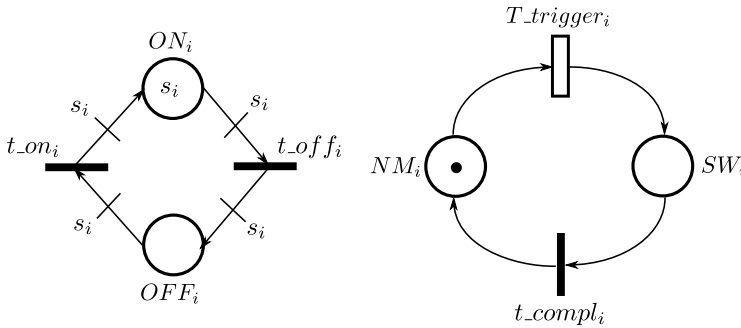
$$P_{CM_d}^i = Pr\{\#CQ_i = q_{CQ_i}\} \tag{2}$$

while, according to Little’s law, the *expected CM waiting time* ( $T_{CM_w}^i$ ) can be computed as:

$$T_{CM_w}^i = E\{\#CQ_i\}/Th\{t_{inst_i}\}. \tag{3}$$

Finally, information about the utilization of the  $i^{th}$  Cloud blade center can be provided computing the *expected number of local running VMs* ( $R_i^i$ ) as:

$$R_i^i = E\{\#VM_i\} \tag{4}$$



**Fig. 5.** The SRN model for the Energy Manager of the  $i^{th}$  private Cloud

**Energy Manager** - Fig. 5 depicts the SRN model for the EM of the  $i^{th}$  private Cloud. Places  $ON_i$  and  $OFF_i$  model the power status of the blade center. In particular, if  $s_i$  tokens are present in place  $OFF_i$  the blade center is off. Otherwise, if zero tokens are present in place  $OFF_i$ , the blade center is on. The presence of  $s_i$  tokens in place  $ON_i$  in Fig. 5 is motivated by the fact that the EM sub-model is designed in order to be composed with the CM sub-model as explained in Section 4. Immediate transitions  $t_{on_i}$  and  $t_{off_i}$  model the power on and power off of the blade center, respectively, moving  $s_i$  tokens from place  $OFF_i$  to place  $ON_i$  and vice-versa. Transition  $t_{off_i}$  is associated with the enabling function  $f_{off_i}$  that allows it to be enabled only if a token is present in place  $SW_i$ :

$$f_{off_i} = \begin{cases} 1, & \text{if } \#SW_i = 1 \\ 0, & \text{otherwise} \end{cases}$$

A token can be moved in such place by the exponentially distributed transition  $T\_trigger_i$  that models the condition for the power off action to be triggered. In particular, transition  $T\_trigger_i$  presents a rate equal to  $1/t_{off_i}$ . As detailed in Section 2,  $t_{off_i}$  is the time for which the power off condition needs to be verified

for a power off action to be triggered<sup>1</sup>. Enabling function  $f_{trigger_i}$ , associated with transition  $T\_trigger_i$ , allows it to be enabled only if the power off condition is verified:

$$f_{trigger_i} = \begin{cases} 1, & \text{if } \#ON_i = s_i \text{ AND } \#CQ_i = \#BQ_i = 0 \text{ AND} \\ & (\forall p \Rightarrow \#FW\_PR_i^p = \#MG\_PR_i^p = 0) \text{ AND} \\ & (\forall q \Rightarrow \#FW\_PU_i^q = \#MG\_PU_i^q = 0) \\ 0, & \text{otherwise} \end{cases}$$

with  $1 \leq p \leq N$  and  $p \neq i$  and  $1 \leq q \leq M$ .

Immediate transition  $t\_compl_i$  fires as soon the power off is completed, i.e., it has an enabling function  $f_{compl_i}$  that allows it to be enabled if  $\#OFF_i = s_i$ :

$$f_{compl_i} = \begin{cases} 1, & \text{if } \#OFF_i = s_i \\ 0, & \text{otherwise} \end{cases}$$

Similarly, the enabling function  $f_{on_i}$  associated with transition  $t\_on_i$  allows it to be enabled only if the condition for a power on action to be triggered is verified:

$$f_{on_i} = \begin{cases} 1, & \text{if } \#CQ_i + \#BQ_i > P_i \\ 0, & \text{otherwise} \end{cases}$$

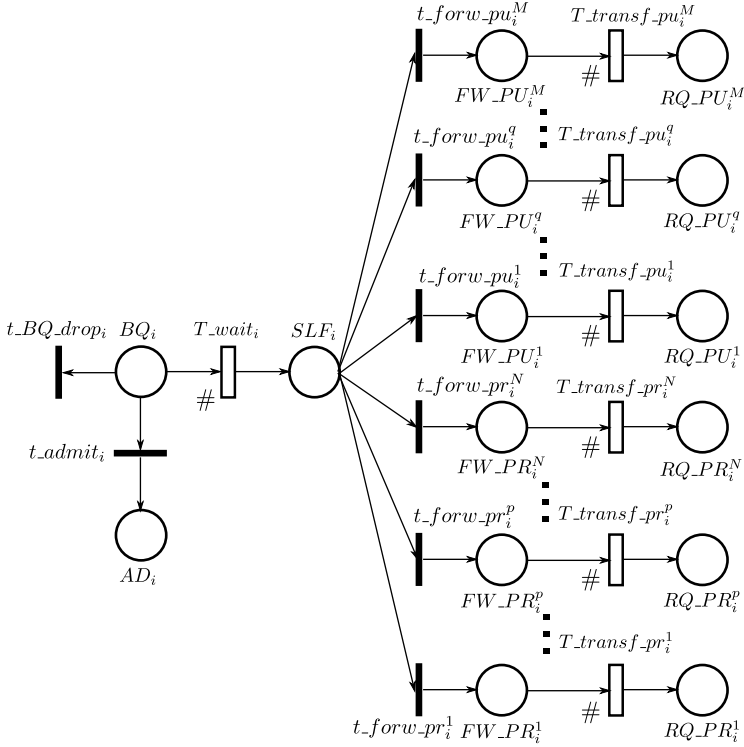
From the model of Fig. 5, the *Off probability* ( $P_{off}^i$ ) can be computed. This is the probability that the blade centre of the  $i^{th}$  Cloud is off. Over a long period, such a probability can be considered as the percentage of the overall time during which the blade centre can be turned off (e.g., a value of 0.4 means that a blade centre can be turned off, during a year, for a cumulative period equal to 146 days). It can be computed as:

$$P_{off}^i = Pr\{\#OFF_i = s_i\}. \quad (5)$$

**Federation Broker** - The SRN model for the FB of the  $i^{th}$  private Cloud is depicted in Fig. 6. Place  $BQ_i$  represents the FB queue. A token in this place models a VM request waiting for a decision about its destination, i.e., whether it should be accepted locally or it should be forwarded to one of the federated Clouds. Immediate transition  $t\_BQ\_drop_i$  models the drop of a single request and it is associated with the enabling function  $f_{BQ\_drop_i}$  that allows it to be enabled if the maximum queue capacity has been reached, i.e., as soon as the number of tokens in place  $BQ_i$  is equal to  $q_{BQ_i} + 1$ :

$$f_{BQ\_drop_i} = \begin{cases} 1, & \text{if } \#BQ_i = q_{BQ_i} + 1 \\ 0, & \text{otherwise} \end{cases}$$

<sup>1</sup> The choice to model timeouts through the use of exponentially distributed transitions is motivated by the decision to keep the model complexity as low as possible. Other solutions can be considered in order to increase the model accuracy, e.g., the use of phase type distributions as done in [21][22].



**Fig. 6.** The SRN model for the Federation Broker of the  $i^{th}$  Cloud

Transitions  $t_{admit_i}$  and  $T_{wait_i}$  model the federation policy. In particular, the firing of immediate transition  $t_{admit_i}$  extracts a token from place  $BQ_i$  and puts it in place  $AD_i$  representing a request being accepted and enqueued locally. Its enabling function  $f_{admit_i}$  allows it to be enabled if the condition for a request to be accepted by the local Cloud is verified:

$$f_{admit_i} = \begin{cases} 0, & \text{if } \#OFF_i = s_i \text{ OR } \#CQ_i = q_{CQ_i} \text{ OR} \\ & \#SW_i = \#PMG_i = 1 \\ 0, & \text{otherwise} \end{cases}$$

Exponentially distributed transition  $T_{wait_i}$  models the forwarding of the request to a partner Cloud. It is associated with a marking dependent rate equal to  $\#BQ_i \cdot 1/t_{f_i}$  ( $t_{f_i}$  has been defined as the time during which a request needs to wait in queue before being scheduled in a federated Cloud) and with an enabling function  $f_{wait_i}$  that allows it to be enabled only if there is a federated Cloud (either private or public) available to accept a forwarded request. Moreover, the enabling function of transition  $T_{wait_i}$  takes into consideration the maximum number of simultaneous transfer that can be performed (as will be clear in the

following the total number of tokens in places  $FW\_PR_i^p$  and  $FW\_PU_i^q$  needs to be less than  $T_i$ ):

$$f_{wait_i} = \begin{cases} 1, & \text{if } ((\exists p : \#CQ_p + \#BQ_p \leq E_p \text{ AND } \#SW_p = \#OFF_p = \#PMG_p = 0) \text{ OR} \\ & (\exists q : \#AV_q = 1 \text{ AND } \#PR_q < R_q)) \text{ AND} \\ & \sum_p \#FW\_PR_i^p + \sum_q \#FW\_PU_i^q < T_i \\ 0, & \text{otherwise} \end{cases}$$

When transition  $T_{wait_i}$  fires, it puts a token in place  $SLF_i$  and the conflict among the immediate transitions  $t\_forw\_pr_i^p$  and  $t\_forw\_pu_i^q$  models the selection of the federated Cloud (either private or public) to which the request will be forwarded. Each transition in the set  $t\_forw\_pr_i^p$  ( $t\_forw\_pu_i^q$ ) has an associated enabling function  $f_{forw\_pr_i^p}$  ( $f_{forw\_pu_i^q}$ ) that allows it to be enabled if the corresponding Cloud is able to accept the request:

$$f_{forw\_pr_i^p} = \begin{cases} 1, & \text{if } \#CQ_p + \#BQ_p \leq E_p \text{ AND} \\ & \#SW_p = \#PMG_p = \#OFF_p = 0 \\ 0, & \text{otherwise} \end{cases}$$

$$f_{forw\_pu_i^q} = \begin{cases} 1, & \text{if } \#AV_q = 1 \text{ AND } \#PR_q < R_q \\ 0, & \text{otherwise} \end{cases}$$

If more than one Cloud is available, a random choice is performed and the token is moved to the corresponding place  $FW\_PR_i^p$  ( $FW\_PU_i^q$ ). Higher priority is associated to the set of transitions  $t\_forw\_pr_i^p$  in order to give priority to private Clouds with respect to public Clouds. A token in one of such places models a VM image being transferred to the target Cloud. Transitions  $T\_transf\_pr_i^p$  ( $T\_transf\_pu_i^q$ ) with marking dependent rate  $\#FW\_PR_i^p \cdot \lambda_{t_i}$  ( $\#FW\_PU_i^q \cdot \lambda_{t_i}$ ) represent the completion of a transfer process and the insertion of the request in the remote queue, i.e., a token in one of the places  $RQ\_PR_i^p$  ( $RQ\_PU_i^q$ ).

Several measures can be computed based on the model of Fig. 6. First of all, the *FB drop probability* ( $P_{FB_d}^i$ ), i.e., the probability for a request to be dropped due to the  $i^{th}$  FB queue being full, can be computed as:

$$P_{FB_d}^i = Pr\{\#BQ_i = q_{BQ_i}\}. \quad (6)$$

The *expected FB waiting time* ( $T_{FB_w}^i$ ), i.e., the time elapsed from the request arrival to the decision about where to submit it, can be computed, according to Little's law, as follows:

$$T_{FB_w}^i = E\{\#CQ_i\} / (Th\{t_{admit_i}\} + Th\{T_{wait_i}\}). \quad (7)$$

Similarly, the *expected FB transmission time* ( $T_{FB_t}^i$ ) can be computed by considering the time that a request waits before being transferred and the actual transfer time as follows:

$$T_{FB_t}^i = E\{\#SLF_i\} / Th\{T_{wait_i}\} + 1 / \lambda_{t_i} \quad (8)$$

The goal of the FB is to make a decision about where to submit a request. The *local admission probability* ( $P_l^i$ ) can be computed as:

$$P_l^i = Th\{admit_i\} / (Th\{admit_i\} + Th\{wait_i\}) \quad (9)$$

while the *remote admission probability* ( $P_r^i$ ) is simply given by:

$$P_r^i = 1 - P_l^i. \quad (10)$$

Finally, the requests that are redirected to a partner Cloud can be accepted in a private or public Cloud. The *remote admission in a private Cloud probability* ( $P_{r_{pr}}^i$ ) is given by:

$$P_{r_{pr}}^i = \frac{\sum_{p=1}^N Th\{T\_transf\_pr_i^p\}}{\sum_{p=1}^N Th\{T\_transf\_pr_i^p\} + \sum_{q=1}^M Th\{T\_transf\_pu_i^q\}} \quad (11)$$

while the *remote admission in a public Cloud probability* ( $P_{r_{pu}}^i$ ) is, of course, equal to:

$$P_{r_{pu}}^i = 1 - P_{r_{pr}}^i. \quad (12)$$

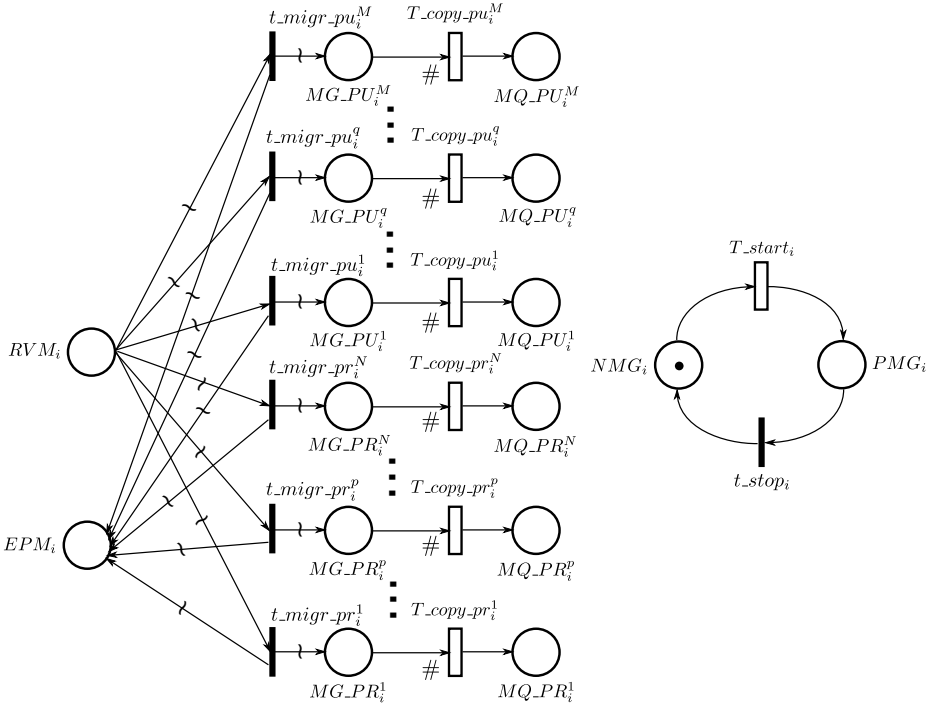
**Migration Manager** - The MM of the  $i^{th}$  private Cloud can be modeled through the SRN depicted in Fig. 7. The structure of the sub-model is very similar to the one related to the FB. Tokens in place  $RVM_i$  represent the VMs currently running in the Cloud blade center while a token in place  $NMG_i$  models the normal operative condition of the Cloud infrastructure in which no consolidation action is being performed. Exponentially distributed transition  $T\_start_i$  models such action to be triggered. In particular, it presents a rate equal to  $1/t_{m_i}$  and it is associated with the enabling function  $f_{start_i}$  that allows it to be enabled only if the particular condition associated to the triggering of a consolidation action is verified:

$$f_{start_i} = 1 = \begin{cases} 1, & \text{if } \#VM_i < M_i \text{ AND } \#CQ_i = \#BQ_i = 0 \text{ AND} \\ & ((\exists p : \#ON_p > C_p \text{ AND } \#SW_p = \#PMG_p = \#OFF_p = 0) \text{ OR} \\ & (\exists q : \#AV_q = 1 \text{ AND } \#PR_q < R_q)) \\ 0, & \text{otherwise} \end{cases}$$

$t_{m_i}$  has been defined as the time during which such condition needs to be verified for a consolidation to be scheduled. As soon as transition  $T\_start_i$  moves a token to place  $PMG_i$ , the consolidation is triggered and one of the transitions  $t\_migr\_pr_i^c$  (with  $1 \leq p \leq N$  and  $p \neq i$ ) and  $t\_migr\_pu_i^q$  (with  $1 \leq q \leq M$ ) will be enabled and will fire. In fact, each of such transitions is associated with an enabling function  $f\_migr\_pr_i^p$  ( $f\_migr\_pu_i^q$ ) that allows it to be enabled if a token is present in place  $PMG_i$  and the corresponding federated private (public) Cloud is able to accept the VMs being migrated:

$$f_{migr\_pr_i^p} = \begin{cases} 1, & \text{if } \#PMG_i = 1 \text{ AND} \\ & \#ON_p > C_p \text{ AND } \#SW_p = \#PMG_p = \#OFF_p = 0 \\ 0, & \text{otherwise} \end{cases}$$





**Fig. 7.** The SRN model for the Migration Manager of the  $i^{th}$  Cloud

$$f_{migr\_pu_i^q} = \begin{cases} 1, & \text{if } \#PMG_i = 1 \text{ AND } \#AV_q = 1 \text{ AND } \#PR_q < R_q \\ 0, & \text{otherwise} \end{cases}$$

Their conflict represents a random choice among the available Clouds. Also in this case, a higher priority is associated to private Clouds. Input and output arcs of transitions  $t_{migr\_pr_i^p}$  ( $t_{migr\_pu_i^q}$ ) have a marking dependent multiplicity. They behave as flushing arcs moving all the tokens from place  $RVM_i$  to one of the  $MG\_PR_i^p$  ( $MG\_PU_i^q$ ) places. Such a particular semantic is pictorially represented with a  $\sim$  on the arc,  $\sim$  meaning  $\#RVM_i$ .

Tokens in such places model the VMs being migrated. The time necessary for the migration to be completed is represented by the exponentially distributed transitions  $T\_copy\_pr_i^p$  ( $T\_copy\_pu_i^q$ ) whose rate is equal to  $\#MG\_PR_i^p \cdot \lambda_{m_i}$  ( $\#MG\_PU_i^q \cdot \lambda_{m_i}$ ) in order to model a concurrent transfer. As soon as the migration is completed the corresponding VM requests are enqueued in the remote queue by inserting tokens in places  $MQ\_PR_i^p$  ( $MQ\_PU_i^q$ ) and transition  $t\_stop_i$  is enabled to fire, modeling the conclusion of the consolidation action. Transition  $t\_stop_i$  has an enabling function  $f\_stop_i$  associated that allows it to be enabled only if no tokens are present in places  $MG\_PR_i^p$  and  $MG\_PU_i^q$ :

$$f_{stop_i} = \begin{cases} 1, & \text{if } (\forall p \Rightarrow \#MG\_PR_i^p = 0) \text{ AND } (\forall q \Rightarrow \#MG\_PU_i^q = 0) \\ 0, & \text{otherwise} \end{cases}$$

Transitions  $t\_migr\_pr_i^p$  ( $t\_migr\_pu_i^q$ ) also put back tokens in place  $EPM_i$  that models the empty blade center PMs. Finally, priorities of transitions  $t\_migr\_pr_i^p$  and  $t\_migr\_pu_i^q$  have to be higher than priority of transition  $t\_stop_i$  in order for the whole mechanism to work properly.

From the model of Fig. 7 the probability that a migration is performed toward a private Cloud ( $P_{m_{pr}}^i$ ) can be computed as:

$$P_{m_{pr}}^i = \frac{\sum_{p=1}^N Th\{t\_migr\_pr_i^p\}}{\sum_{p=1}^N Th\{t\_migr\_pr_i^p\} + \sum_{q=1}^M Th\{t\_migr\_pu_i^q\}} \quad (13)$$

while the probability that a migration is performed toward a public Cloud ( $P_{m_{pu}}^i$ ) is, of course, equal to:

$$P_{m_{pr}}^i = 1 - P_{m_{pu}}^i. \quad (14)$$

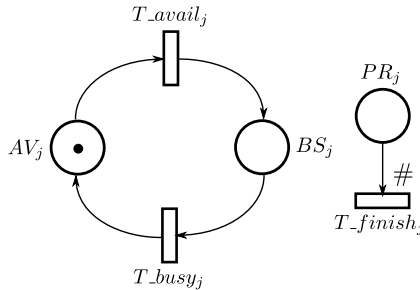


Fig. 8. The SRN model for the  $j^{th}$  public Cloud

**Public Cloud** - Fig. 8 shows the SRN sub-model for the  $j^{th}$  (with  $1 \leq j \leq M$ ) public Cloud. Places  $AV_j$  and  $BS_j$  model the available and busy states, respectively. Exponentially distributed transitions  $T\_avail_j$  and  $T\_busy_j$  represent the switching among the two states. In order to model the availability guaranteed by the public Cloud to its clients, rates  $\lambda_{av_j}$  and  $\lambda_{bs_j}$ , associated to such transitions, have to be set such that  $\lambda_{bs_j}/(\lambda_{av_j} + \lambda_{bs_j}) = a_j$ . Place  $PR_j$  represents the computational resources of the public Cloud, hidden to the external clients, while exponentially distributed transition  $T\_finish_j$  models the execution time of a VM within the public infrastructure. Its rate  $\#PR_j \cdot \lambda_f$  depends on the number of token in place  $PR_j$  in order to model a concurrent execution.

The expected number of VMs running in a public Cloud ( $R_{pu}^j$ ) can be easily computed as:

$$R_{pu}^j = E\{\#PR_j\}. \quad (15)$$

## 4 Modeling Hybrid Federations of Clouds

In this section, we provide the guidelines to compose the SRN sub-models associated to each of the private Cloud building blocks of Fig. 11 and to the generic

public Cloud infrastructure, according to the structure of the Cloud federated environment that has to be analyzed. We also present a case study and show how global measures can be computed on top of the SRN composed model associated to it in order to obtain high level information about energy and management costs.

#### 4.1 Composition Criteria

If no energy, federation, or consolidation management is put in place a simple Cloud infrastructure model can be obtained by composing place  $RQ_i$  in the load sub-model with place  $CQ_i$  of the CM sub-model. In such a way, the tokens representing the VM requests are inserted in the place modeling the CM queue.

On the other side, in a private Cloud infrastructure in which an energy management policy is put in place, the SRN of Fig. 5 has to be composed with the CM sub-model of Fig. 4 in order to represent the on/off cycles that the blade center undergoes. In particular, place  $ON_i$  in the EM sub-model needs to be composed with place  $PM_i$  in the CM sub-model. This is the main motivation for the presence of  $s_i$  tokens in place  $ON_i$  in Fig. 5.

Similarly, the presence of a FB in the  $i^{th}$  private Cloud can be taken into account by composing the SRN represented in Fig. 6 with the SRNs modeling the load and the CM of the same Cloud and the SRNs associated to the federated Clouds. In particular, place  $BQ_i$  in the FB sub-model needs to be composed with place  $RQ_i$  of the load SRN in order to represent the submission of new requests to the FB queue. Similarly, place  $AD_i$  in the SRN associated to the FB has to be composed with place  $CQ_i$  in the CM sub-model so that the local accepted requests are enqueued in the CM queue. Finally, each place  $RQ\_PR_i^p$  ( $RQ\_PU_i^q$ ) needs to be composed with place  $CQ_p$  ( $PR_q$ ) in the SRN associated to the  $p^{th}$  ( $q^{th}$ ) federated private (public) Cloud so that the forwarded requests are actually enqueued in the remote queue of the partner system.

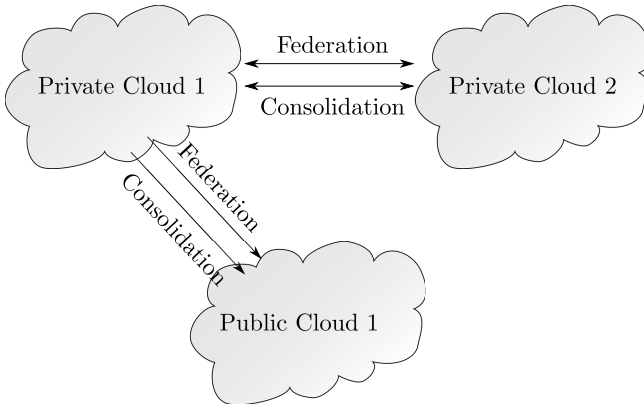
The composition rules in case of presence of a MM in the  $i^{th}$  private Cloud are very similar. The SRN represented in Fig. 7 needs to be composed with the SRN modeling the corresponding Cloud CM and with the sub-models associated to the partner Clouds. In particular, place  $RVM_i$  in the MM sub-model needs to be composed with place  $VM_i$  of the CM SRN in order to represent the migration of currently executing VMs while each place  $MQ\_PR_i^p$  ( $MQ\_PU_i^q$ ) has to be composed with the corresponding place  $CQ_p$  ( $PR_q$ ) in the SRN associated to the  $p^{th}$  ( $q^{th}$ ) federated private (public) Cloud.

With respect to the model scalability, complex scenarios in which several sub-models are composed could give rise to a significant growth of the model state space. Given that the state space cardinality is the parameter that mainly influences the performance of numerical solution techniques, increases in system complexity could make the analytical solution unfeasible. In such cases, it is always possible to study the model through alternative techniques. In the last years, in fact, symbolic techniques have been applied to manage huge state spaces, as described in [23], [24] where a synthesis of the techniques based on appropriate data structures (binary decision diagram and its evolutions) and

Kronecker algebra is done. Thanks to this representation, asynchronous systems with state spaces of size  $10^{60}$  up to  $10^{600}$  can be managed for particular regular systems, as experimentally shown in [25]. Other techniques are also applicable. For example, in [26] interactive sub-models approach is applied to an IaaS Cloud system. In any case, when the state space becomes unmanageable the discrete-event simulation represents a valid alternative, and it can be applied to SRNs without any restrictions either on the state space dimension or on the firing time event distributions.

## 4.2 Case Study

To demonstrate how the proposed modeling approach works, we take into consideration a scenario where a hybrid federation of Clouds is composed of two geographically distributed private IaaS Clouds and one public Cloud, as represented in Fig. 9. We choose to represent a hybrid federation of Clouds in order to highlight the advantages offered by this kind of configuration.



**Fig. 9.** The hybrid Cloud federation under exam

By composing the basic SRN models of the Cloud components described in the previous sections, it is possible to obtain the global SRN model for the actual Cloud infrastructure under analysis. Such a model is shown in Fig. 10 where the presence of three sub-models, each of which is associated to one of the three IaaS Clouds, is highlighted. The corresponding enabling functions are obtained by applying the rules reported in Section 3. The model reported in Fig. 10 corresponds to a configuration where both federation and VM consolidation techniques are used. Other configurations will be used as comparison: one where only the federation is adopted, one composed of the single Cloud 1 with only the Energy Manager, and one composed of the single Cloud 1 without any energy saving strategy. The SRN models of such further configurations can be easily obtained by removing the corresponding sub-models from the model of Fig. 10 according to the composition criteria reported in Section 4.1.

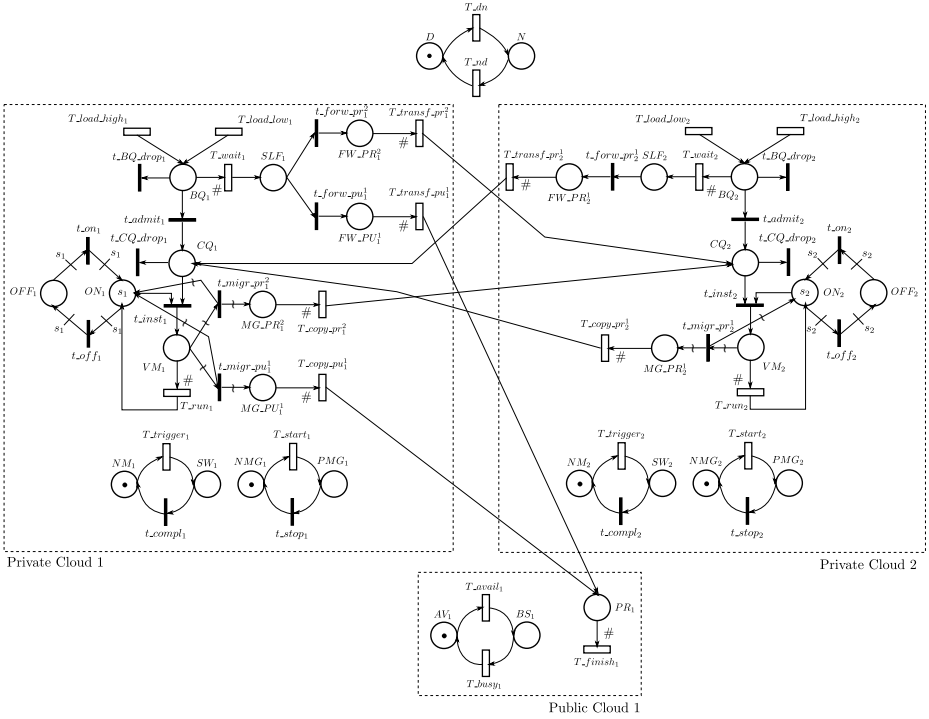


Fig. 10. The SRN model

### 4.3 Global Measures

In addition to the measures derived from a single model (as described in Section 3), a set of global measures can be obtained on top of the composed model. In the following, we will focus our analysis on the private Cloud 1 that is the one that exploits a hybrid federation configuration.

**Expected Waiting Time ( $T_w^1$ )** - This is the time elapsed from the request arrival to the VM instantiation. We have to distinguish among the local waiting time (that is computed summing the time spent in the FB queue and the time spent in the CM queue) and the remote waiting time (that is computed summing the time spent in the FB queue, the time to transfer the VM, and, if the VM is transferred to a private Cloud, the time spent in the remote CM queue). In order to obtain an average index, the local and remote waiting time can be composed by a weighted sum with respect to the local and remote admission probabilities:

$$T_w^1 = T_{FB_w}^1 + T_{CM_w}^1 \cdot P_l^1 + T_{FB_t}^1 \cdot P_r^1 + T_{CM_w}^2 \cdot P_r^1 \cdot P_{rpr}^1. \quad (16)$$

**Drop Probability ( $P_d^1$ )** - This is the probability that a request is dropped by Cloud 1. It can be computed combining the drop probability of the FB queue with the drop probability of the CM queue:

$$P_d^1 = P_{FB_d}^1 + (1 - P_{FB_d}^1) \cdot P_{CM_d}^1. \quad (17)$$

**Migration Probability ( $P_m^1$ )** - This is the probability that a VM is migrated due to a VM consolidation action. It can be computed as follows:

$$P_m^1 = (Th\{t_{migr\_pr_1^2}\} + Th\{t_{migr\_pu_1^1}\})/Th\{t_{inst_1}\}. \quad (18)$$

**Weekly Operating Costs ( $O_c^1$ )** - The operating costs can be computed by summing over a week the costs related to the energy consumptions and the costs related to the use of the public Cloud:

$$O_c^1 = OT * 3600 \cdot [(1 - P_{off}^1) \cdot EC/3600 + R_{pu}^1 \cdot CC/3600] \quad (19)$$

where  $OT$  is the observation time (a week) in hours,  $EC$  is the energy cost per hour, and  $CC$  is the public Cloud cost per hour related to the use of a single VM.

**Expected Number of Accepted Users During a Week ( $A_u^1$ )** - This is the number of users served by Cloud 1 during a week and it can be computed starting from the number of requests and the drop probability:

$$A_u^1 = OT \cdot A^1 \cdot (1 - P_d^1). \quad (20)$$

## 5 Numerical Results

In this section, we focus on the performance, in terms of energy saving, that can be reached using federation and VM consolidation techniques and we also quantify the impact of such techniques on the QoS perceived by the users. Our aim is to provide a quantitative evaluation of the costs/benefits associated to a particular cloud strategy, thus allowing system managers to properly set the configuration parameters with respect to a specific working condition.

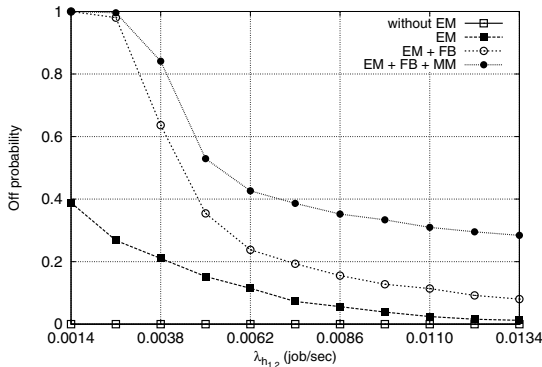
The model of Fig. 10 was solved using the WebSPN simulation tool [27], and fixing the model parameters as listed in Table 1. For the sake of simplicity and without loss of generality, we consider two Clouds with the same parameters. Such parameters have been set in accordance to the relevant literature. In particular, Cloud dimensions, VM execution times, and buffer sizes have been chosen to represent a medium size Cloud as reported in [26]. The arrival rate has been correspondingly set in order to produce as high as possible utilization values as usually desired by Cloud providers in real scenarios. Migration times are calculated considering the average dimension of a typical VM and the standard WAN network bandwidth. However, due to the large variety of Cloud systems, other parameter configurations can be easily adopted, without invalidating the model capabilities.

**Table 1.** Parameter configuration

Parameter	Value	Parameter	Value
$N$	2	$M$	1
$s_{1,2}$	32	$E_{1,2}, M_{1,2}$	5
$P_{1,2}$	[1,6]	$R_1$	20
$T_{1,2}$	10	$C_{1,2}$	2
$q_{BQ_{1,2}}, q_{CQ_{1,2}}$	10	$a_1$	0.98
$\lambda_{h_{1,2}}^{-1}$	[75,715] (sec)	$\lambda_{t_{1,2}}^{-1}$	$10 * \lambda_{h_{1,2}}^{-1}$
$\lambda_{dn}^{-1}, \lambda_{nd}^{-1}$	12 (hours)	$\lambda_{r_{1,2}}^{-1}, \lambda_{f_1}^{-1}$	1 (hours)
$\lambda_{t_{1,2}}^{-1}, \lambda_{m_{1,2}}^{-1}$	30 (min)	$t_{off_{1,2}}, t_{f_{1,2}}, t_{m_{1,2}}$	60 (sec)
$OT$	168 (hours)	$EC$	2 (\$/hour)
$CC$	0.2 (\$/hour · VM)		

Simulation has been carried out by performing 5,000 independent runs and by asking for a confidence level of 90%. However, the confidence intervals are not shown in the following graphs since they are very tight. The final measures have been obtained in the order of hours on a single processor core.

Fig. 11 shows the steady state probability that the blade centre of Cloud 1 is off versus the arrival rate. If Cloud 1 adopts only an energy manager, it can

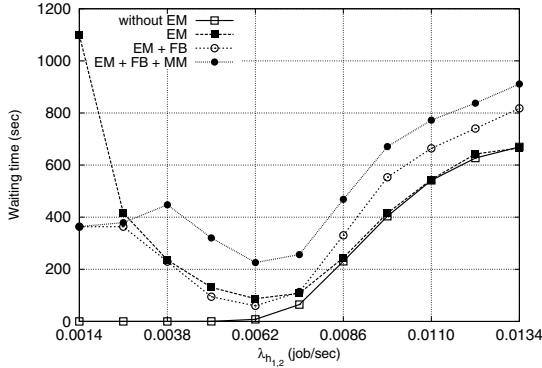


**Fig. 11.** Probability  $P_{off}^1$  that a blade centre is off versus the arrival rate varying the energy saving strategies

be observed that the off probability is about 0.4 when the system is low loaded and it decreases when the arrival rate increases reaching a value near 0. Remarkable improvements can be obtained by using the federation. In fact, redirecting, when possible, the load toward the other Clouds we are able to increase the off probability. However, if only the federation technique is adopted, such improvements become less evident when the load increases. The best performance can be obtained by using also the VM consolidation technique. In this way, the system is able to adapt its behavior to the actual load, thus improving the energy efficiency. For example, when  $\lambda_{h_{1,2}} = 0.0134$  job/sec the federation technique

does not allow us to frequently turn off the blade centre, while through the VM consolidation technique an off probability near to 0.3 is reached.

A QoS-oriented performance analysis can be carried out by investigating the delays introduced in the service provisioning. Fig. 12 shows the waiting time perceived by users with respect to the different energy saving strategies. Such a value is compared with the waiting time obtained using a blade centre always switched on, in order to distinguish the delays due to the energy saving techniques from those related to the blade centre capacity. We can highlight two

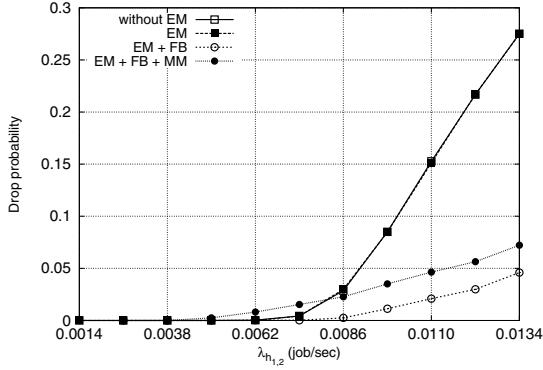


**Fig. 12.** Waiting time  $T_w^1$  versus the arrival rate varying the energy saving strategies

different conditions. When the load is low ( $\lambda_{h_{1,2}} < 0.074$  job/sec) the waiting time associated to the blade centre always switched on is almost null. On the other hands, if an energy saving strategy is adopted (curve 'EM'), it is possible to observe that the waiting time increases when the load decreases due to the time the blade centre is off while waiting for a consistent number ( $P_1$ ) of users to serve. When the load increases ( $\lambda_{h_{1,2}} > 0.074$  job/sec), the waiting time associated to the energy saving strategy is equal to that of a blade centre always switched on. From the analysis of such a figure, it is also possible to quantify the QoS reductions due to the federation and VM consolidation techniques. For example, when  $\lambda_{h_{1,2}} = 0.0110$  job/sec the waiting time associated to the adoption of the federation (curve 'EM + FB') is about 620 sec and it reaches a value of about 800 sec if also the VM consolidation technique is used (curve 'EM + FB + MM') due to the increased number of migrated VMs from the federated private Cloud 2. Such values can be compared with the waiting time experienced by users when no energy saving policies are adopted. In this way, it is possible to find a trade off between the advantages obtained in terms of power saving (see Fig. 11) and the effects on the QoS perceived by the users.

Another QoS index is represented by the probability that a request is dropped (see Fig. 13). In this case, we can appreciate the improvements, in terms of perceived QoS, produced by the federation and consolidation techniques. In fact, due to the presence of the public Cloud that is characterized by a high availability, the drop probability can be maintained to a low value, notwithstanding

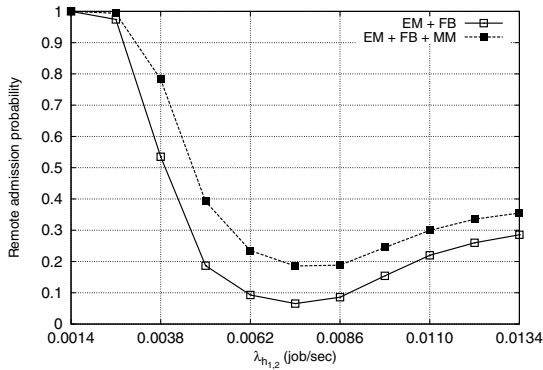




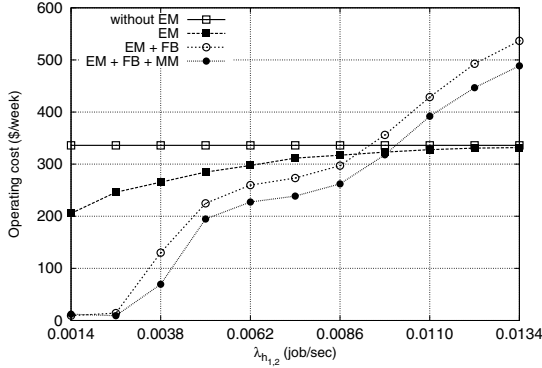
**Fig. 13.** Drop probability  $P_d^1$  versus the arrival rate varying the energy saving strategies

the system load. On the contrary, if the Cloud is not federated, the probability rapidly increases when the arrival rate increases and the system reaches a saturation condition.

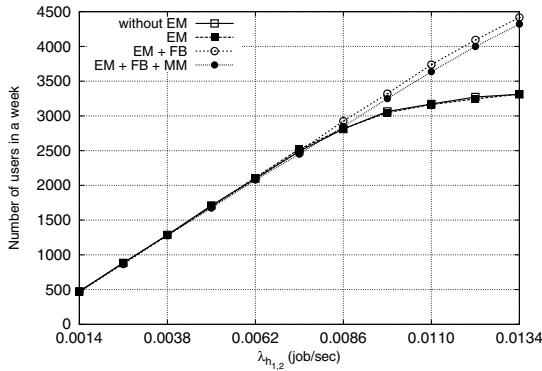
From the provider point-of-view, interesting indexes can be obtained measuring the probability that a new VM is redirected from Cloud 1 to federated Clouds (see Fig. 14). From such an index it is possible to estimate the QoS levels perceived by the users: in fact, foreign VMs could be affected by latency or sudden disconnections due to the network conditions. It can be noted that when the load is low such a probability decreases. This is due to the fact that the probability that Cloud 1 is off decreases and then the VMs can be locally admitted. When the system is overloaded the probability increases because the Cloud 1 reaches its capacity and the incoming request have to be redirected to the public Cloud.



**Fig. 14.** Probability  $P_r^1$  that a VM is redirected from Cloud 1 to federated Clouds versus the arrival rate



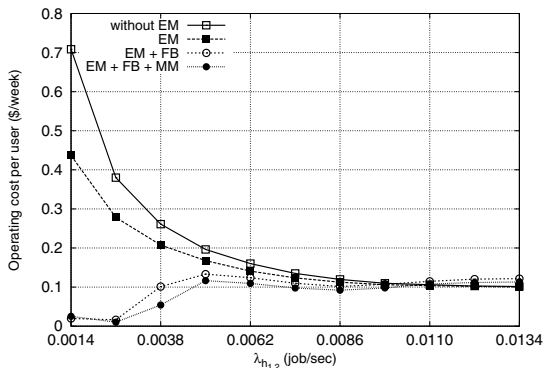
**Fig. 15.** Weekly operating costs  $O_c^1$  versus the arrival rate varying the energy saving strategies



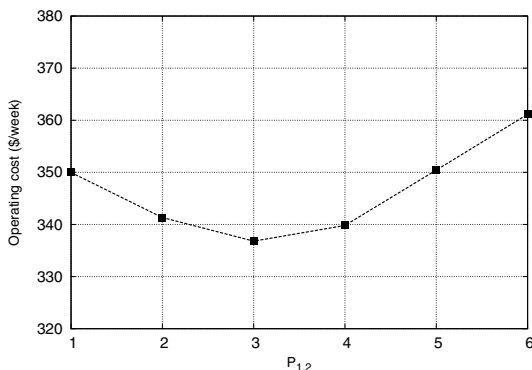
**Fig. 16.** Number of accepted users  $A_u^1$  during a week

From a combination of the proposed indexes, it is possible to carry out a costs/benefits analysis, by quantifying both the advantages (in terms of energy/money saving) due to the blade centre switch off and the penalties related to possible SLA violations.

For example, it is possible to compute the operating costs as the sum of the energy costs and the costs related to the public Cloud. Fig. 15 shows the operating costs obtained varying the energy saving strategies. Such costs can be compared with the costs related to a Cloud always active (curve 'without EM'). If only the Energy Manager is adopted (curve 'EM') it is possible to obtain only a moderated cost reduction and only if the load is very low. On the contrary, using the federation and VM consolidation techniques it is possible to adapt the costs to the system load thus obtaining a quasi-linear trend. In this case, it is possible to observe that when the load is high, the operating costs are higher than those of the single Cloud without federation. This is due to the fact that the number of accepted users is higher. In fact, observing Fig. 16 it



**Fig. 17.** Weekly operating cost per user versus the arrival rate varying the energy saving strategies



**Fig. 18.** Weekly operating cost  $O_c^1$  versus the power on threshold  $P_1$

is possible to observe that, thanks to the federation with the public Cloud, the number of accepted users linearly increases with the arrival rate (curves 'EM + FB' and 'EM + FB + MM') and the system does not saturate even if the number of requests exceed its capacity. Such considerations can be summarized by calculating the operating cost per user, as shown in Fig. 17. In this graph we can observe that when the load is low the federate Clouds allow to reduce the operating cost per user through the exploitation of opportune energy saving strategies. Such advantages become irrelevant when the system is high loaded. In fact, in this case the operating costs per user are similar notwithstanding the adopted strategy. However, in such a condition the benefits related to the use of federated Clouds are related to the capacity to serve a greater number of users without affect the operating cost per user and without the need to carry out investments to increase the Cloud capacity in terms of available CPU.

Finally, let us give an example of how the model can be used to obtain useful insights on the system parameter settings. Fig. 18 shows the variations on

the weekly operating costs obtained by changing the power on threshold, i.e., the number of users that the system has to wait before a power on action is triggered. Such a parameter influences the system behavior by forcing the Cloud infrastructure to be always on when its value is low, or by making a strong use of federated Clouds when its value is high. We can observe that an optimal parameter configuration can be obtained by balancing the costs related to the energy consumption and to the use of the public Cloud. Such an example demonstrates the power of the proposed model and its usefulness in the management of a real Cloud system.

## 6 Related Work

Modeling and performance evaluation are still emerging topics in the Cloud area. From an analysis of the state of the art, we can observe that the majority of works study the problem by resorting to simulation [28, 29, 30] or by conducting experimental trials on real Cloud systems [31, 32, 33] in order to monitor and benchmark cloud based applications [15, 34].

To the best of our knowledge, only few works propose a formal analytical study also focusing on the provider point-of-view. A first attempt to adopt state-space Markovian techniques to model Cloud systems is found in [26]. An interacting stochastic model approach is used in order to reduce the model complexity and to analyze very large systems. The proposed models are represented by continuous time Markov chains and take into account aspects related to the system reliability and system performability. The work mainly focuses on the analytical technique while the analysis part shows only some examples with a limited system size. In a subsequent work, authors extend the applicability of the proposed model to the energy consumption analysis [35]. However, there are some differences between such a work and our proposed model. In fact, the authors model a system broker able to balance the load among three server pools characterized by a different status of the PMs: hot (running), warm (turned on, but not ready), and cold (turned off). They do not take into consideration the possibility to extend the broker with federation capabilities as presented here. Moreover, no VM consolidation techniques are taken into consideration within the same pool or among different pools. Nevertheless, the interacting model approach is very interesting and it could be also used to extend our model to more complex scenarios. Another adopted technique to model Cloud systems is queueing theory (e.g., see [36] and references therein).

With respect to Cloud workload characterization, the arrival process in Cloud systems can be hardly predictable and/or variable with the time of day. For example, in [37] authors show how periodic patterns with a time period of one day can be observed in the workload of the production data center of a multinational company. In [38], Pacheco-Sanchez *et al.* demonstrate that Markovian arrival processes can be adopted as a tool for performance prediction of servers deployed in the Cloud. For such reasons, we choose to characterize the Cloud workload with an MMPP arrival process.

Other interesting works, mainly focusing on performance evaluation of Cloud systems, are the following. Yigitbasi *et al.* [39] proposed an experimental framework to analyze the resource acquisition and resource release times in Amazon EC2 under variety of workloads while Voorsluys *et al.* [40] showed performance analysis of VM live migration and its impacts on SLAs. In [41], Wu *et al.* provided performance models for live migration which can be used to predict the VM migration time given the application behavior and the resources available for migration. Our analytic model can benefit from these studies for parameter tuning. In [42], Gmach *et al.* proposed three different cost models that can be used by service providers who make significant investments in new virtualized data centers in order to recover costs for infrastructure resources. They presented a detailed study involving hundreds of workloads in order to demonstrate the results. Such a work can be used to validate our model and to demonstrate how it can be exploited for cost analysis of IaaS Clouds. In [43], Govindan *et al.* described a practical technique for predicting performance interference due to shared processor caches among VMs consolidated on the same PM. In [44], Rhoden *et al.* focused on per-node efficiency, performance, and predictability in Cloud data centers. The results of these work could be used to complement our model and improve our placement decisions for given performance and cost objectives. In [45], Goudarzi *et al.* considered SLA-based resource allocation problem for multi-tier applications in Cloud. The processing, memory requirement, and communication resources were considered as three search dimensions, in which optimization was performed. Such a work could be useful to extend our performance model to consider heterogeneous requests. In [46], Lenk *et al.* proposed a new method for performance evaluation of IaaS Clouds by taking into account the type of application running in a VM and can be used to evaluate the performance actually available on a certain IaaS platform. Validation of our performance model can be performed w.r.t this work.

There is a considerable literature more related to the improvement of the power consumption of computer systems. For an overview of recent developments related to Cloud Computing, see [47]. Some economic heuristics for the allocation of servers in a cluster were presented in [48]. More recently, in [49] a queueing model is used to analyze the problem of managing a service center where clients leave the system if they have to wait too long before starting service. In such a work, a block of servers is designated as reserve and the reserves are powered up when the number of jobs in the system is sufficiently high, and are powered down when that number is sufficiently low. The question of how to choose the number of reserves, and the up and down thresholds are answered. Similar models were examined in [50], [51], [52]. The main difference with the proposed approach, apart from the analytic technique used, is related to federation and VM migration policies that are not faced in such works.

## 7 Conclusions

This paper addressed the problem of energy management in a Cloud environment. We presented a SRN model that was successfully solved in order to analyze

several management strategies of IaaS Clouds with the final objective to reduce energy costs. We are currently organizing a real testbed where the presented strategies will be implemented and their impact measured and compared against the analytical results presented in this paper, to further validate our modeling approach.

## References

1. Murugesan, S.: Harnessing green it: Principles and practices. *IT Professional* 10(1), 24–33 (2008)
2. Foster, I., Zhao, Y., Raicu, I., Lu, S.: Cloud computing and grid computing 360-degree compared. In: *Grid Computing Environments Workshop, GCE 2008*, pp. 1–10 (2008)
3. Eucalyptus official site, <http://www.eucalyptus.com/>
4. Amazon EC2, <http://aws.amazon.com/ec2>
5. IBM SBDTC, <http://www-180.ibm.com/cloud/enterprise/beta/dashboard>
6. IBM Cloud Computing, <http://www.ibm.com/ibm/cloud/>
7. Eucalyptus official site, <http://www.eucalyptus.com/>
8. openqrm official site, <http://www.openqrm.com/>
9. Rochwerger, B., Breitgand, D., Epstein, A., Hadas, D., Loy, I., Nagin, K., Tordsson, J., Ragusa, C., Villari, M., Clayman, S., Levy, E., Maraschini, A., Massonet, P., Muñoz, H., Tofetti, G.: Reservoir - when one cloud is not enough. *Computer* 44(3), 44–51 (2011)
10. Ye, K., Huang, D., Jiang, X., Chen, H., Wu, S.: Virtual machine based energy-efficient data center architecture for cloud computing: A performance perspective. In: *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pp. 171–178 (2010)
11. Eugen Feller, D.L., Morin, C.: State of the art of power saving in clusters + results from the EDF case study. Tech. Rep. (2010)
12. Takeda, S., Takemura, T.: A rank-based VM consolidation method for power saving in datacenters. *IPSSJ Online Transactions* 3, 88–96 (2010)
13. Liu, L., Wang, H., Liu, X., Jin, X., He, W.B., Wang, Q.B., Chen, Y.: GreenCloud: a new architecture for green data center. In: *Proceedings of the 6th International Conference Industry Session on Autonomic Computing and Communications Industry Session*, pp. 29–38 (2009)
14. Bruneo, D., Scarpa, M., Puliafito, A.: Performance evaluation of glite grids through gspns. *IEEE Transactions on Parallel and Distributed Systems* 21(11), 1611–1625 (2010)
15. Yigitbasi, N., Iosup, A., Epema, D., Ostermann, S.: C-Meter: A framework for performance analysis of computing Clouds. In: *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pp. 472–477 (2009)
16. Buyya, R., Ranjan, R., Calheiros, R.N.: Modeling and simulation of scalable cloud computing environments and the cloudsim toolkit: Challenges and opportunities. In: *HPCS* (2009)
17. Bruneo, D., Longo, F., Puliafito, A.: Evaluating energy consumption in a cloud infrastructure. In: *IEEE WoWMoM*, pp. 1–6 (June 2011)

18. Ciardo, G., Blakemore, A., Chimento, P.F., Muppala, J.K., Trivedi, K.S.: Automated generation and analysis of Markov reward models using stochastic reward nets. *IMA Volumes in Mathematics and its Applications: Linear Algebra, Markov Chains, and Queueing Models* 48, 145–191 (1993)
19. Rochwerger, B., Breitgand, D., Levy, E., Galis, A., Nagin, K., Llorente, I.M., Montero, R., Wolfsthal, Y., Elmroth, E., Cáceres, J., Ben-Yehuda, M., Emmerich, W., Galán, F.: The reservoir model and architecture for open federated cloud computing. *IBM J. Res. Dev.* 53, 535–545 (2009)
20. Marsan, M.A., Balbo, G., Conte, G.: A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems* 2, 93–122 (1984)
21. Machida, F., Kim, D.S., Trivedi, K.: Modeling and analysis of software rejuvenation in a server virtualized system. In: 2010 IEEE Second International Workshop on Software Aging and Rejuvenation, WoSAR, pp. 1–6 (November 2010)
22. Bruneo, D., Distefano, S., Longo, F., Puliafito, A., Scarpa, M.: Evaluating wireless sensor node longevity through markovian techniques. *Computer Networks* 56(2), 521–532 (2012)
23. Ciardo, G., Lüttgen, G., Siminiceanu, R.I.: Efficient Symbolic State-Space Construction for Asynchronous Systems. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 103–122. Springer, Heidelberg (2000)
24. Ciardo, G., Marmorstein, R., Siminiceanu, R.: The saturation algorithm for symbolic state space exploration. *Software Tools for Technology Transfer* 8(1), 4–25 (2006)
25. Miner, A.S., Ciardo, G.: Efficient Reachability Set Generation and Storage Using Decision Diagrams. In: Donatelli, S., Kleijn, J. (eds.) ICATPN 1999. LNCS, vol. 1639, pp. 6–25. Springer, Heidelberg (1999)
26. Ghosh, R., Trivedi, K., Naik, V., Kim, D.S.: End-to-end performability analysis for infrastructure-as-a-service cloud: An interacting stochastic models approach. In: 2010 IEEE 16th Pacific Rim International Symposium on Dependable Computing, PRDC, pp. 125–132 (December 2010)
27. Bobbio, A., Puliafito, A., Scarpa, M., Telek, M.: Webspn: A web-accessible Petri net tool. In: Conference on Web-Based Modeling & Simulation (1998)
28. Buyya, R., Ranjan, R., Calheiros, R.: Modeling and simulation of scalable cloud computing environments and the cloudsim toolkit: Challenges and opportunities. In: International Conference on High Performance Computing Simulation, HPCS 2009, pp. 1–11 (June 2009)
29. Kim, J.H., Lee, S.M., Kim, D.S., Park, J.S.: Performability analysis of iaas cloud. In: 2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, IMIS, June 30–July 2, pp. 36–43 (2011)
30. Iosup, A., Yigitbasi, N., Epema, D.: On the performance variability of production cloud services. In: 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid, pp. 104–113 (May 2011)
31. Iosup, A., Ostermann, S., Yigitbasi, M., Prodan, R., Fahringer, T., Epema, D.: Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Transactions on Parallel and Distributed Systems* 22(6), 931–945 (2011)
32. Stantchev, V.: Performance evaluation of cloud computing offerings. In: Third International Conference on Advanced Engineering Computing and Applications in Sciences, ADVCOMP 2009, pp. 187–192 (October 2009)

33. Ostermann, S., Iosup, A., Yigitbasi, N., Prodan, R., Fahringer, T., Epema, D.: A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing. In: Avresky, D.R., Diaz, M., Bode, A., Ciciani, B., Dekel, E. (eds.) *Cloudcom 2009*. LNICST, vol. 34, pp. 115–131. Springer, Heidelberg (2010)
34. Koeppe, F., Schneider, J.: Do you get what you pay for? using proof-of-work functions to verify performance assertions in the cloud. In: 2010 IEEE Second International Conference on Cloud Computing Technology and Science, *CloudCom*, November 30–December 3, pp. 687–692 (2010)
35. Ghosh, R., Naik, V.K., Trivedi, K.S.: Power-performance trade-offs in iaas cloud: A scalable analytic approach. In: *Dependable Systems and Networks Workshops*, pp. 152–157 (2011)
36. Khazaei, H., Mistic, J., Mistic, V.: Performance analysis of cloud computing centers using m/g/m/m + r queueing systems. *IEEE Transactions on Parallel and Distributed Systems* PP(99), 1 (2011)
37. Verma, A., Dasgupta, G., Nayak, T.K., De, P., Kothari, R.: Server workload analysis for power minimization using consolidation. In: *Proceedings of the 2009 Conference on USENIX Annual Technical Conference, USENIX 2009*, p. 28. USENIX Association, Berkeley (2009), <http://dl.acm.org/citation.cfm?id=1855807.1855835>
38. Pacheco-Sanchez, S., Casale, G., Scotney, B., McClean, S., Parr, G., Dawson, S.: Markovian workload characterization for qos prediction in the cloud. In: *IEEE International Conference on Cloud Computing*, pp. 147–154 (2011)
39. Yigitbasi, N., Iosup, A., Epema, D.: C-meter: A framework for performance analysis of computing clouds. In: *International Workshop on Cloud Computing (2009)*
40. Voorshuys, W., Broberg, J., Venugopal, S., Buyya, R.: Cost of Virtual Machine Live Migration in Clouds: A Performance Evaluation. In: Jaatun, M.G., Zhao, G., Rong, C. (eds.) *CloudCom 2009*. LNCS, vol. 5931, pp. 254–265. Springer, Heidelberg (2009)
41. Wu, Y., Zhao, M.: Performance modeling of virtual machine live migration. In: *2011 IEEE International Conference on Cloud Computing, CLOUD*, pp. 492–499 (July 2011)
42. Gmach, D., Rolia, J., Cherkasova, L.: Resource and virtualization costs up in the cloud: Models and design choices. In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks, DSN*, pp. 395–402 (June 2011)
43. Govindan, S., Liu, J., Kansal, A., Sivasubramaniam, A.: Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines. In: *Proceedings of the 2nd ACM Symposium on Cloud Computing, SOCC 2011*, pp. 22:1–22:14. ACM, New York (2011)
44. Rhoden, B., Klues, K., Zhu, D., Brewer, E.: Improving per-node efficiency in the datacenter with new os abstractions. In: *Proceedings of the 2nd ACM Symposium on Cloud Computing, SOCC 2011*, pp. 25:1–25:8. ACM, New York (2011)
45. Goudarzi, H., Pedram, M.: Multi-dimensional sla-based resource allocation for multi-tier cloud computing systems. In: *2011 IEEE International Conference on Cloud Computing, CLOUD*, pp. 324–331 (July 2011)
46. Lenk, A., Menzel, M., Lipsky, J., Tai, S., Offermann, P.: What are you paying for? performance benchmarking for infrastructure-as-a-service offerings. In: *2011 IEEE International Conference on Cloud Computing, CLOUD*, pp. 484–491 (July 2011)
47. Berl, A., Gelenbe, E., Di Girolamo, M., Giuliani, G., De Meer, H., Dang, M.Q., Pentikousis, K.: Energy-efficient cloud computing. *The Computer Journal* 53(7), 1045–1051 (2010), <http://comjnl.oxfordjournals.org/content/53/7/1045.abstract>



48. Chase, J.S., Anderson, D.C., Thakar, P.N., Vahdat, A.M., Doyle, R.P.: Managing energy and server resources in hosting centers. In: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles, SOSP 2001, pp. 103–116. ACM, New York (2001), <http://doi.acm.org/10.1145/502034.502045>
49. Mitrani, I.: Service center trade-offs between customer impatience and power consumption. *Perform. Eval.* 68, 1222–1231 (2011), <http://dx.doi.org/10.1016/j.peva.2011.07.017>
50. Artalejo, J.R., Economou, A., Lopez-Herrero, M.J.: Analysis of a multiserver queue with setup times. *Queueing Syst. Theory Appl.* 51, 53–76 (2005), <http://dl.acm.org/citation.cfm?id=1110977.1111005>
51. Gandhi, A., Harchol-Balter, M., Adan, I.: Server farms with setup costs. *Perform. Eval.* 67, 1123–1138 (2010), <http://dx.doi.org/10.1016/j.peva.2010.07.004>
52. Mazzucco, M., Dyachuk, D., Dikaiakos, M.: Profit-aware server allocation for green internet services. In: Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2010, pp. 277–284. IEEE Computer Society, Washington, DC (2010), <http://dx.doi.org/10.1109/MASCOTS.2010.36>

# A SAN-Based Modeling Approach to Performance Evaluation of an IMS-Compliant Conferencing Framework

Stefano Marrone<sup>1</sup>, Nicola Mazzocca<sup>2</sup>, Roberto Nardone<sup>2</sup>, Roberta Presta<sup>2</sup>,  
Simon Pietro Romano<sup>2</sup>, and Valeria Vittorini<sup>2</sup>

<sup>1</sup> Seconda Università di Napoli, Dipartimento di Matematica  
viale Lincoln, 5 - 81100, Caserta, Italy  
`stefano.marrone@unina2.it`

<sup>2</sup> Università di Napoli “Federico II”, Dipartimento di Informatica e Sistemistica  
via Claudio, 21 - 80125, Napoli, Italy  
{`nicola.mazzocca, roberto.nardone, roberta.presta, spromano,`  
`valeria.vittorini`}@unina.it

**Abstract.** This paper proposes a Stochastic Activity Networks (SANs) based approach to performance analysis of a conferencing framework compliant with the IP multimedia core network subsystem specification. The proposed approach relies on the OsMoSys modeling methodology and applies some concepts of component software engineering to the development of formal models. The paper introduces the possibility of building template models in order to enable the definition of families of models and describes the implementation of a library of reusable SANs modeling the components of the conferencing framework. Starting from the model library, we analyze the performance of a current implementation of the conferencing framework by instantiating and composing SAN models. Scalability and computational complexity are also addressed. We validate the resulting model of the conferencing system and discuss the advantages of the proposed approach through a comparative analysis with the results of an experimental campaign conducted over a real-world testbed implementation.

**Keywords:** Stochastic Activity Networks, models composition, performance evaluation, network management and monitoring, IP Multimedia Subsystem.

## 1 Introduction

The IP Multimedia Subsystem (IMS) is a standardized Next Generation Networking (NGN) architecture that has been conceived for telecom operators willing to provide advanced services on top of both mobile and fixed networks [1]. In the IMS envisioned scenario, heterogeneous devices are supported and users have to be able to ubiquitously exploit the entire portfolio of available services, which entails support for roaming as well as for flexible and transparent adaptation to

context changes. To achieve the aforementioned goals, IMS makes extended use of Voice over IP (VoIP) technologies and open IP protocols.

Conferencing services are one of the most challenging among those engineered on top of such framework. They offer advanced communication experience to end-users exploiting a combination of audio, video, instant messaging, desktop sharing and other media, and impose a number of stringent requirements to the underlying network infrastructure.

Due to its recent birth, the IMS architecture is currently far from reaching its steady-state with respect to the complete definition of the overall infrastructure and related standards. Furthermore, to date only a few early trials and deployments of the architecture are underway. This leaves space to a number of open issues that still have to be faced, both at the infrastructure and at the service level, as well as, crosswise, for a systematic approach to the performance evaluation.

In this paper we introduce and apply an extension of the OsMoSys modeling methodology [2] to an IMS-compliant implementation of a conferencing platform [3] in order to perform performance and QoS (Quality of Service) analysis of the system. Specifically, the paper describes the development process of a library of reusable Stochastic Activity Networks (SANs) modeling the components of the conferencing framework. The contribution of this work is twofold: i) it describes a case study motivated by a real need, providing the performance and scalability models of the conferencing system, ii) it proposes a modeling approach which copes with both reuse and complexity of models. To this aim the paper introduces the concept of *Model Templates* enabling the definition of a family of models from one model description, as well as the usage of *model stubs* which can be used in order to reduce both simulation time and memory consumption during the analysis phase.

The paper is organized as follows. First the conferencing framework is described in Section 2 to provide the reader with the needed background information. We motivate and position our work in Section 3. Section 4 presents the methodological contribution of the paper and introduces the concept of *Model Template*. The SANs models obtained by applying the modeling process are presented in Section 5. In Section 6 the role played by model stubs and reduction techniques is discussed. The models are validated in Section 7 through a comparative analysis with the results of an experimental campaign conducted over a real-world testbed implementation of an IMS conferencing framework. We show the advantages of the proposed modeling approach by exploiting both complete and reduced models. Finally Section 8 contains some concluding remarks and discusses some directions of our future work.

## 2 Conferencing Framework

In this Section we help the reader understand the main aspects related to the conferencing system which has been the subject of our modeling efforts.

The conferencing platform we analyzed offers advanced conferencing features: system users, named “participants” or “conferencing clients”, are enabled to

create and join conferences involving any kind of media stream, including audio and video, as well as instant messaging or even gaming. Different conference kinds (“blueprints”) are supported and can be highly customized by users aiming to create conferences best fitting their needs. The system is conceived to be scalable, i.e., able to support an increasing number of conferencing clients. This property is obtained by a proper co-operation mechanism among different “centralized conferencing islands”, which will be illustrated afterwards.

As already mentioned, the conferencing system has the considerable advantage of being realized by exploiting standard architectural design specifications and protocols, deployed by eminent standardization organizations of both the Internet and the Telecommunication communities, namely the 3GPP and the IETF (Internet Engineering Task Force). As for the IMS specification, it complies with the standard defined in the 3GPP document [4] describing how IMS logical elements can be orchestrated in order to make the IMS network capable to support conferencing services. On the IETF hand, the system is an actual implementation of the RFC document dedicated to Centralized Conferencing (also known by the acronym “XCON”) [5], that defines both the protocols and the components needed to fit advanced conferencing service requirements. For the sake of conciseness, we herein describe only the main concepts and entities of the aforementioned standards that are relevant for this work.

Based on the mentioned 3GPP specification, two logical planes can be identified: the “control plane” and the “media plane”. The control plane deals with issues related to the set-up of the multi-media multi-user communication, as well as those related to overcoming the heterogeneity of both the access networks and the end-user devices. The media plane faces all the matters related to the media flows transmitted and received by users, such as, for example, the transcoding across different media formats and the switching of different media streams among conferencing participants.

The most important entities for our work are: *User Equipment (UE)*: the device used by the conferencing user to participate in the conference; *Application Server (AS)*: the server-side entity responsible for the implementation of the conferencing application logic; *Media Resource Function Controller (MRFC)*: the logical entity devoted to control operations on media streams according to information provided by the AS; *Media Resource Function Processor (MRFP)*: the entity in charge to perform media processing by acting as a media mixer.

According to the IETF XCON standard, the conferencing application logic involves different functionality, including the signaling management for the call set-up, the delivery of conference notifications to participants, the creation and modification of conference instances according to user preferences, the handling of moderation and so on.

Given this bird’s-eye overview of the main reference standards, we now provide further details of the real-world conferencing system we took under analysis. In such system, the AS provides all the XCON functionality. Moreover, it acts as a MRFC by managing the MRFP, which is implemented as a different component that we will call from now on “media mixer”. According to the identified logical

entities, the IMS elements have been replaced in the system with real-world components, either by properly extending existing open source components (like, e.g., in the case of the Application Server elements), or by creating them from scratch (like, e.g., in the case of the media mixer). The resulting IMS compliant architecture, as well as its implementation details, are introduced in [3,6,7].

In what follows, two different conferencing scenarios are considered: a centralized conferencing scenario and a distributed conferencing scenario. The second one has been introduced in order to improve the scalability of the conferencing framework. A recent implementation of such scenario has been proposed in [3].

## 2.1 Centralized Conferencing Scenario

Figure 1 presents a simplified view of the system, showing the main IMS components we mentioned before. We refer to such IMS cloud as to a “centralized island” of the system, since all signaling messages from conferencing clients are directed to the same centralized AS and hence realize, at the level of the control plane, a typical star topology. Being the center of such topology, the AS is also called “conferencing focus”, or simply “focus”.

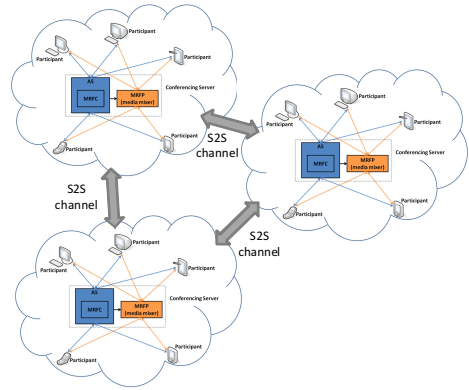
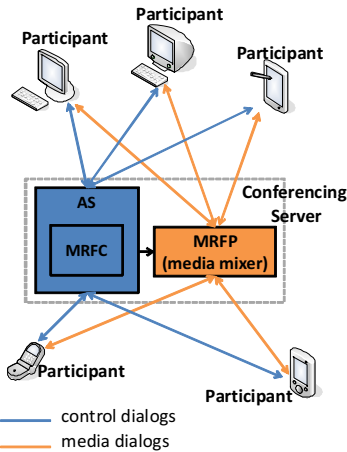
On the basis of the adopted conference blueprint, the AS issues commands to the media mixer, which is the system entity in charge of performing the audio and video mixing functions needed to deliver to each conference user the proper mixed stream, according to her/his preferences as well as to the supported capabilities of the device she/he uses. The multimedia streams generated and/or consumed by each participant are in fact always exchanged with the media mixer, hence realizing a star topology also on the media plane.

The AS and the media mixer can be logically grouped into an integrated server-side logical entity, called “Conferencing Server”, making available all the functions needed to provide the conferencing service, both on the control plane and on the media plane.

## 2.2 Distributed Conferencing Scenario

We herein move the attention to the distributed scenario, in which several centralized conferencing clouds are interconnected and cooperate in order to provide the conferencing service in a distributed manner (Figure 2).

Under the assumption that all conference participants refer to the focus in the home network of the conference initiator, which we call the “main” focus, the IMS centralized conferencing solution keeps on working also in the scenario where the conference participants belong to networks owned by different telecom operators. Though, in a fully distributed scenario, the above case should be dealt with by providing each involved network with a specific focus managing the associated local users who subscribed to the conference. It is up to the main focus in the conference initiator’s home network to provide all other focus entities with up-to-date conference information. Such foreign focus entities thus play a twofold role. On one hand, they act as a regular conference focus for the participants



**Fig. 1.** A view of the conferencing system **Fig. 2.** Distributed conferencing scenario

belonging to their underlying managed network; on the other hand, they appear as normal participants to the focus in the conference initiator’s home network.

Similarly, on the media plane, multiplexing/demultiplexing of multimedia streams generated by local users is up to each local media mixer. The resulting stream coming from foreign media mixers to the main media mixer (i.e., the mixer located in the island where the distributed conference has been created) is handled in the same way as the media flow of an ordinary conferencing client. In order to achieve consistency in a conference involving more islands, a dedicated communication channel exists between each pair of focus entities participating in the conference (“Server to Server (S2S) Channel” in Figure 2). Such channel is used to exchange conference information, as well as to manage synchronization issues.

### 3 Motivation and Related Work

This work is willing to provide a modeling approach to accomplish performance evaluation of actual IMS-compliant conferencing platforms in order to support their design and deployment phases. With this goal in mind, we start considering the IMS-compliant conferencing framework described in Section 2. Such a system is a very good example of the effort aimed at providing a contribution to the solution of the above mentioned IMS-related open issues, with special regard to the need for actual implementations of IMS architectures and services.

The considered system has been already the subject of a thorough experimental benchmarking campaign carried out on a real testbed [6,7] aimed at conducting a performance and scalability analysis of both the centralized and the distributed architectures. The experimental results showed that, in a controlled network scenario, as the number of users increases, CPU utilization of the centralized server becomes the most critical performance index, since such

server is responsible for the management of both the signaling dialogs with each conferencing client and the media streams mixing function. In particular, managing the media plane turned out to be much more onerous than controlling the signalling plane. This led the developers to focus on the performance of the media plane, which definitely represents the limiting factor as far as scalability is concerned.

The mentioned experimentations required a lot of efforts. For each trial, a real laboratory testbed has been properly set up and a realistic set of conferencing sessions has been reproduced over it. Each such session was actually instantiated by replaying a proper set of recorded traces associated with actual conferences and hence mimicking in a reliable fashion the actual behavior of end-users and servers. This approach does not clearly solve the issue of assessing as timely as possible (i.e., during the design and capacity planning phases, before the actual deployment of the numerous infrastructure components) the main non-functional features of a conference, like performance and dependability. This clearly calls for a need to make available a set of flexible tools for the assessment of the aforementioned requirements, which naturally lend themselves to a multidimensional, multi-faceted characterization.

A common approach to performance evaluation of networked systems which does not require real testbed implementations is of course based on the usage of network simulators. Nevertheless the current leading network simulators (such as ns-2, ns-3, OPNET Modeler, etc.) do not offer complete support for IMS [8]; in particular, they do not support the modeling of the media plane through embedded components, but usually rely on third-party implementations of some of the required IMS media plane functions. Compared with the experimental and the simulation approaches, formal modeling presents several advantages. It allows for performance evaluation and performance prediction thus supporting the early phases of the development of the system and providing the possibility to easily evaluate several conference blueprints. This also allows for sensitivity and stability analysis on a number of parameters, including those associated with Quality of Service, which gives the service providers an effective means to plan and schedule conferences according with the expected QoS levels. A further advantage in using formal modeling is the possibility to model and analyze anomalous behaviors due to attacks, faults, overloads or degraded operating modes. In this paper we mainly address the aspects related to the scalability and performance of the system. Several works can be found in the literature aiming at mainly analyzing access control and protocols by developing formal models, but focusing on security [9], protection [10,11], dependability [12] and protocol validation [13]. To the best of our knowledge there are not many attempts at modeling the IMS-compliant conferencing frameworks specifically oriented towards the study of the media plane.

Multimedia conferencing systems have already been a playground for the application of formal modeling. Several works focused on the orchestration of the different building-blocks a multimedia conferencing platform relies on, as it is the case in [14]. In the cited work, the authors mainly use Petri Nets and related

modeling languages to perform composition correctness validation. Other researchers leverage Petri nets to model conferencing media flows characteristics, as well as time synchronization mechanisms. These approaches are definitely more relevant to our work, since we are concerned with media plane modeling rather than compositional aspects. In [15] authors use TPNs to properly describe the temporal scheduling of the multimedia presentation process, within the inter- and intra-streams synchronization constraints. TPN models of such mechanism are used to support the implementation of a system capable to control streams synchronization in a video conferencing application. As opposed to the mentioned work, we do not analyze media streams management to drive the construction of a system from scratch, but rather to model the workload of an existing conferencing media mixer and study its performance.

Since our goal is to provide a flexible tool for the evaluation of performance and scalability, we use SAN models to describe the conferencing server behavior, while dealing with users mixing preferences and coping with the diversity of the involved media and related codecs. SANs are more suitable than TPNs to our aims because of their modeling power and efficiency. Specifically, they provide the basic modeling mechanisms to easily integrate, in the models, data structures representing messages, as well as to replicate and compose submodels.

As anticipated, the focus of this paper is on both scalability and performance aspects of the system. The analysis is carried out by alternately varying: a) the scenario (centralized vs distributed); b) the number of participants; c) the kind of deployed conference (e.g., presence/absence of the moderator, presence/absence of transcoding mechanisms). From now on, for the sake of simplicity, we consider conferences with only audio streams (audio conferences): this simplification allows us to better explain our approach without losing in generality.

## 4 Model Development Process

In this Section we present the development process adopted to build the performance model of the conferencing system described in Section 2. The modeling approach is founded on well known principles: it is compositional and hierarchical, and promotes model reuse. In particular we refer to the OsMoSys modeling methodology [2] which was born to support compositional and multiformalism modeling. Here we do not exploit multiformalism but we rather take advantage from the separation between interface and implementation of submodels. The IMS-conferencing case study requires that a clear separation of concerns is adopted in modeling the behavior of the different entities of the framework. Moreover, the heterogeneity of devices, the variability of the number of participants and the need for scalability suggest that proper solutions must be found to provide the modeling approach with the necessary flexibility. Possible answers to these issues are proposed:

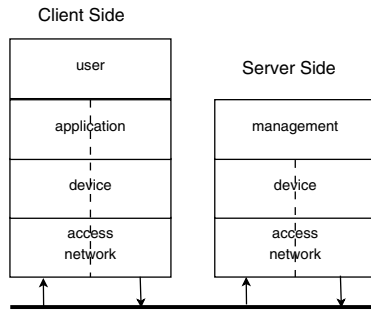
1. separation of concerns: the first step of the model development process is to define the behavioral levels of the system. They will be modeled in isolation and then composed through model interfaces.



2. enabling model reuse: we propose to extend the OsMoSys methodology by introducing the concept of *Model Template* in order to easily define a family of models sharing part of their definition and allowing for parametric specification of some elements of the models themselves.
3. dealing with model scalability: the simulation time and space occupation needed to solve the models are very high as the number of participants increases: we propose two solutions to this issue, described in Section 6.

#### 4.1 Separation of Concerns: Modeling Levels

In Figure 3 the structure of both the Participant (Client Side) and the Conferencing Server (Server Side) entities is shown. Since we deal with performance and scalability analysis of the system, we concentrate the modeling efforts on the media plane: in fact, as remarked in Section 3, experimental campaigns showed that operations like transcoding and mixing of the participants' media streams are the most demanding ones in terms of required resources and hence have the strongest impact on the overall system performance.



**Fig. 3.** Reference modeling schema

Client Side is composed by the following levels. **User:** at this level the media streams are produced (output streams) or consumed (input streams). The user is characterized by the role she/he plays in the conference (moderator, speaker, observer, etc.) and by the load that she/he produces. **Application:** at this level the streams are encoded/decoded. For example, the application samples and encodes the audio signal coming from the user's microphone. The production rate of the messages depends on the type of encoding (i.e., codecs adopted), as well as on both the power and the number of CPUs. **Device:** this level is in charge of bundling output streams into packets or viceversa (i.e., assembling input streams). At this level the features of the participant's communication device (the User Equipment) are taken into account. The main parameters are the power and the number of CPUs. **Access network:** it represents the access network of the device and is characterized by its connection speed.

Server Side levels are the following ones. **Management:** it includes the server-side functionality for the provisioning of the conferencing service on the media

plane in each centralized conferencing cloud. More precisely, as already mentioned, we skip the modeling of the Application Server component, while stressing the details of the media mixer component in charge to perform media streams management on the basis of the particular kind of conference. **Device** and **Access network** levels are the same as the client-side ones.

## 4.2 Enabling Model Reuse: Components and Templates

The modeling approach we adopt to model the conferencing framework is based on the OsMoSys (Object-baSeD multi-formalism MOdeling of SYStems) modeling methodology [2]. The vision underlying OsMoSys is to apply the concepts of component software engineering to the development of formal models in order to provide a practical support to model engineering. According to OsMoSys a model is an object of a Model Class which encapsulates the details of its implementation (the model structure is expressed by a formal language). Hence, models are components which communicate via interfaces. An interface is a subset of elements of the model structure which may be used to exchange information among models (for example, the value of parameters, indices or the overall state of the model). The role of the interfaces has already been addressed in [16,17].

The OsMoSys methodology may be very effective in modeling the conferencing system since a component-based approach emphasizes the separation of concerns. Nevertheless, the problem we deal with is characterized by a variable number of peers (Conferencing Servers and/or Participants), as well as by the heterogeneity of the involved applications and devices. Hence, we need mechanisms to easily provide a specification for generating models based on parameters. With this aim in mind, in this paper we explore the possibility of extending OsMoSys by defining *Model Templates*.

*Model Templates* may introduce a powerful feature in formal modeling, since they allow to specify with a single model description an entire family of related models, called template models. Similarly to Class Templates introduced by several programming languages, *Model Templates* require one or more type parameters to specify how to generate a specific instance from a generic model template. The type parameters refer to the type of one or more elements of the model structure, including the type (i.e., the Model Class) of submodels. It is also possible to use *non-type* parameters, e.g., to specify the number of replicas of a subset of elements (including submodels). The complete definition of *Model Templates* in OsMoSys is part of an ongoing work. In this paper we define and exploit a specific case of *non-type* parameters. The formal definition and application of *type* parameters are out of the scope of this work.

Some research papers have been published proposing template based approaches, all limited to some extent: in [18] typed parameters template are introduced for Petri Nets models. In [19] the focus shifts onto Stochastic Activity Networks and on the possibility to change the behavior of the model according to *non-type* parameter values. Other works focus on the possibility to both replicate and join submodels inside a compositional approach [20,21]. To the best of our knowledge, there are no works aimed at defining a generic modeling approach in

which formal models are templates developed in terms of *type* and/or *non-type* parameters that can be specified at instantiation time.

In order to define the *Model Templates* according to the OsMoSys notation, some preliminary definitions are due. Formally, a *Model Class*  $MC_{\mathcal{F}}$  is a triplet  $(T, S, SM)$  where:  $T$  is the class name,  $S$  is the graph structure of the class,  $SM$  is a set of submodels.  $MC_{\mathcal{F}}$  is compliant with a formalism  $\mathcal{F}$  and its structure  $S$  is a graph of elements of  $\mathcal{F}$ . Specifically, from now on we will call  $N$  the set of nodes and  $E$  the set of edges of  $S$ . For example, if a *Model Class* is compliant with the Petri Nets formalism, its structure will comprise *Place*, *Transition* (the nodes of the graph) and *Arc* (its edges). It holds:  $S = External_S \cup Internal_S$ ;  $External_S \cap Internal_S = \emptyset$  where  $External_S$  is the subset of the *interface* elements of the *Model Class* and  $Internal_S$  is the subset of elements that are encapsulated by the class.

Let us define a *Model Template*  $MT_{\mathcal{F}}$  as a pair  $(MC_{\mathcal{F}}, PAR)$  where  $MC_{\mathcal{F}}$  is a *Model Class* and  $PAR$  is a set of *non-type* parameters:

$$PAR = \{p_1, p_2, \dots, p_n\}, n \geq 1.$$

**Definition 1.** A *non-type parameter* is a triplet  $p_i = (l_i, SS_i, f_i)$ , where  $l_i$  is the name of the parameter;  $SS_i$  is a subgraph of  $S$ ,  $SS_i = (NN_i, EE_i)$  with  $NN_i \subseteq N$  and  $EE_i = \{e = (m, m_1) \in E | m, m_1 \in NN_i\}$ .

A synthetic notation for the Model Template is  $MT_{\mathcal{F}} < l_1, l_2, \dots, l_n >$ .

**Definition 2.** An *instancing function* is  $f_i : \mathbb{N} \rightarrow \mathcal{M}_{\mathcal{F}}^1$  where  $\mathcal{M}_{\mathcal{F}}^1$  is the set of the *Model Classes* compliant with the formalism  $\mathcal{F}$ .

An *instancing function*  $f_i$  must specify how the Model Template should be (automatically) instantiated by using the *non-type* parameters. A synthetic notation for an instance is  $MT_{\mathcal{F}} < v_1, v_2, \dots, v_n >$  where  $v_i$  is the value on which  $f_i$  is computed.

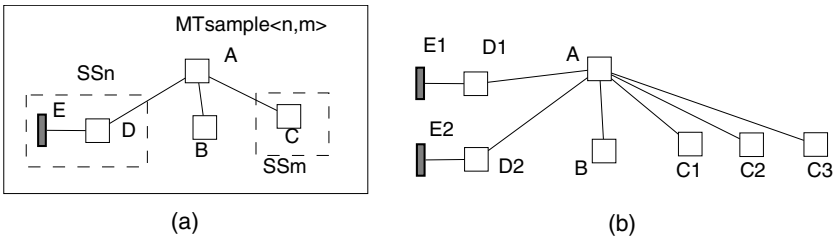


Fig. 4. A Model Template example

An example is depicted in Figure 4, in which a simple case of replication is shown. A Model Template is on the left (Figure 4(a)), let it be:

$$MT_{sample} = (MC_{sample}, \{p_n, p_m\})$$

where:  $p_n = (n, SS_n, f_n)$  and  $p_m = (m, SS_m, f_m)$  are the *non-type* parameters. The subsets of the model structure in the dashed boxes ( $SS_n$  and  $SS_m$ ) may be replicated and a template model may be generated by providing the value of the two non-type parameters. Both subnets include a submodel (the white square). The Model Template is denoted by  $MT_{sample} < n, m >$  while the template model is denoted by  $MT_{sample} < 2, 3 >$  and is shown in Figure 4(b). The model is obtained by specifying the values 2 and 3 for the number of replicas of  $SS_n$  and  $SS_m$ , respectively<sup>1</sup>. In this example we use a simple instancing function that copies a sub-graph as many times as indicated in the parameter value: all the replicas are then connected to the rest of the model by replicating arcs connecting the sub-graph with the rest of the structure, too. In the example, the subgraph  $SS_n$  and the arc from A to D are replicated twice. Of course, a renaming function is also needed to avoid conflicting names.

The *instancing functions* must be provided in order to state how the template models have to be generated. In the example a simple instancing function is used which allows for replication of a part (or the whole) of the model structure. It is defined by induction as follows.

We denote as  $MT_{\mathcal{F}} < x >$  a Model Template with a parameter  $(x, SS_x, f)$  where  $SS_x = (NN_x, EE_x)$ . Let  $\overline{EE}_x$  be the subset of  $EE_x$  connecting nodes of  $NN_x$  and of  $N - NN_x$ .

- $S_1 = S$ : the function does not change anything in the structure of the Model Class;
- $\forall v \geq 2$ , let  $f(v-1) = (T_{v-1}, SS_{v-1}, SM_{v-1})$  and  $f(v) = (T_v, SS_v, SM_v)$ :
  1.  $T_v$  is calculated by a renaming function;
  2.  $NN_v = NN_{v-1} \cup NN_x$  and  $EE_v = EE_{v-1} \cup EE_x \cup \overline{EE}_x$ , i.e., the graph generated at  $v$  is built by joining the one generated at  $v-1$  with a new replica of  $SS_x$  and the arcs connecting the new replica with the rest of the structure (not included in  $SS_x$ );
  3.  $SM_v = SM$

Notice that it might hold that  $SS_x = S$ , thus allowing for the replication of an entire model.

In the next Section, the modeling approach herein described is exploited to develop a library for the implementation of the models associated with an IMS-compliant conferencing framework. Such library contains both Model Classes and Model Templates, compliant with the SAN formalism, that must be customized and instantiated according to the different available blueprints.

## 5 Modeling the Conferencing Framework

This Section details how the modeling approach described in Section 4 is applied to generate the SAN models of the conferencing system. Models are described

<sup>1</sup> Note that  $E$  is an interface element: this approach can be used to replicate model interfaces, too.

according to a bottom-up approach: first, we introduce the models realizing each level of the conferencing system in isolation (Figure 3); then, we show how they are composed in order to obtain the models of both the Client Side and the Server Side. Some of the models described in this Section are template models, since they are obtained by instantiating specific Model Templates (specifically, User, Management, Client Side and Server Side). The subnets we will introduce to perform the composition in Subsection 5.6 are template models, too. The remaining models (Application, Device and Access Network) are not templates. For the sake of conciseness, only the formalization of the User Model Template, according to the notation presented in Subsection 4.2, is reported in the following.

### 5.1 User Level

In the conferencing context a user has the same behavior independently from the specific medium she/he uses (audio, video, etc). This behavior may be modeled by a state machine: a User Model Template that produces and consumes a generic multimedia stream  $User_{SAN} = (UserMC_{SAN}, \{(K, S, f_K)\})$  is shown in Figure 5, where  $UserMC_{SAN} = (userMedia, S, \emptyset)$ : the SM set of  $UserMC_{SAN}$  is empty and  $f_K$  is the instancing function described in the previous Section. User has one parameter  $K$  that represents the number of the media streams related to the user. Note that  $SS_K = S$ , i.e., this is the trivial case in which the entire structure of  $UserMC_{SAN}$  must be replicated. Hence,  $User_{SAN} < 1 >$  coincides with the Model Class  $UserMC_{SAN}$ . In order to show the flexibility of this mechanism, Figure 6 shows the template model  $User_{SAN} < 2 >$  modeling a user who makes use of both video and audio.

The interface of this model is represented by the two places *MediaInbound* and *MediaOutbound*, respectively for inbound and outbound traffic. The user may be in a *Producing* or in a *Waiting* state (i.e., “talking” and “not talking” for the audio media) represented by ordinary places. The timed transitions ( $P2W$  and  $W2P$ ) model the switch between the two states. When the transition  $P2W$  fires, a token is removed from the *MediaOutbound* place by the output gate  $OG0$ . On the contrary, when the transition  $W2P$  fires, a token is added to the place *MediaOutbound* by the gate  $OG1$ .

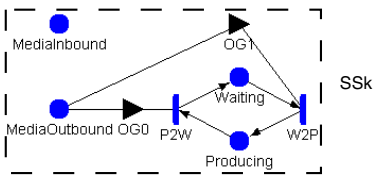


Fig. 5. The User Model Template

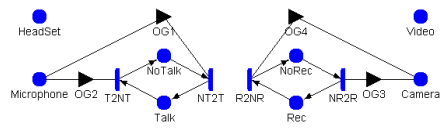


Fig. 6. The audio conference user model

### 5.2 Application Level

At this level, we model the behavior of the encoding and decoding processes, on both Client and Server Sides. For each medium associated with the Participant, two different models are created, one for each stream direction: inbound-decode and outbound-encode. Therefore, we have a number of instances of the inbound and outbound Application model depending on the media enjoyed by the participant. The two patterns of inbound and outbound application models are depicted in Figure 7 and Figure 8, respectively. The former model is able to maintain a token in the *output* place starting from a set of structured tokens (messages) arriving at the *app\_in* place. The latter is instead devoted to the production of structured tokens to be put into the *app\_out* place, with the presence of an incoming media flow being represented by a token in the *input* place. In detail, the interfaces of the inbound application model are represented by the places *output* and *app\_in*: a structured token in *app\_in* indicates the presence of a message ready to be decoded; after the decoding process (that keeps the CPU busy), the decoded data are ready to be played out. On the left of the image there is a watchdog that is responsible for maintaining the token in the *output* place: two places *playing* and *noPlaying* with the connected transitions and gates implement the state machine related to the watchdog.

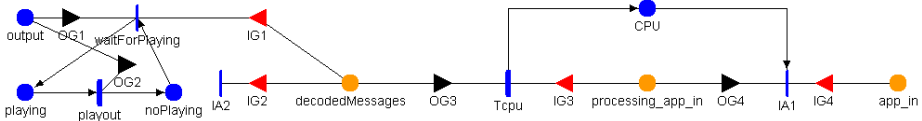


Fig. 7. The inbound Application model

A specular behavior is manifested by the outbound application model. A token in the *input* place indicates the presence of an external flow that needs to be encoded. In this case, *IG1* enables the *sampling* transition that fires after a sample length and produces a new sample in the *samples* place. Upon arrival of a specific number of tokens in the *samples* place, depending on the encoding standard used, *IA1* is enabled and, if the CPU is not busy, the generation of a message is activated. After a predefined processing time, *Tcpu* fires and the message is made available at the output interface *app\_out*.

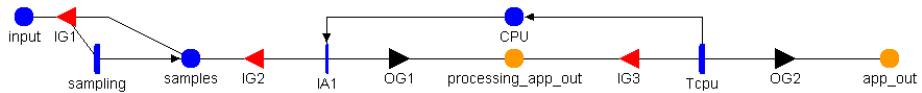


Fig. 8. The outbound Application model

### 5.3 Device Level

The device layer is responsible for the decomposition of all output streams produced by a Participant, encoded with application messages, into one or more packets, or, viceversa, for the assembly of input packets into application messages. All the application level media streams are conveyed in an outbound Device model that decomposes each message into a number of packets, according to the length of the message itself, assigning to each of them an incremental identifier. Therefore, each packet is characterized by the message number and packet number that allow receivers to assemble the packets into application messages. Figure 9 depicts the SAN model of the device on both Client Side and Server Side: this model can be connected either to a network layer model or to an application layer working in both inbound and outbound configurations. The model interface resides in the *dev\_in* (input) and the *dev\_out* (output) places. In detail, the two input gates, *IG1* and *IG2*, verify either the presence of a message (in the output configuration) or the arrival of the last frame of a message (in the input configuration) on the *dev\_in* place. These two gates are able to recognize the direction in which the model has been assembled thanks to the packet number: if this value is not specified (or it is set to a meaningless conventional value, like, e.g., 0) it means that the model is working in outbound configuration, inbound otherwise. When an application message is (packets are) ready to be decomposed (assembled), the User Equipment processes it (them) until the related packets are (application message is) put into the output interface.

### 5.4 Access Network Level

The network layer is responsible for getting packets from a source peer (Conferencing Server or Participant) and transferring them to the backbone network, when in outbound configuration, or viceversa, when in inbound configuration. The transfer time depends on the nature of the communication medium (Wireless, Wired, DSL, UMTS, etc.) to which the User Equipment is connected. Figure 10 depicts the SAN model of the access network: as for the device model, it is generic in the direction of communication (inbound/outbound) and the transfer time actually represents a parameter of the model. The interfaces of this model reside into *net\_in* (input) and *net\_out* (output) places, respectively.

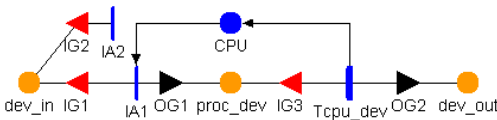


Fig. 9. The Device model



Fig. 10. The Access Network model

## 5.5 Management Level

The Management level is specific to the Conferencing Server and it is not present in the model on the Client Side. This level is responsible for transcoding between the different media codecs adopted by participants, as well as for forwarding messages to them, by relying on stored information about the conference (that the Conferencing Server needs to keep in memory). The overall model of this level instantiates the aforementioned application model templates (inbound and outbound) to transcode messages, in conjunction with another model, described in this paragraph, that deals with the generation and transmission of messages. The overall Management model, like the application models, is specific to the particular media and can hence be instantiated ST times (ST being the number of media streams involved in the conference). Figure 11 depicts the SAN representation of this additional model.

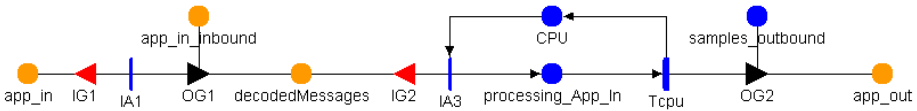


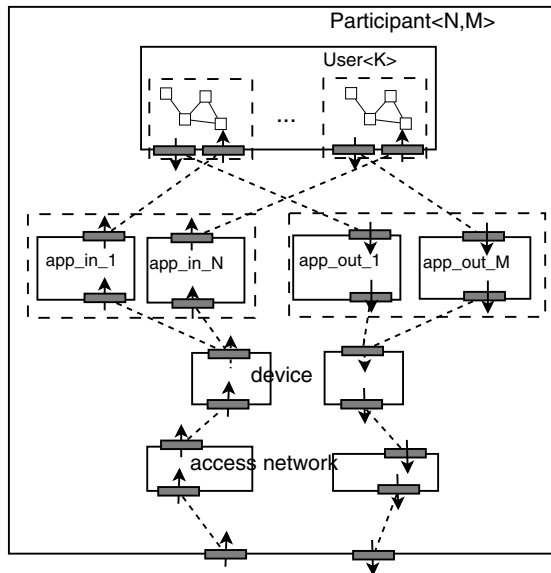
Fig. 11. The Server Side Management additional model

Given its function, the additional model offers two interfaces to the device level (*app\_in* and *app\_out*), as well as four connection points with the inbound application model (*app\_in\_inbound* and *decodedMessages*) and with the outbound application model (*samples\_outbound* and *app\_out*), both used for potential messages transcoding. When a new message arrives at the server application level on the interface *app\_in*, *IG1*, *IA1*, *OG1* check whether the codec used in the message is the standard codec chosen for the conference. In this case the message is put into the *decodedMessages* place; otherwise, it may be transferred to the inbound application model through the *app\_in\_inbound* place. The transcoded messages produced by the inbound application model are put into the *decodedMessages* place, where the CPU finds only messages encoded with the standard codec, which are elaborated and sent to all participants through *OG2*. Before putting messages into the *app\_out* place, the server checks the codec of choice for each participant and, if not consistent with the standard one, it puts the samples into the *samples\_outbound* place (connected to the outbound application model) for the correct message encoding.

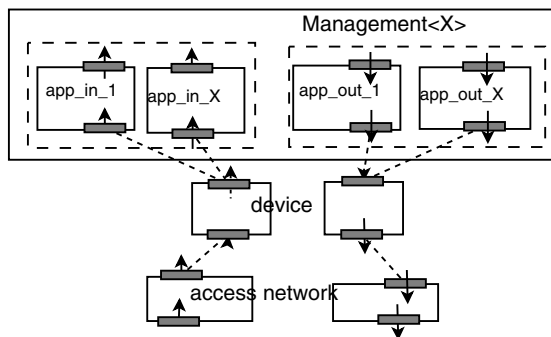
## 5.6 Composing Model Stacks

The aim of this Subsection is twofold: it first shows how composition has been applied and then how it has been implemented in the Möbius framework [22]. A two-level composition strategy has been implemented: an “intra-stack” composition to create Participant and Conferencing Server composed models, and





**Fig. 12.** Client Side stack Model Template



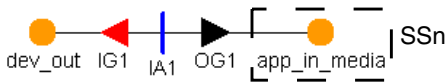
**Fig. 13.** Server Side stack Model Template

an “inter-stack” composition at a higher level to create the overall conferencing system model. The “intra-stack” compositions representing the Client Side and the Server Side model stacks are shown in Figure 12 and Figure 13, respectively.

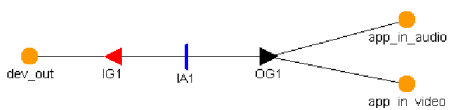
Starting from the top of the Figure 12, the Model Template described in Subsection 5.1 is used inside the User level according to the parameter  $K$  (the number of involved media). At the Application level, several models described in Subsection 5.2 are used. The number of inbound and outbound application model replicas is determined from  $N$  and  $M$ , representing, respectively, the number of received and transmitted media streams. The two values of  $N$  and  $M$  can be different (and not equal to  $K$ ) in those cases when a client is producing but not receiving a specific medium (like, e.g., for a mobile phone, not equipped

with a camera, which is not capable to generate its own video stream, but can nonetheless receive video sent by other participants). In the simplest case of an audio conference where all participants can both listen and speak we have  $K = M = N = 1$ . Both at device and at access network levels two models, according to the direction of the media flows, are instantiated. It is clear that the whole Participant model is itself a Model Template since entire submodels can be replicated. Figure 13 depicts the “intra-stack” composition at Server Side where the different part is represented by the Management model (since Server Side and Client Side use the same Device and Access Network models). The model is in charge of representing the Conferencing Server’s audio and video mixing functions: in order to capture such concepts, the model is built by using hierarchical composition. In fact, transcoding media streams may be necessary depending on the adopted encoding-decoding formats, as well as on the capabilities of the involved devices. Encoding and decoding are modeled by using the application model templates (respectively *app\_out* and *app\_in*) described in the previous paragraph. These are model templates in X parameter, that is the number of the different encoders and decoders used in the conference.

The “intra-stack” composition is performed by place superpositions, where it is naturally possible, or by using some interconnection SANs: Figure 14 shows the Model Template of the interconnection network between device and application models on the inbound branch of both Participant Side and Server Side. Figure 15 shows the real instantiation obtained by replicating, for two media (audio and video), the  $SS_n$  subgraph. The latter model allows to connect the device level of a Participant/Conferencing Server to the two inbound application models. It basically represents a simple messages separator, working in accordance with the media to which they refer.



**Fig. 14.** The Device-to-Application composition network Model Template



**Fig. 15.** A Device-to-Application composition network model

The “inter-stack” composition, starting from the definition of conference deployment (in terms of the number of conference participants, as well as overall Conferencing Server configuration), joins all the instances of the peers stack model (Participants and Conferencing Server) through a proper network, on the basis of a provided topology.

The two composition steps described above are implemented into the Möbius framework by means of the Rep and Join operators [22]. The set of participants is created by replication and instantiation of the entire Client Side model stack. Figure 16 shows an example of an overall composed model of a centralized audio conferencing system. The “intra-stack” composition has been implemented through the *participant* and *server* Join models; similarly, the “inter-stack”

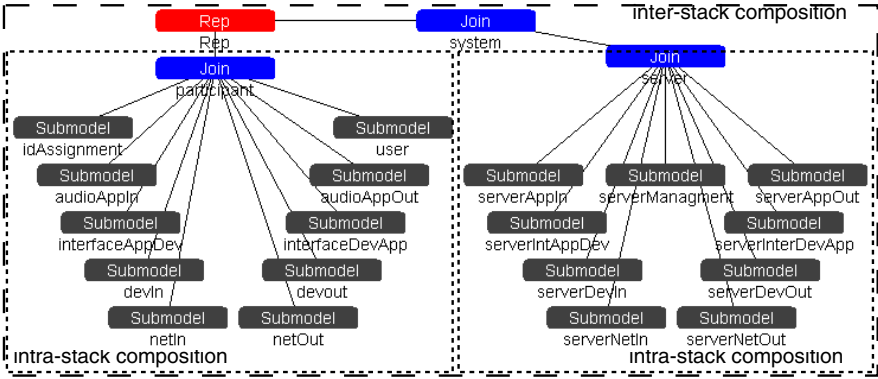


Fig. 16. The overall composed system model

composition is given by the combined action of both the *Rep* (to replicate participants) and the *system* Join model (acting as a top level joining all the involved entities). To identify single participants, by assigning a specific id to each of them, a simple *idAssignment* model has been implemented. This model is similar to the one proposed in [21] and allows to associate an index to each participant instance. Indeed, having non-anonymous participants is critical when, for example, a measure on a specific instance is requested or a different behavior must be modeled for it. The composition SAN is in this case trivial, since different models have to share some communication places.

## 6 Dealing with Model Scalability: Stub and Reduced Models

The composed model of a complete conferencing system, shown at the end of the previous section, requires increasing simulation time and memory space during the solving phase and thus call for the introduction of scalable solutions. These problems are due to the huge use of extended places containing data structures arrays that need a high start time to be solved. A possible solution relies on storing data in a database, while keeping in memory just the references to the tuples. However, this strategy is not effective enough for our scope because it just helps overcome the memory consumption issue, while leaving unaffected the computation time. Therefore, we herein propose two different approaches to face scalability issues: (i) stub models, that replace some levels of peer stack with simplified models, and (ii) reduced models, that take the place of the entire peer stack. These two techniques are not mutually exclusive and can be jointly used to model the system. They introduce different approximation levels that we have analyzed by means of simulations. We show the obtained results in the following section.

## 6.1 Stub Model

A stub model, as in software development techniques, may simulate the behavior of an existing, more detailed, model or be a temporary substitute for a yet to be developed model, hence reducing the complexity of a complete one. The stub models can be used to perform analysis at a precise level among those previously identified, by simulating the behavior of the remaining ones. The use of stubs is simplified by the layered nature of system and models, as well as by the compositional nature of the modeling methodology: clearly, a stub for a generic model layer must exhibit the same interface of the models it replaces. Stub models, like the complete one, do not prevent from identifying the single participant and assigning it a different behavior. They also reduce memory occupation for the complete stack; though, as it will be shown in the following, they still entail a significant computational complexity. The approximation introduced by a stub model is proportional to its ability to represent the missing levels: the greater the accuracy of the model, the lower the error on results. The adoption of stubs can be useful for two different purposes: i) to focus on the model of a specific level, in which case a stub acts as a substitute for the models of the lower levels; ii) to build a smaller scale model of the overall conferencing system, by considering just one or few levels both for participants and for the server.

## 6.2 Reduced Model

In many cases the analysis of a conferencing system is conducted to calibrate or to estimate the performance features of the conferencing server: in these cases we can ignore the identification of each single participant, hence allowing for more participant stacks to be collapsed into a *reduced* model. The *reduced* model is here presented: it can be connected directly to the Conferencing Server model stack and it generates the load related to  $n$  participants without instantiating all of them. This model is depicted in Figure 17: (i) the distribution function of the *production* transition models the average rate with which messages are produced by a single participant; (ii) the *activeClients* place is used to store, at each instant of time, the number of transmitting participants; (iii) the *verifyClients* input gate checks the presence of at least one token in the associated place and is used to either increase or decrease the number of active participants, for example in case of configurations involving moderation; (iv) the *generateMessages* output gate injects packets into the network. This simple model introduces an acceptable approximation in the evaluation of server-side performance, with reasonable complexity, as well as reduced memory occupation, as demonstrated in the next Section. Though, it does not allow to conduct performance evaluations at the client side.

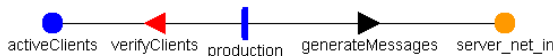


Fig. 17. The client reduced model

## 7 Evaluation of the Proposed Approach

This Section assesses the proposed approach by means of experimental campaigns. Our objectives are: i) validation of the proposed models through a comparison with real data; ii) assessment of the scalability of the modeling methodology; iii) analysis of its power and flexibility when addressing different conference configurations (e.g., with/without transcoding, with/without moderation, etc.).

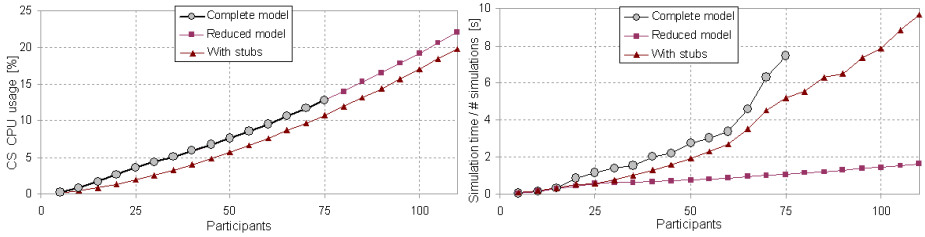
To this purpose and according to related works [6,7] we choose CPU usage of the Conferencing Server (for an audio conference) as a key performance indicator. Such measure is usually computed with respect to the number of participants connected to the conferencing system. Before showing the results of our analysis, a description of the system configuration parameters used in the trials must be given (Table 1). For the sake of coherence, such parameters have been chosen according to the above cited works. The first two objectives are fulfilled in Subsection 7.1 and the third one in Subsection 7.2

### 7.1 Validation with Real Data

This group of simulations aims at demonstrating the equivalence between our models and real experimental data. In order to achieve this goal, we compared the estimated CPU usage obtained by solving models with the values reported in [6,7]. This validation requires a high level of scalability in terms of number of

Table 1. System parameters

Parameter	Description	Value
pkts_per_msg	Packets generated at device level for each application level message.	1
sample_len	Duration of a single audio sample (according to G.711 specification).	0.125 ms
samples_per_msg	Samples contained in an application level message (according to G.711 specification).	160
sample_dim	Dimension of an audio sample	8 bit
app_in_Tcpu_mean	Mean CPU time needed to decode/encode a message at the application level.	1.3E-4 ms
app_out_Tcpu_mean	Mean CPU time needed to decode/encode a message at application level.	1.3E-5 ms
app_in_Tcpu_var	Variance of CPU time needed to decode/encode a message at application level.	1.3E-5 ms
app_out_Tcpu_var	Variance of CPU time needed to decode/encode a message at application level.	1.3E-5 ms
mngmt_Tcpu_mean	Mean of CPU time spent by the server to generate a single application message	1.3E-4 ms/msg
mngmt_Tcpu_var	Variance of CPU time spent by the server to generate a single application message	1.3E-5 ms/msg
dev_Tcpu_mean	Mean CPU time to decompose/recompose a message at device level.	0.6E-6 ms
dev_Tcpu_var	Variance of CPU time to decompose/recompose a message at device level.	0.6E-7 ms
net_band	Upstream/downstream bandwidth needed for a message	67 Kbit/s



**Fig. 18.** Scalability analysis: CPU usage

**Fig. 19.** Scalability analysis: simulation time

participants: for this reason, we introduce at first a comparative analysis of the complete model with *stub* and *reduction* approaches to evaluate their scalability features.

*Complete vs stub vs reduced model.* Based on the specific well-defined reference scenario, the three models are compared: a complete model obtained by composing the server stack with the replicas of the participant stack; a model obtained by substitution of the *device* and *access network* levels with stub SAN models; a model built using reduction techniques as defined in Section 6. The accuracy of the *stub* and *reduced* models compared to the complete one, that is the estimated CPU usage of the Conferencing Server against the number of connected participants, is reported in Figure 18. The simulation times associated with these studies are instead reported in Figure 19. The two analyses have been conducted for a number of participants between 5 and 110: a quick look at the graphs tells us that the reduced model is able to analyze up to 110 participants within reasonable simulation time. The complete model is instead capable to arrive at a maximum of 75 participants on the computer used for the simulation (Intel(R) Core(TM) 2 Duo @ 2.53GHz, 2 GB RAM) due to RAM saturation during the solving phase. The reduced model scales very well in simulation time and, according to the first diagram, it is also as accurate as the complete model for the considered conference scenario. The drawback in its use resides in the potentially unsatisfactory level of detail: some analyses that require non-anonymous participants (e.g., the QoS of the conference as perceived by end-user), cannot be conducted because of its limited description of the participants' model stacks. On the contrary, a stub solution seems to best strike the balance between accuracy, simulation time and level of detail. In particular, the considered stub models superpose communication places without introducing CPU effort at device and network levels: the accuracy can be further improved by constructing more realistic (yet less performing) stub networks.

*Reduced model vs real data.* We compare the solutions obtained with the reduced model with the real experimental data for the two different configurations described in Section 2: a) centralized scenario, where a single focus acts as the Conferencing Server for a number of participants varying from 25 and 300; b) distributed scenario, where a “two islands” configuration is used for load balancing purposes (150 participants per focus). The results are reported in Figure 20

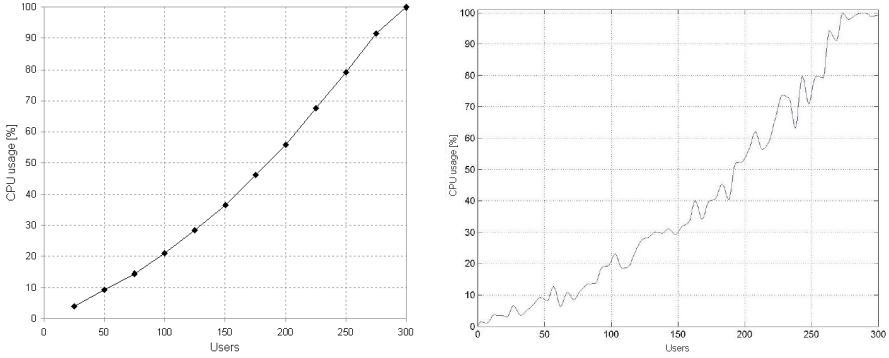


Fig. 20. Reduced model vs real data: centralized scenario

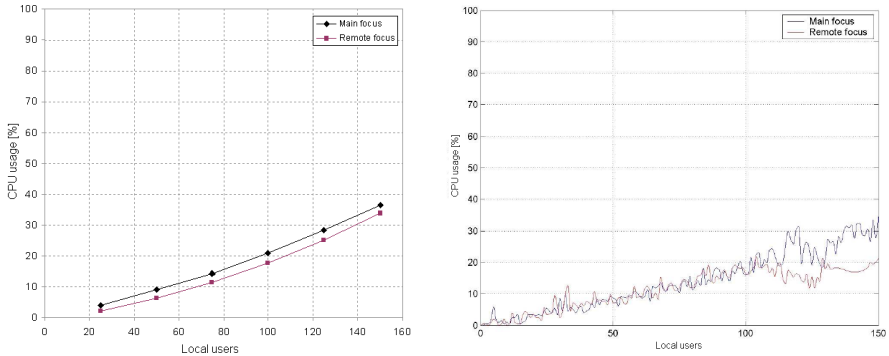
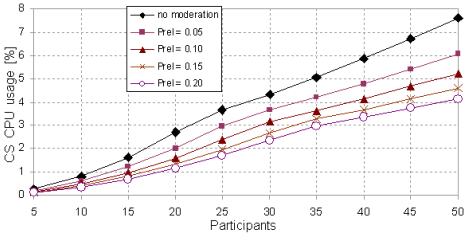


Fig. 21. Reduced model vs real data: distributed scenario

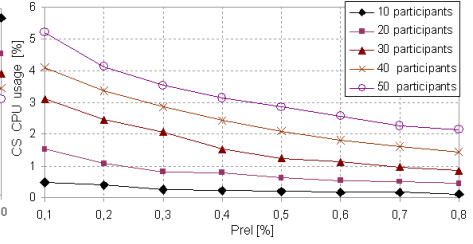
and Figure 21, respectively. In both scenarios the reduced model fits well the real measured data.

### 7.2 Evaluating Power and Flexibility

So far we have validated our approach both in terms of accuracy of the results and scalability. The next two experiments point out the power and flexibility of the proposed modeling approach by evaluating the system’s performance under different conference configurations. More precisely, we evaluate the effects of moderation and transcoding on the CPU usage of the central Conferencing Server for an audio conference. The aforementioned experiments are conducted by varying two key simulation parameters: the probability of being muted by the moderator and the probability of a flow to be transcoded. For both of them, we employed complete models, limiting to a value of 50 the number of participants. In the performed simulations, all participants use identical devices and present the same probability to talk and probability to have their flows be transcoded. However, the adoption of a complete stack model allows us to assign different



**Fig. 22.** CPU usage vs number of users for different Prel values

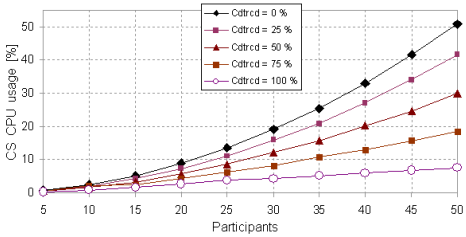


**Fig. 23.** CPU usage vs Prel for different conference sizes

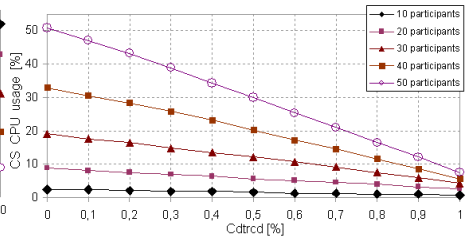
behaviors and different devices to each user. This paves the way for further analysis that can be realized by introducing heterogeneity in the composition of the participants set. Moreover, this enables future works on performance studies conducted from a user’s perspective, by focusing on parameters like, e.g., the performance of user devices, endpoint CPU utilization, perceived delay, etc..

*Complete model: moderation-based conference.* In this study, starting from a sample conference, we evaluate the difference between the CPU usage of the Conferencing Server either in the presence or in the absence of moderation. According to the User model described in Section 5, the evaluation can be characterized by the *Prel* parameter, that is the probability to release a conversation when talking (or, similarly, to be muted by the moderator in a moderation-based conference). Several values of the *Prel* parameter have been used for the analysis. Figure 22 and Figure 23 show the results of these simulations, by highlighting a reduction on the CPU usage as long as the *Prel* parameter increases. This is clearly due to the reduced number of incoming audio flows to be processed.

*Complete model: conference with transcoding* In the last experiment we evaluate the potential impact due to the presence of transcoding. We suppose that a portion *Cdtrcd* of participants does not need any transcoding, while the remaining part requires that the Conferencing Server transcodes messages (which increases server CPU usage). Several values of *Cdtrcd* have been used. The results are shown in Figure 24 and Figure 25. We can notice how the CPU workload is



**Fig. 24.** CPU usage vs number of users for different Cdtrcd



**Fig. 25.** CPU usage vs Cdtrcd for different conference sizes



progressively lighter as long as the number of participants needing server-side transcoding decreases (i.e., by increasing the value of the *Cdtrcd* parameter).

## 8 Conclusions and Future Works

In this paper we have used Stochastic Activity Networks to analyze the performance of a conferencing framework compliant with the most recent standard proposals currently under discussion within the international research community. We do believe that our work presents a number of interesting facets.

First, it does represent a successful example of cross fertilization between two extremely active (yet often uncorrelated) research communities: (i) the networking community on one side, which mainly focuses on real-world implementations of the designed architectures, by embracing a pure engineering approach; (ii) the “performability” community on the other side, which usually opts for a more structured approach, whereby the designed architectures are mostly studied through the application of formal methods. Thanks to such cross fertilization, we were able to reach the twofold objective of both validating the suitability of the formal characterization of our real-world implementation of the distributed conferencing scenario and assessing the validity of the measurements we performed on the field.

Second, once demonstrated the applicability of the formal approach, we moved the focus to the potential improvements deriving from the adoption of a compositional approach, which introduces the possibility of building model templates in order to enable the definition of families of models. By implementing a library of reusable SANs specifically devised to model the behavior of the various components of the conferencing framework, we were able to analyze the main performance figures associated with our current implementation of the conferencing framework through the proper composition of SAN templates. The results of the experiments we conducted clearly demonstrate that the proposed approach can be fruitfully exploited in order to: (i) easily compose and build the complete system model; (ii) analyze in an agile fashion the system’s behavior; (iii) simplify the very first phases of the entire life cycle of a real system (i.e., before its actual deployment).

The promising results obtained so far, encourage us to keep on investigating the proposed approach, by extending the modeling power through the adoption of multi-formalism techniques. Our aim is to further investigate model templates by moving the focus to the study of advanced functionality of the real system, with special regard to non functional requirements associated with its dependability properties. We have already started to work on the analysis of the system’s behavior in the presence of faults (either accidental or due to malicious users’ behaviors) and we plan to present the results of this further study as part of the dissemination activities related to our research efforts.

Automatic generation of Model Classes from Model Templates is made possible by the definition of *instancing functions*. Its full implementation is part of the ongoing work about the development of a complete generic formal modeling approach.

## References

1. Camarillo, G., Garcia-Martin, M.A.: The 3G IP Multimedia Subsystem (IMS): Merging the Internet and the Cellular Worlds, 3rd edn. (September 2008)
2. Vittorini, V., Iacono, M., Mazzocca, N., Franceschinis, G.: The OsMoSys approach to multi-formalism modeling of systems. *Software and System Modeling* 3(1), 68–81 (2004)
3. Buono, A., Loreto, S., Miniero, L., Romano, S.P.: A distributed IMS enabled conferencing architecture on top of a standard centralized conferencing framework. *IEEE Communications Magazine* 45(3), 152–159 (2007)
4. 3GPP. Conferencing using the IP multimedia (IM) core network (CN) subsystem; stage 3. Technical report, 3GPP (March 2006)
5. Barnes, M., Boulton, C., Levin, O.: Rfc 5239 - a framework for centralized conferencing. Request for comments, IETF (June 2008)
6. Amirante, A., Castaldi, T., Miniero, L., Romano, S.P.: Improving the Scalability of an IMS-Compliant Conferencing Framework Part II: Involving Mixing and Floor Control. In: Schulzrinne, H., State, R., Niccolini, S. (eds.) IPTComm 2008. LNCS, vol. 5310, pp. 174–195. Springer, Heidelberg (2008)
7. Amirante, A., Castaldi, T., Miniero, L., Romano, S.P.: Improving the Scalability of an IMS-Compliant Conferencing Framework Part II: Involving Mixing and Floor Control. In: Schulzrinne, H., State, R., Niccolini, S. (eds.) IPTComm 2008. LNCS, vol. 5310, pp. 174–195. Springer, Heidelberg (2008)
8. Truchly, P., Golha, M., Tomas, F., Radoslav, G., Legen, M.: Simulation of IMS using current simulators. In: 50th International Symposium on ELMAR, vol. 2, pp. 545–548 (September 2008)
9. Gupta, P., Shmatikov, V.: Security analysis of voice-over-ip protocols. In: Proceedings of the 20th IEEE Computer Security Foundations Symposium, pp. 49–63. IEEE Computer Society, Washington, DC (2007)
10. Shankesi, R., AlTurki, M., Sasse, R., Gunter, C.A., Meseguer, J.: Model-Checking DoS Amplification for VoIP Session Initiation. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 390–405. Springer, Heidelberg (2009)
11. Luo, A., Lin, C., Wang, K., Lei, L., Liu, C.: Quality of protection analysis and performance modeling in IP multimedia subsystem. *Comput. Commun.* 32, 1336–1345 (2009)
12. Guimarães, A.P., Maciel, P.R.M., Matias Jr., R.: Quantitative Analysis of Dependability and Performability in Voice and Data Networks. In: Meghanathan, N., Kaushik, B.K., Nagamalai, D. (eds.) CCSIT 2011, Part II. CCIS, vol. 132, pp. 302–312. Springer, Heidelberg (2011)
13. Huaxu, W., GuiPing, S., Yanlan, D.: SIP Modeling and Simulation, pp. 397–431. CRC Press (2008)
14. Bo, C., Junliang, C., Min, D.: Petri net based formal analysis for multimedia conferencing services orchestration. *Expert Systems with Applications* 39, 696–705 (2012)
15. Owezarski, P., Boyer, M.: Modeling of Multimedia Architectures: The Case of Videoconferencing with Guaranteed Quality of Service, pp. 501–525. ISTE (2010)
16. Franceschinis, G., Gribaudo, M., Iacono, M., Marrone, S., Moscato, F., Vittorini, V.: Interfaces and binding in component based development of formal models. In: Proceedings of the 4th International ICST Conference on Performance Evaluation Methodologies and Tools, VALUETOOLS 2009, pp. 44:1–44:10. ICST (2009)

17. Moscato, F., Vittorini, V., Amato, F., Mazzeo, A., Mazzocca, N.: Solution work-flows for model-based analysis of complex systems. *IEEE T. Automation Science and Engineering* 9(1), 83–95 (2012)
18. Zhang, W.J., Li, Q., Bi, Z.M., Zha, X.F.: A generic Petri net model for flexible manufacturing systems and its use for FMS control software testing. *International Journal of Production Research* 38(50), 1109–1131 (2000)
19. Bondavalli, A., Lollini, P., Montecchi, L.: Analysis of User Perceived QoS in Ubiquitous UMTS Environments Subject to Faults. In: Brinkschulte, U., Givargis, T., Russo, S. (eds.) *SEUS 2008*. LNCS, vol. 5287, pp. 186–197. Springer, Heidelberg (2008)
20. Tiassou, K., Kanoun, K., Ka, M., Seguin, C., Papadopoulos, C.: Modeling Aircraft Operational Reliability. In: Flammini, F., Bologna, S., Vittorini, V. (eds.) *SAFECOMP 2011*. LNCS, vol. 6894, pp. 157–170. Springer, Heidelberg (2011)
21. Chiaradonna, S., Di Giandomenico, F., Lollini, P.: Definition, implementation and application of a model-based framework for analyzing interdependencies in electric power systems. *International Journal of Critical Infrastructure Protection* 4(1), 24–40 (2011)
22. Courtney, T., Gaonkar, S., Keefe, K., Rozier, E., Sanders, W.H.: Möbius 2.3: An extensible tool for dependability, security, and performance evaluation of large and complex system models. In: *DSN*, pp. 353–358. IEEE (2009)

# Aggregating Causal Runs into Workflow Nets

Boudewijn F. van Dongen<sup>1</sup>, Jörg Desel<sup>2</sup>, and Wil M.P. van der Aalst<sup>1</sup>

<sup>1</sup> Department of Mathematics and Computer Science,  
Technische Universiteit Eindhoven, The Netherlands  
{B.F.v.Dongen,W.M.P.v.d.Aalst}@tue.nl

<sup>2</sup> Department of Software Engineering,  
FernUniversität in Hagen, Germany  
joerg.desel@fernuni-hagen.de

**Abstract.** This paper provides three aggregation algorithms for deriving system nets from sets of partially-ordered causal runs. The three algorithms differ with respect to the assumptions about the information contained in the causal runs. Specifically, we look at the situations where labels of conditions (i.e. references to places) or events (i.e. references to transitions) are unknown. Since the paper focuses on aggregation in the context of process mining, we solely look at workflow nets, i.e. a class of Petri nets with unique start and end places. The difference of the work presented here and most work on process mining is the assumption that events are logged as partial orders instead of linear traces. Although the work is inspired by applications in the process mining and workflow domains, the results are generic and can be applied in other application domains.

## 1 Introduction

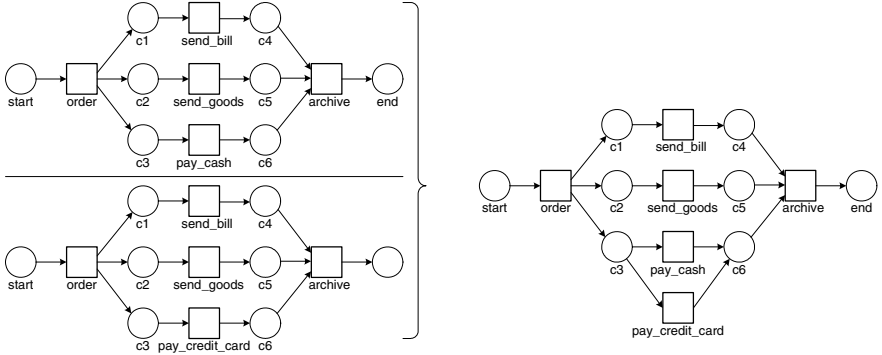
This paper proposes different approaches to “discover” process models from observed runs, i.e., runs (also known as causal nets or occurrence nets, cf. [14]) are aggregated into a single Petri net that captures the observed behavior. Runs provide information about events together with pre- and post-conditions which constitute a (partial) order between these events. This is useful in many domains where processes are studied based on their recorded behavior, such as:

- Discovering administrative processes by following the document flows in the organization with the goal to improve efficiency.
- Auditing processes in organizations in order to make sure that they conform to some predefined rules.
- Constructing enterprise models by observing transaction logs or document flows in enterprise systems such as SAP, Peoplesoft and Oracle.
- Monitoring the flow of SOAP messages between web-services to see how different services interact.
- Observing patient flows in hospitals to improve careflows and to verify medical guidelines.

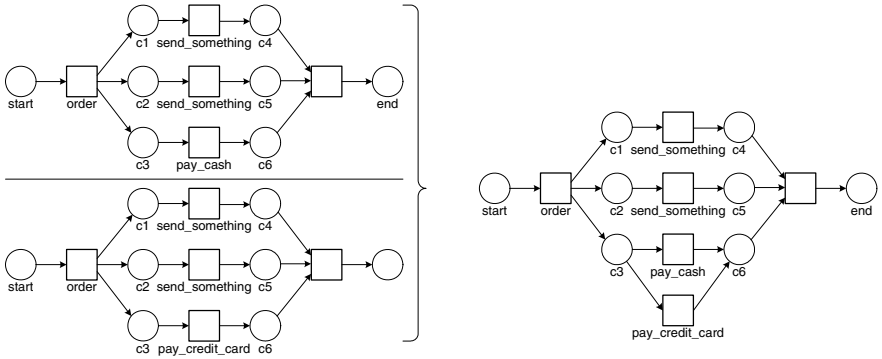
There are many techniques to discover process models based on sequential event logs (also known as transaction logs, audit trails, etc). People working on process mining techniques [6] generally tackle situations where processes may be concurrent and the set of observations is incomplete. Especially since the set of possible sequences is typically larger than the number of process instances, it is unrealistic to assume that all possible sequences have been observed.

In many applications, event logs are assumed to be linear, for example since all events are ordered in time. However, there are many processes where it is possible to monitor causal dependencies (e.g., by analyzing the dataflows). In the examples mentioned before, it is easy to identify situations where activities are causally linked by documents or explicit messages which can be monitored and hence explicit information about the causal dependencies between events is available. Consider for example service-oriented systems where one service calls another service. These services have input and output data. Using these dataflow one can find explicit causal dependencies. Furthermore, we encountered several Business Process Management (BPM) systems that actually log behavior using a representation similar to runs. The ad-hoc workflow management system InConcert of Tibco (formerly Xerox) allows end users to define and modify process instances (e.g., customer orders) while capturing the causal dependencies between the various activities. The representation used by these systems directly corresponds to the notion of runs. The analysis tool ARIS PPM (Process Performance Monitor) of IDS Scheer can extract runs represented as so-called *instance EPCs* (Event-driven Process Chains) from systems such as SAP R/3 and Staffware. These examples show that in real-life systems and processes runs can be recorded or already are being recorded, thus motivating the work presented in this contribution.

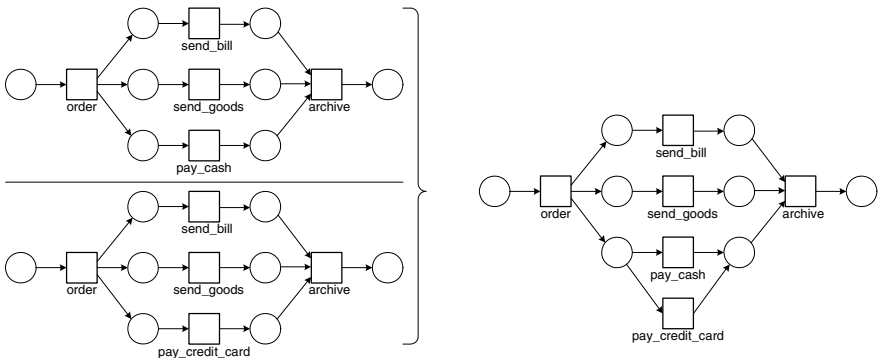
The remainder of this paper is structured as follows. After discussing related work in Section 2 and some preliminary definitions in Section 3, we provide algorithms for the aggregation of runs. In Section 4, three algorithms are presented for the aggregation of runs for the situations depicted in Figures 1 to 3. Figures 1 to 3 each show two runs on the left-hand side and the most likely candidate for the aggregated model on the right hand side. The first algorithm we present assumes we have full knowledge of each event, its preconditions and its postconditions. This is shown in Figure 1, where all events and conditions are labeled and these labels uniquely identify the corresponding transition or place in the aggregated model. Then, we assume that we cannot uniquely identify events, i.e. the label of an event may refer to multiple transitions, as shown in Figure 2, where *send\_goods* and *send\_bill* cannot be distinguished, since both of them are recorded as *send\_something*. In the aggregated model however, two occurrences of the transition *send\_something* have been identified. Finally, we provide an algorithm that assumes less knowledge about pre- and post-conditions, as shown in Figure 3, where no conditions have labels, while the corresponding aggregated model shows the same structure as in Figure 1. In Section 5, we formally prove that the algorithms we presented in Section 4 are correct, i.e. that the aggregated nets can reproduce the original causal nets. We conclude the paper in Section 6.



**Fig. 1.** Example of aggregating runs with known event and condition labels (Section 4.1)



**Fig. 2.** Example of aggregating runs with known condition labels and unknown or non-unique event labels (Section 4.2)



**Fig. 3.** Example of aggregating runs with known event labels and unknown condition labels (Section 4.3)

## 2 Related Work

For an extensive overview of the process mining domain, we refer to the recent book on process mining [2].

Since the mid-nineties several groups have been working on techniques for automated process discovery based on event logs [5,7,8,12,13,20,28,29]. In [6] an overview is given of the early work in this domain. The idea to apply process mining in the context of workflow management systems was introduced in [8]. In parallel, Datta [13] looked at the discovery of business process models. Cook et al. investigated similar issues in the context of software engineering processes [12]. Herbst [22] was one of the first to tackle more complicated processes, e.g., processes containing duplicate tasks. Most of the classical approaches have problems dealing with concurrency. The  $\alpha$ -algorithm [7] is an example of a simple technique that takes concurrency as a starting point. However, this simple algorithm has problems dealing with complicated routing constructs and noise (like most of the other approaches described in literature).

In all of the algorithms mentioned above, these tasks (i.e., events) in each case are *totally ordered* (typically based on the timestamps). In this paper, we take a different approach. We start by looking at so-called runs. These runs are a *partial* ordering on the tasks within each case. However, in addition to the partial ordering of tasks, we may have information about the local states of the system from which the logs originated, i.e. for each event the pre- and post-conditions are known. This closely relates to the process mining algorithms presented in [17] and [18]. However, also in these papers only causal dependencies between events are considered and no state information is assumed to be known.

The generation of system nets from their causal runs has been investigated before. The first publication on this topic is [27]. Here the basis is assumed to be the set of all runs. These runs are folded, i.e., events representing the occurrence of the same transition are identified, and so are conditions representing a token on the same place. In [15] a similar folding approach is taken, but there the authors start with a set of causal runs, as we do in the present paper. [15] does not present algorithms in details for the aggregation of runs but rather concentrates on correctness criteria for the derived system net. [11] presents an aggregation algorithm that constructs event-driven process chains from sets of partially ordered sets of events (without conditions).

The problem tackled in this paper is closely related to the so-called synthesis problem of Petri nets (see [16] and [19] for the synthesis of elementary net systems and [9] for more general cases). In this work, the behavior is given in the form of state graphs (where the events are known but the states are anonymous). In process mining, the observed behavior is not complete and it is not known, which process executions lead to identical global states. More recently, [26] extracts Petri nets from models which are based on Message Sequence Charts (MSCs), a concept quite similar to causal runs. Less related is the work presented in [21], where a special variant of MSCs is used to generate a system implementation.

In [24], so-called regions are defined for partial orders of events representing runs. These regions correspond to anonymous places of a synthesized place/

transition net, which can generate these partial orders. In contrast to our work, the considered partial orders are any linearizations of causal orders, i.e., two ordered events can either occur in a sequence (then there is a causal run with a condition "between" the events) or they can occur concurrently. Consequently, conditions representing tokens on places are not considered in these partial orders whereas our approach heavily depends on these conditions. More recently, this region-based approach was used for the synthesis of place/transition nets from sets of finite [10] or infinite [11] partially ordered sets of events.

### 3 Preliminaries

In this section, we introduce some basic definitions used in the remainder of this paper and formalize the starting point for the aggregation of partially ordered runs. Typically, a partial order is represented by a graph, and therefore we introduce some concepts related to graphs, such as a complete subgraph and a graph coloring. A graph-coloring is a way to label the nodes of a graph in such a way that no two neighboring nodes (i.e. nodes connected by an edge) have the same color.

#### Definition 3.1. (Graphs)

Let  $G = (N, E)$  be a directed graph, i.e.  $N$  is the set of nodes and  $E \subseteq N \times N$  is the set of edges. If  $N' \subseteq N$ , we say that  $G' = (N', E \cap (N' \times N'))$  is a subgraph of  $G$ .  $G$  is a *complete* graph if and only if  $E = (N \times N)$ .

In the sequel, we assume  $G = (N, E)$  is a directed graph.

#### Definition 3.2. (Undirected path in a graph)

Let  $a \in N$  and  $b \in N$ . We define an undirected path from  $a$  to  $b$  as a sequence of nodes denoted by  $\langle n_1, \dots, n_k \rangle$  with  $k \geq 1$  such that  $n_1 = a$  and  $n_k = b$  and  $\forall_{i \in \{1..k-1\}} ((n_i, n_{i+1}) \in E \vee (n_{i+1}, n_i) \in E)$ .

#### Definition 3.3. (Connected graph)

$G$  is a *connected* graph if for all  $n_1, n_2 \in N$  holds that there is an undirected path from  $n_1$  to  $n_2$ . A set of vertices  $N' \subseteq N$  generates a *maximal connected subgraph* if it is a maximal set of vertices generating a connected subgraph.

#### Definition 3.4. (Graph coloring)

Let  $\mu$  be a set of colors. A function  $f : N \rightarrow \mu$  is a coloring function if, for all  $(n_1, n_2) \in E$ , either  $n_1 = n_2$  or  $f(n_1) \neq f(n_2)$ .

#### Lemma 3.5. (Colorings on subgraphs can be combined)

Let  $E_1, E_2 \subseteq E$ , such that  $E_1 \cup E_2 = E$ . Furthermore, let  $f : N \rightarrow \mu$  be a coloring function on the graph  $(N, E_1)$  as well as a coloring function on the graph  $(N, E_2)$ . Then  $f$  is also a coloring function on  $G$ .

*Proof.* Let  $(n_1, n_2) \in E$  and  $n_1 \neq n_2$ . Since  $E = E_1 \cup E_2$ , we either have  $(n_1, n_2) \in E_1$  or  $(n_1, n_2) \in E_2$ . Since  $f$  is a coloring function on both  $(N, E_1)$  and  $(N, E_2)$ ,  $f(n_1) \neq f(n_2)$ .  $\square$



In graphs, we would like to be able to talk about predecessors and successors of nodes. Therefore, we introduce a special notation for that.

**Definition 3.6. (Pre-set and post-set)**

Let  $n \in N$ . We define  $\overset{G}{\bullet}n = \{m \in N \mid (m, n) \in E\}$  as the pre-set and  $n\overset{G}{\bullet} = \{m \in N \mid (n, m) \in E\}$  as the post-set of  $n$  with respect to the graph  $G$ . If the context is clear, the superscript  $G$  may be omitted, resulting in  $\bullet n$  and  $n\bullet$ .

As stated in the introduction, our starting point is not only a partial order of events within a case, but also information about the state of a case. Since we want to be able to represent both events and states, Petri nets provide a natural basis for our approach. In this paper, we use the standard definition of finite marked place/transition (P/T-nets) nets  $N = (P, T, F, M_0)$ .

**Definition 3.7. (Bag)**

Let  $S$  be a set. A *bag* over  $S$  is a function from  $S$  to the natural numbers  $\mathbb{N}$ .

**Definition 3.8. (Place/Transition net)**

$N = (P, T, F, M_0)$  is a marked place/transition net (or P/T-net) if:

- $P$  is a finite set of places,
- $T$  is a finite, non-empty set of transitions, such that  $P \cap T = \emptyset$ ,
- $F \subseteq (P \times T) \cup (T \times P)$  is the flow relation of the net,
- $M_0 : P \rightarrow \mathbb{N}$  represents the initial marking of the net, where a marking is a bag over the set of places  $P$ .

Note that any P/T-net  $N = (P, T, F, M_0)$  defines a directed graph  $((P \cup T), F)$ . In this paper, we restrict ourselves to P/T-nets where for each transition  $t$  holds that  $\bullet t \neq \emptyset$  and  $t\bullet \neq \emptyset$ .

**Definition 3.9. (Bag notations)**

We use square brackets for the enumeration of the elements of a bag representing a marking of a P/T-net. The sum of two bags  $(X \uplus Y)$ , the presence of an element in a bag ( $a \in X$ ), and the notion of subbags ( $X \leq Y$ ) are defined in a straightforward way, and they can handle a mixture of sets and bags. Furthermore,  $\biguplus_{a \in A} (f(a))$  denotes the sum over the bags that are results of function  $f$  applied to the elements  $a$  of a bag  $A$ .

Petri nets specify processes. The behavior of a Petri net is given in terms of causal nets, representing process instances (i.e. cases). Therefore, we introduce some concepts (notation taken from [14]). First, we introduce the notion of a causal net, this is a specification of one process instance.

**Definition 3.10. (Causal net)**

The P/T-net  $(C, E, K, S_0)$  is called a causal net if:

- for every place  $c \in C$  holds that  $|\bullet c| \leq 1$  and  $|c\bullet| \leq 1$ ,
- the transitive closure of  $K$  is irreflexive, i.e. it is a partial order on  $C \cup E$ ,
- for each place  $c \in C$  holds that  $S_0(c) = 1$  if  $\bullet c = \emptyset$  and  $S_0(c) = 0$  if  $\bullet c \neq \emptyset$ .

In causal nets, we refer to places as *conditions* and to transitions as *events*.

Each event of a causal net should refer to a transition of a corresponding P/T-net and each condition should refer to a token on some place of the P/T-net. These references are made by mapping the conditions and the events of a causal net onto places and transitions, respectively, of a Petri net. We call the combination of a causal net and such a mapping a *run*.

**Definition 3.11. (Run)**

A run  $(N, \alpha, \beta)$  of a P/T-net  $(P, T, F, M_0)$  is a causal net  $N = (C, E, K, S_0)$ , together with two mappings  $\alpha : C \rightarrow P$  and  $\beta : E \rightarrow T$ , such that:

- For each event (transition)  $e \in E$ , the mapping  $\alpha$  induces a bijection from  $\bullet e$  to  $\bullet\beta(e)$  and a bijection from  $e\bullet$  to  $\beta(e)\bullet$ ,
- $\alpha(S_0) = M_0$  where  $\alpha$  is generalized to markings by  $\alpha : (C \rightarrow \mathbb{N}) \rightarrow (P \rightarrow \mathbb{N})$ , such that  $\alpha(S_0)(p) = \sum_{c|\alpha(c)=p} S_0(c)$ .

The causal behavior of the P/T-net  $(P, T, F, M_0)$  is defined as its set of runs. To avoid confusion, the P/T-net  $(P, T, F, M_0)$  is called *system net* in the sequel.

In this paper, we take a set of runs as a starting point. From these runs, we generate a system net describing the behavior of all individual runs. Remember that we do not assume to have all runs as a starting point.

## 4 Aggregation of Runs

In this section, we introduce an approach that takes a set of runs as a starting point. From this set of runs, a system net is constructed. Moreover, we need to find a mapping from all the events and conditions in the causal nets to the transitions and places in the system net. From Definition 3.11, we know that there should exist a bijection between all conditions in the pre- or post-set of an event in the causal net and the pre- or post-set of a transition in a system net. *Therefore, two conditions belonging to the pre- or post-set of a single event should not be mapped onto the same place.* This restriction is in fact merely another way to express the fact that our P/T-nets do not allow for more than one edge between a place and a transition or vice versa. More generally, we define a labeling function on the nodes of a graph as a function that does not give the same label to two nodes that have a common element in their pre-sets or a common element in their post-sets.

**Definition 4.1. (Labeling function)**

Let  $\mu$  be a set of labels. Let  $G = (N, E)$  be a graph. Let  $R = \{(n_1, n_2) \subseteq N \times N \mid n_1 \overset{G}{\bullet} \cap n_2 \overset{G}{\bullet} \neq \emptyset \vee \overset{G}{\bullet} n_1 \cap \overset{G}{\bullet} n_2 \neq \emptyset\}$ . We define  $f : N \rightarrow \mu$  to be a *labeling function* if  $f$  is a coloring function on the graph  $(N, R)$ .

We focus on the aggregation of runs that originate from a Petri net with clearly defined starting state and completion state, i.e. processes that describe a lifespan of some case. This assumption is very natural in the context of workflow management systems. However, it applies to many other domains where processes are instantiated for specific cases. Hence, we will limit ourselves to a special class of Petri nets, namely workflow nets.

**Definition 4.2. (Workflow nets)**

A P/T-net  $N = (P, T, F, M_0)$  is a *workflow net* (WF-net) if:

1.  $P$  contains an input place  $p_{ini}$  such that  $\bullet p_{ini} = \emptyset$ ,
2.  $P$  contains an output place  $p_{out}$  such that  $p_{out} \bullet = \emptyset$ ,
3. there is a path from  $p_{ini}$  to every node and a path from every node to  $p_{out}$ ,
4.  $M_0 = [p_{ini}]$ , i.e. the initial marking marks only  $p_{ini}$ .

As a consequence, a WF-net has exactly one input place. When looking at a run of a WF-net, we can therefore conclude that there is exactly one condition containing a token initially and all other conditions do not contain tokens. A set of causal nets fulfilling this condition and some structural consequences is called a *causal set*.

**Definition 4.3. (Causal set)**

Let  $n \in \mathbb{N}$  and let  $\Phi = \{(C_i, E_i, K_i, S_i) \mid 0 \leq i < n\}$  be a set of causal nets. We call this set a *causal set* if the sets  $C_i$ ,  $E_i$  and  $K_i$  are pairwise disjoint and, for  $0 \leq i < n$  holds:

- $\sum_{c \in C_i} S_i(c) = 1$ , i.e. exactly one condition has an empty pre-set,
- If for some  $c \in C_i$ , holds that  $S_i(c) = 1$  and  $e \in c \bullet$ , then  $\{c\} = \bullet e$ , i.e. each event in the postset of an initially marked condition has only this condition in its preset,
- If for some  $c \in C_i$ , holds that  $c \bullet = \emptyset$  and  $e \in \bullet c$ , then  $e \bullet = \{c\}$ , i.e. each event in the preset of a condition with empty postset (representing a token on the place  $p_{out}$ ) has only this condition in its postset.

The concept of constructing a system net from a causal set is called *aggregation*. This concept can be applied if we assume that each causal net in the given set can be called a run of some system net. From Definition 3.11 we know that we need two mappings  $\alpha$  and  $\beta$  satisfying the two properties mentioned. Using the definition of a system net and the relation between system nets and runs, we can conclude that any aggregation algorithm should have the following functionality:

- it should provide the set of places  $P$  of the system net,
- it should provide the set of transitions  $T$  of the system net,
- it should provide the flow relation  $F$  of the system net,
- it should provide the initial marking  $M_0$  of the system net,
- for each causal net in the causal set, it should provide the mappings  $\alpha_i : C_i \rightarrow P$  and  $\beta_i : E_i \rightarrow T$ , in such a way that for all causal nets,  $\alpha_i(S_i)$  is the same (i.e. they have the same initial marking) and they induce bijections between pre- and post-sets of events and their corresponding transitions.

Each event that appears in a causal net has a corresponding transition in the original system net. Moreover, bijections exist between the pre- and post-sets of this event and the corresponding transitions. In order to express this in terms of labeling functions of causal nets, we formalize this concept using the notion of *transition equivalence*.

**Definition 4.4. (Transition equivalence)**

Let  $\mu, \nu$  be two disjoint sets of labels. Let  $\Phi = \{N_i = (C_i, E_i, K_i, S_i) \mid 0 \leq i < n\}$  be a causal set, and let  $\Psi = \{(\alpha_i : C_i \rightarrow \mu, \beta_i : E_i \rightarrow \nu) \mid 0 \leq i < n\}$  be a corresponding set of labeling functions for each  $(C_i, E_i, K_i, S_i)$ . We define  $(\Phi, \Psi)$  to *respect transition equivalence* if and only if for each  $e_i \in E_i$  and  $e_j \in E_j$  with  $\beta_i(e_i) = \beta_j(e_j)$  the following holds:

- for each  $(c_i, e_i) \in K_i$  there is a  $(c_j, e_j) \in K_j$  such that  $\alpha_i(c_i) = \alpha_j(c_j)$ ,
- for each  $(e_i, c_i) \in K_i$  there is a  $(e_j, c_j) \in K_j$  such that  $\alpha_i(c_i) = \alpha_j(c_j)$ .

Using the concepts of a causal set and transition equivalence, we introduce three aggregation algorithms with different requirements on the available information. First, in Section 4.1 we introduce an algorithm to aggregate causal nets where all places and transitions have known labels. Then, in Section 4.2 we show an algorithm that can deal with the situation where different transitions have the same label. The final algorithm, presented in Section 4.3, deals with the situation where transitions are correctly labeled, but places are not labeled at all.

**4.1 Aggregation with Known Labels**

In this section, we present an aggregation algorithm that assumes that we know all mapping functions, and that these mapping functions adhere to the definition of a run. To illustrate the aggregation process, we make use of a running example. Consider Figure 4 where four parts of runs are shown. We assume that the events  $A, B, C, D, E, F$  and  $G$  do not appear in any other part of each run.

Our first aggregation algorithm is called *ALK* (short for “All Labels Known”). This algorithm assumes known labels for events and known labels for conditions, such as in Figure 4. These labels refer to concrete transitions and places in the aggregated system net.

**Definition 4.5. (ALK aggregation algorithm)**

Let  $\mu, \nu$  be two disjoint sets of labels. Let  $\Phi$  be a causal set of size  $n$  with causal nets  $(C_i, E_i, K_i, S_i)$  ( $0 \leq i < n$ ).

Furthermore, let  $\{(\alpha_i : C_i \rightarrow \mu, \beta_i : E_i \rightarrow \nu) \mid 0 \leq i < n\}$  be a set of labeling functions respecting transition equivalence, such that for all causal nets  $\alpha_i(S_i)$  is the same. We construct the system net  $(P, T, F, M_0)$  belonging to these runs as follows:

- $P = \bigcup_{0 \leq i < n} \text{rng}(\alpha_i)$  is the set of places (note that  $P \subseteq \mu$ <sup>1</sup>),
- $T = \bigcup_{0 \leq i < n} \text{rng}(\beta_i)$  is the set of transitions (note that  $T \subseteq \nu$ ),
- $F = \bigcup_{0 \leq i < n} \{(\alpha_i(c), \beta_i(e)) \in P \times T \mid (c, e) \in K_i \cap (C_i \times E_i)\} \cup \bigcup_{0 \leq i < n} \{(\beta_i(e), \alpha_i(c)) \in T \times P \mid (e, c) \in K_i \cap (E_i \times C_i)\}$  is the flow relation,
- $M_0 = \alpha_0(S_0)$  is the initial marking.

<sup>1</sup> With  $\text{rng}$  we denote the range of a function, i.e.  $\text{rng}(f) = \{f(x) \mid x \in \text{dom}(f)\}$ .

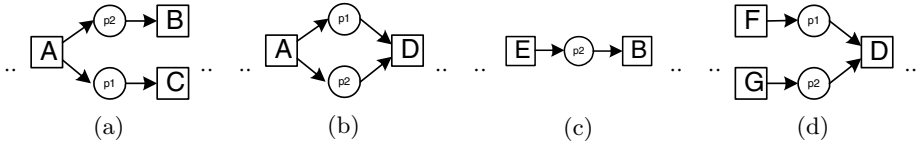


Fig. 4. Four examples of parts of runs

The result of the ALK aggregation algorithm applied to the parts presented in Figure 4 is shown in Figure 5. Another example is given in Figure 1.

The aggregated net shown in Figure 5 can generate the runs of Figure 4. However, it also allows for the possibility to execute transitions *F* followed by *C*. The token flow from *F* to *C* through place *p1* was never directly observed in any of the runs. Nonetheless, from the run in Figure 4(a) we can see that the *C* can fire using a token from *p1* and from the run in Figure 4(d) we can derive that transition *F* indeed produces this token, hence no “new” behavior is introduced.

The ALK algorithm is a rather trivial aggregation over a set of runs. Although we prove its correctness in Section 5.1, the algorithm relies on the assumption that the mapping functions  $\alpha_i$  and  $\beta_i$  are known for each causal net. Furthermore, we assume two sets of labels  $\mu$  and  $\nu$  to be known. However, when applying these techniques in the context of process mining, it is often not realistic to assume that all of these are present. Therefore, in the remainder of this paper, we relax some of these assumptions to obtain more usable aggregation algorithms for process mining.

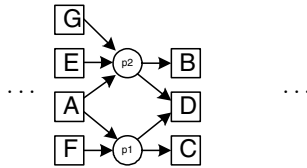


Fig. 5. The aggregated Petri net

### 4.2 Aggregation with Duplicate or Missing Transition Labels

In this section, we assume that the causal set used to generate the system net and the labeling functions do not respect transition equivalence (Definition 4.4). We introduce an algorithm to change the labeling function for events in such a way that this property holds again. In the domain of process mining, the problem of so-called “duplicate transitions” (i.e. several transitions with the same label) is well-known (cf. [3, 23, 25]). Therefore, there is a need for algorithms to find out which events actually belong to which transition. We assume that we have causal nets with labeling functions, where some events have the same label, even though they may refer to different transitions (see Figure 6). Note that this

figure is similar to Figure 4, except that we now labeled the events previously labeled with  $F$  and  $G$  with a new label  $X$ .

Since the previous aggregation algorithm given in Definition 4.5 assumes that transition equivalence holds, we provide an algorithm to redefine the labeling functions for events if this is not the case.

**Definition 4.6. (Relabeling algorithm)**

Let  $\mu, \nu$  be two disjoint sets of labels. Let  $\Phi = \{N_i \mid N_i = (C_i, E_i, K_i, S_i) \wedge 0 \leq i < n\}$  be a causal set and let  $\Psi = \{(\alpha_i : C_i \rightarrow \mu, \beta_i : E_i \rightarrow \nu) \mid 0 \leq i < n\}$  be a set of labeling functions in  $(C_i, E_i, K_i, S_i)$  such that  $\alpha_i(S_i)$  is the same for all causal nets. Furthermore, assume that  $\mu$  and  $\nu$  are minimal, i.e.  $\bigcup_{0 \leq i < n} \text{rng}(\alpha_i) = \mu$  and  $\bigcup_{0 \leq i < n} \text{rng}(\beta_i) = \nu$ . Let  $E^* = \bigcup_{0 \leq i < n} E_i$  be the set of all events in the causal set.

We define the relabeling algorithm as follows:

1. Define  $\bowtie \subseteq E^* \times E^*$  as an equivalence relation on the elements of  $E^*$  in such a way that  $e_i \bowtie e_j$  with  $e_i \in E_i$  and  $e_j \in E_j$  if and only if  $\beta_i(e_i) = \beta_j(e_j)$ ,  $\alpha_i(\overset{N_i}{\bullet}e_i) = \alpha_j(\overset{N_j}{\bullet}e_j)$ , and  $\alpha_i(e_i \overset{N_i}{\bullet}) = \alpha_j(e_j \overset{N_j}{\bullet})$ .
2. For each  $e \in E^*$ , we say  $\text{eqvl}(e) = \{e' \in E^* \mid e \bowtie e'\}$ .
3. Let  $\nu'$  be the set of equivalence classes of  $\bowtie$ , i.e.  $\nu' = \{\text{eqvl}(e) \mid e \in E^*\}$ .
4. For all causal nets  $(C_i, E_i, K_i, S_i)$  and labeling functions  $\alpha_i$ , define a labeling function  $\beta'_i : E_i \rightarrow \nu'$  such that for an event  $e_i$ ,  $\beta'_i(e_i) = \text{eqvl}(e_i)$ , i.e. it returns the equivalence class of  $\bowtie$  containing  $e_i$ .

After re-labeling the events, the part of the run shown in Figure 6(d) is relabeled to include the pre- and post-conditions. Figure 7 shows the fragment after relabeling. (We only show the relabeling with respect to the post-conditions.) Applying the ALK algorithm of Definition 4.5 to the relabeled runs yields the result as shown in Figure 8. Note that we do not show the  $\nu'$  labels explicitly, i.e.  $B$  refers to the equivalence class of events labeled  $B$ .

What remains to be shown is that our algorithm does not only work for our small running example, but also in the general case. The only difference between the assumptions in Definition 4.5 and Definition 4.6 is the requirement with respect to transition equivalence. Therefore, it suffices to show that after applying the relabeling algorithm on a causal set, we can establish transition equivalence.

**Property 4.7. (Transition equivalence holds after relabeling)**

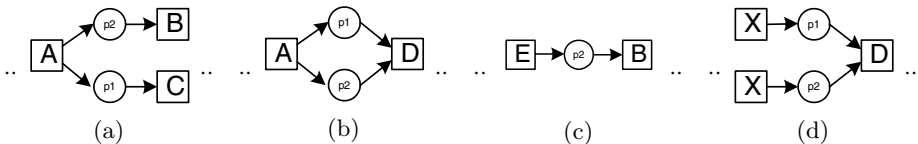


Fig. 6. Four examples of parts of runs

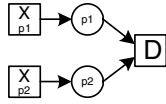


Fig. 7. The relabeled part of Figure 6(d)

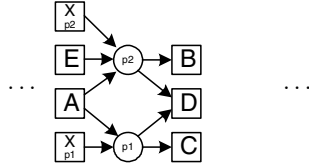


Fig. 8. Part of the aggregated net

Let  $\mu, \nu$  be two disjoint sets of labels. Let  $\Phi = \{(C_i, E_i, K_i, S_i) \mid 0 \leq i < n\}$  be a causal set, and let  $\Psi = \{(\alpha_i : C_i \rightarrow \mu, \beta_i : E_i \rightarrow \nu) \mid 0 \leq i < n\}$  be a set of labeling functions in  $(C_i, E_i, K_i, S_i)$ , such that  $\alpha_i(S_i)$  is the same for all causal nets. After applying the relabeling algorithm, the property of transition equivalence holds for  $(\Phi, \Psi')$ , with  $\Psi' = \{(\alpha_i : C_i \rightarrow \mu, \beta'_i : E_i \rightarrow \nu') \mid 0 \leq i < n\}$ , and  $\beta'_i$  as defined in Definition 4.6

*Proof.* We prove that Property 4.4 holds for  $(\Phi, \Psi')$  after applying the relabeling function. Assume  $(C_i, E_i, K_i, S_i)$  and  $(C_j, E_j, K_j, S_j)$  are two causal nets from  $\Phi$ . The new function  $\beta'_i$  is indeed a function, since for each event  $e_i \in E_i$  there exists exactly one equivalence class containing  $e_i$ . Furthermore, let  $e_i \in E_i$  and  $e_j \in E_j$ , such that  $\beta'_i(e_i) = \beta'_j(e_j)$ . We know that  $e_i \bowtie e_j$  and from the definition of  $\bowtie$ , we know that  $\alpha_i(\bullet e_i) = \alpha_j(\bullet e_j)$  and  $\alpha_i(e_i \bullet) = \alpha_j(e_j \bullet)$ , which directly implies transition equivalence.  $\square$

The algorithm presented above is capable of finding events that have the same label, but correspond to different transitions in the system net. When *no transition labels are known at all*, it can be applied to *find all transition labels*, by using an initial  $\nu = \{\tau\}$  and initial mapping functions  $\beta_i$ , mapping everything onto  $\tau$ . However, in that case, no distinction can be made between events that have the same pre- and post-set, but should have different labels. After applying this relabeling algorithm, the ALK algorithm of Section 4.1 can be used to find the system net belonging to the given causal nets.

### 4.3 Aggregation with Unknown Place Labels

In Section 4.2, we have shown a way to identify the transitions in a system net, based on the labels of events in causal nets. However, what if condition labels are not known? Notice that the difference to other approaches based on partial orders is that here we do know the conditions constituting the order between events but do *not* know which two conditions refer to a token in the same place of the P/T-net representing the process.

So, in this section, we take one step back. We assume all events to refer to the correct transition, as we did in Section 4.1 and we try to identify the labels of conditions. We introduce an algorithm to aggregate causal nets to a system net, such that the original causal nets are indeed runs of that system net.

In Figure 9, we again show our small example of the aggregation problem, only this time there are no labels for conditions  $p1$  and  $p2$ , which we did have in Figures 4 and 6

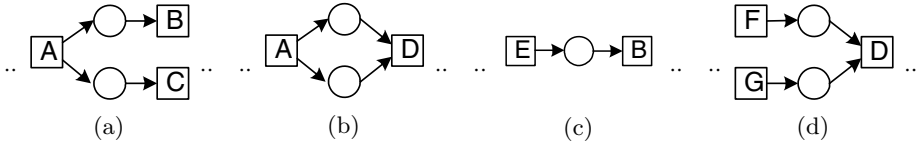


Fig. 9. Four examples of parts of runs

Consider the four runs of Figure 9. Remember that they are *parts* of causal nets, in such a way that the tasks  $A, B, C, D, E, F$  and  $G$  do not appear in any other way in another causal net. In contrast to the algorithms presented in previous sections, we cannot always derive a unique aggregated system net for causal nets if we do not have labels for the conditions. Instead, we define an *aggregation class*, describing a class of WF-nets that could have generated these causal nets. The following table shows some requirements all WF-nets in the aggregation class of our example should satisfy.

Table 1. Information derived from runs shown in Figure 9

Fragment	Conclusions
Fig. 9(a)	$A \bullet = \bullet B \uplus \bullet C$
Fig. 9(b)	$A \bullet = \bullet D$
Fig. 9(c)	$E \bullet = \bullet B$
Fig. 9(d)	$F \bullet \uplus G \bullet = \bullet D$

The information in Table 1 is derived from the runs of Figure 9 in the following way. Figure 9(a) shows that the transition  $A$  produces two tokens in two places and that transitions  $B$  and  $C$  consume these two tokens, while at the same time they do not need more input. Hence, we can conclude that in any aggregated net, the multiset of tokens produced by  $A$  should be equal to the multiset of tokens consumed by  $B$  and  $C$  together, which is stated in the first line of Table 1.

In the general case, this information can be derived using the concept of a segment, which can be considered to be the context of a condition in a causal net. A segment consists of two sets of events (an input set and an output set), such that the tokens produced by the transitions in the system net, corresponding to the events in the input set are exactly the tokens consumed by the transitions corresponding to the events in the output set, i.e. we formally capture the relations described in Table 1.

**Definition 4.8. (Segment)**

Let  $N = ((C, E, K), S_0)$  be a causal net and let  $N' = (C', E_{in}, E_{out})$  be such that  $C' \subseteq C$ ,  $E_{in} \cup E_{out} \subseteq E$ ,  $E_{in} \neq \emptyset$  and  $E_{out} \neq \emptyset$ . We call  $N'$  a *segment* if:

- for all  $c \in C'$  holds that  $\bullet c \subseteq E_{in}$  and  $c \bullet \subseteq E_{out}$ , and
- for all  $e \in E_{in}$  holds that  $e \bullet \subseteq C'$ , and





Fig. 10. Two aggregated nets

- for all  $e \in E_{out}$  holds that  $\bullet e \subseteq C'$ , and
- the subgraph of  $N$  made up by  $C' \cup E_{in} \cup E_{out}$  is connected.

We call the events in  $E_{in}$  the input events and the events in  $E_{out}$  the output events.

A segment is called *minimal* if  $C'$  is minimal, i.e. if there does not exist a segment  $N'' = (C'', E'_{in}, E'_{out})$  with  $C'' \subset C'$  and  $C'' \neq \emptyset$ .

For the fragments of Figure 9, it is easy to see that each of them contains only one minimal segment, where the input events are the events on the left hand side and the output events are the events on the right hand side.

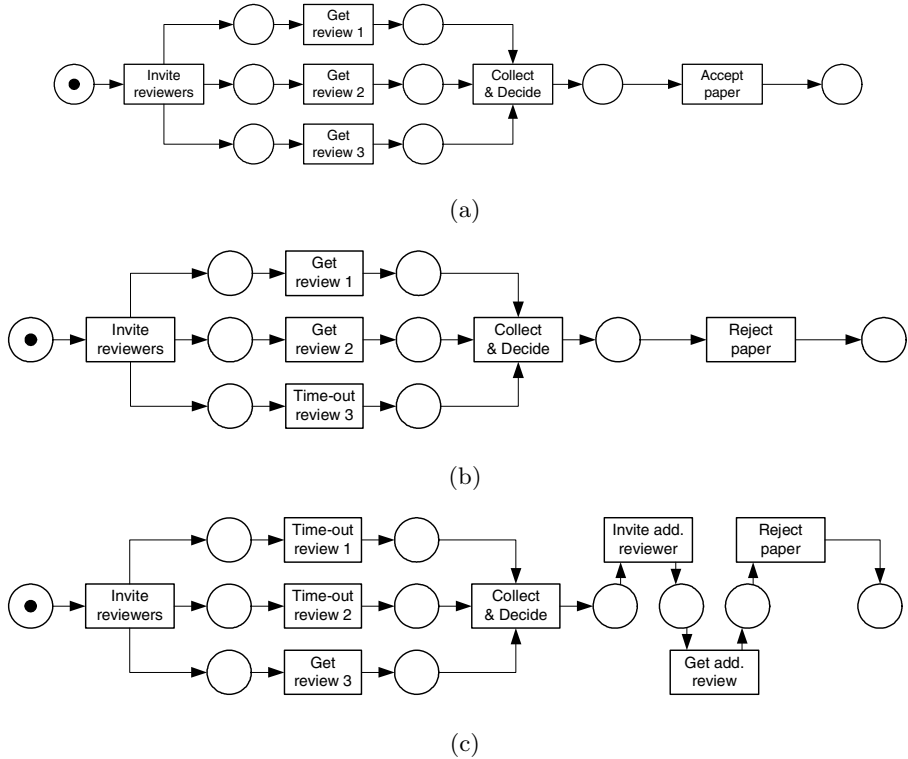
The meaning of a segment is as follows. If we have a run and a segment in that run, then we know that after all events in the input set of the segment occurred, all the events in the output set occurred in the execution represented by this run. This translates directly to a marking in a system net, since the occurrence of a set of transitions would lead to some marking (i.e. a bag over places), which enables another set of transitions. Furthermore, each transition only produces one token in each output place. Combining this leads to the fact that for each minimal segment in a causal net the bag of places following the transitions corresponding to the input events of the segment should be the same as the bag of places preceding the transitions corresponding to the output set of events, as indicated in Table 11.

Clearly, when looking only at these fragments, what we are looking for are the places that should be put between tasks  $A, E, F$  and  $G$  on the one hand, and  $B, C$  and  $D$  on the other hand. Therefore, we only focus on this part of the causal nets. For this specific example, there are two possibilities, both of which are equally correct, namely the two WF-net fragments shown in Figure 10.

From the small example, we have seen that it is possible to take a set of causal nets without labels for any of the conditions (but with labels for all the events) and to define a class of potential system nets of the causal nets. In the remainder of this section, we show that this is indeed possible for all causal sets. For this, we first introduce the NCL algorithm.

#### 4.4 NCL Algorithm

Before presenting the NCL algorithm (which stands for “No Condition Labels”), we first take a look at a more intuitive example. Consider Figure 11, where we



**Fig. 11.** Three causal nets of a review process of a paper

present three causal nets, each of which corresponds to a paper review process. In the first causal net, three reviewers are invited to review the paper and after the three reviews are received, the paper is accepted. In the second causal net, only two reviews are received (the third one is not received on time), but the paper is rejected nonetheless (apparently the two reviewers that replied rejected the paper). In the third example only one review is received in time, and therefore an additional reviewer is invited, which hands in his review in time, but does not accept the paper.

As we stated before, we define an aggregation class of a causal set that contains all WF-nets that are capable of generating the causal nets in the causal set. The information needed for this aggregation class comes directly from the causal nets, using minimal segments. In Table 2, we present the conclusions we can draw based on the three causal nets of Figure 11. In this table we consider *bags* of pre- and post-sets of transitions in the aggregation class. The information in this table is obtained from the causal nets in the following way. Consider for example Figure 11(a), where *Invite reviewers* is followed by *Get review 1*, *Get review 2* and *Get review 3*. This implies that the bag of output places of *invite reviewers* should be the same as the sum over the bags of the input places of *Get review 1*, *Get review 2* and *Get review 3*.

**Table 2.** Information derived from review example

Causal net	Conclusions on transitions in the aggregation class
Fig. 11(a)	<ul style="list-style-type: none"> <li>•“Invite reviewers” = <math>[p_{ini}]</math></li> <li>“Invite reviewers”• = •“Get review 1” <span style="float: right;">⊕</span></li> <li style="padding-left: 1.5em;">•“Get review 2” <span style="float: right;">⊕</span></li> <li style="padding-left: 1.5em;">•“Get review 3”</li> <li>“Get review 1” • ⊕ = •“Collect &amp; Decide”</li> <li>“Get review 2” • ⊕</li> <li>“Get review 3”•</li> <li>“Collect &amp; Decide”• = •“Accept paper”</li> <li> “Accept paper” •   = 1</li> </ul>
Fig. 11(b)	<ul style="list-style-type: none"> <li>•“Invite reviewers” = <math>[p_{ini}]</math></li> <li>“Invite reviewers”• = •“Get review 1” <span style="float: right;">⊕</span></li> <li style="padding-left: 1.5em;">•“Get review 2” <span style="float: right;">⊕</span></li> <li style="padding-left: 1.5em;">•“Time-out review 3”</li> <li>“Get review 1” • ⊕ = •“Collect &amp; Decide”</li> <li>“Get review 2” • ⊕</li> <li>“Time-out review 3”•</li> <li>“Collect &amp; Decide”• = •“Reject paper”</li> <li> “Reject paper” •   = 1</li> </ul>
Fig. 11(c)	<ul style="list-style-type: none"> <li>•“Invite reviewers” = <math>[p_{ini}]</math></li> <li>“Invite reviewers”• = •“Time-out review 1” <span style="float: right;">⊕</span></li> <li style="padding-left: 1.5em;">•“Time-out review 2” <span style="float: right;">⊕</span></li> <li style="padding-left: 1.5em;">•“Get review 3”</li> <li>“Time-out review 1” • ⊕ = •“Collect &amp; Decide”</li> <li>“Time-out review 2” • ⊕</li> <li>“Get review 3”•</li> <li>“Collect &amp; Decide”• = •“Invite add. reviewer”</li> <li>“Invite add. reviewer”• = •“Get add. review”</li> <li>“Get add. review”• = •“Reject paper”</li> <li> “Reject paper” •   = 1</li> </ul>

**Definition 4.9. (NCL algorithm: Aggregation Class)**

Let  $\Phi = \{(C_i, E_i, K_i, S_i) \mid 0 \leq i < n\}$  be a causal set, and let  $N = (P, T, F, M_0)$  be a marked WF-net. For each causal net  $N_i \in \Phi_i$ , let  $\beta_i : E_i \rightarrow T$  be a mapping from the events of that causal net to  $T$ , such that  $\beta_i$  is a labeling function for  $E_i$ . We define  $\mathcal{A}_\Phi$ , the *aggregation class* of  $\Phi$ , as the set of all pairs  $(N, \mathcal{B})$  such that the following conditions are satisfied:

1.  $T = \bigcup_{0 \leq i < n} \text{rng}(\beta_i)$  is the set of transitions, i.e. each transition appears as an event at least once in some causal net,

2. For all  $p \in P$  holds that  $\overset{N}{\bullet}p \cup p\overset{N}{\bullet} \neq \emptyset$ ,
3.  $M_0 = [p_{ini}]$  and  $\overset{N}{\bullet}p_{ini} = \emptyset$ ,
4.  $\mathcal{B}$  is the set of all labeling functions, i.e.  $\mathcal{B} = \{\beta_i \mid 0 \leq i < n\}$ . We use  $\beta_i \in \mathcal{B}$  to denote the labeling function for events belonging to  $N_i \in \Phi$ ,
5. For each causal net  $N_i = (C_i, E_i, K_i, S_i)$ , with  $e \in E_i$  and  $\beta_i(e) = t$  holds that if  $S_i(\overset{N_i}{\bullet}e) = 1$  then  $p_{ini} \in \overset{N_i}{\bullet}t$ ,
6. For each causal net  $N_i = (C_i, E_i, K_i, S_i)$ , with  $e \in E_i$  and  $\beta_i(e) = t$  holds that  $|t\overset{N_i}{\bullet}| = |e\overset{N_i}{\bullet}|$  and  $|\overset{N_i}{\bullet}t| = |\overset{N_i}{\bullet}e|$ ,
7. For each causal net  $N_i = (C_i, E_i, K_i, S_i)$ , with  $e \in E_i$ ,  $\beta_i(e) = t$  and  $T' \subseteq T$  holds that  $|t\overset{N_i}{\bullet} \cap \bigcup_{e' \in T'}(\overset{N_i}{\bullet}e')| \geq \sum_{e' \in E_i, \beta_i(e') \in T'} |e\overset{N_i}{\bullet} \cap \overset{N_i}{\bullet}e'|$ ,
8. For each causal net  $N_i = (C_i, E_i, K_i, S_i)$ , with  $e \in E_i$ ,  $\beta_i(e) = t$  and  $T' \subseteq T$  holds that  $|\bigcup_{e' \in T'}(t'\overset{N_i}{\bullet}) \cap \overset{N_i}{\bullet}t| \geq \sum_{e' \in E_i, \beta_i(e') \in T'} |e'\overset{N_i}{\bullet} \cap \overset{N_i}{\bullet}e|$ ,
9. For each causal net  $N_i = (C_i, E_i, K_i, S_i)$  and any minimal segment  $(C'_i, E_{in}, E_{out})$  of  $N_i$ , holds that  $\uplus_{e \in E_{in}} (\beta_i(e)\overset{N_i}{\bullet}) = \uplus_{e \in E_{out}} (\overset{N_i}{\bullet}\beta_i(e))$ .

Definition 4.9 defines an aggregation class of models in the following way:

- For each workflow net in the class, Items 1 to 4 define the transitions, places, initial marking and the labeling functions, labeling all events and conditions of each causal net with the transitions and places of that workflow net.
- Item 5 guarantees that all events in causal sets consuming the initial tokens are labeled with output transitions of the initially marked place in the workflow net.
- Item 6 guarantees that, for all events, the numbers of input and output conditions correspond to the numbers of input and output places of the corresponding transition.
- Items 7 and 8 refer to the token flow in the model, in relation to the causal nets, i.e. when considering the flow between a set of transitions and one other transitions (in any direction), the number of tokens ever observed in any causal set cannot be larger than the number of tokens allowed according to the model. Hence, choices in the model do not correspond to parallel behavior in any causal net.
- Figure 12 is used to gain more insight into Item 9 of Definition 4.9. In the lower causal net of that figure, there is a token traveling from  $A$  to  $D$  and another one from  $B$  to  $C$ . The upper causal net only connects  $A$  and  $C$ . Assuming that these are the only causal nets in which these transitions appear, we know that the conditions between  $A$  and  $D$  and between  $B$  and  $C$  should represent a token in the same place, since there is a minimal segment  $(\{c_4, c_5, c_6\}, \{A, B\}, \{C, D\})$  in the lower causal net and therefore,  $A \bullet \uplus B \bullet = \bullet C \uplus \bullet D = [p_1, 2p_2]$ .

Consider the information presented in Table 2 and the two Petri nets in Figure 13. Both nets in Figure 13 adhere to all constraints of Table 2. As this

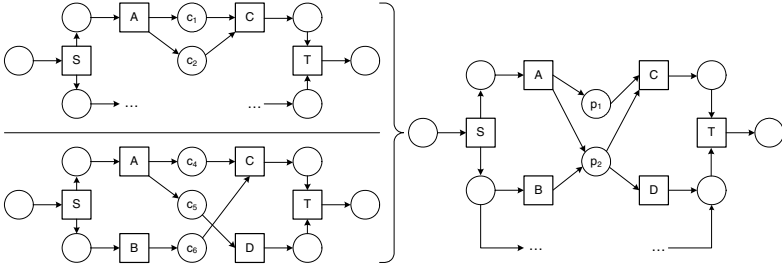


Fig. 12. Example explaining the use of bags

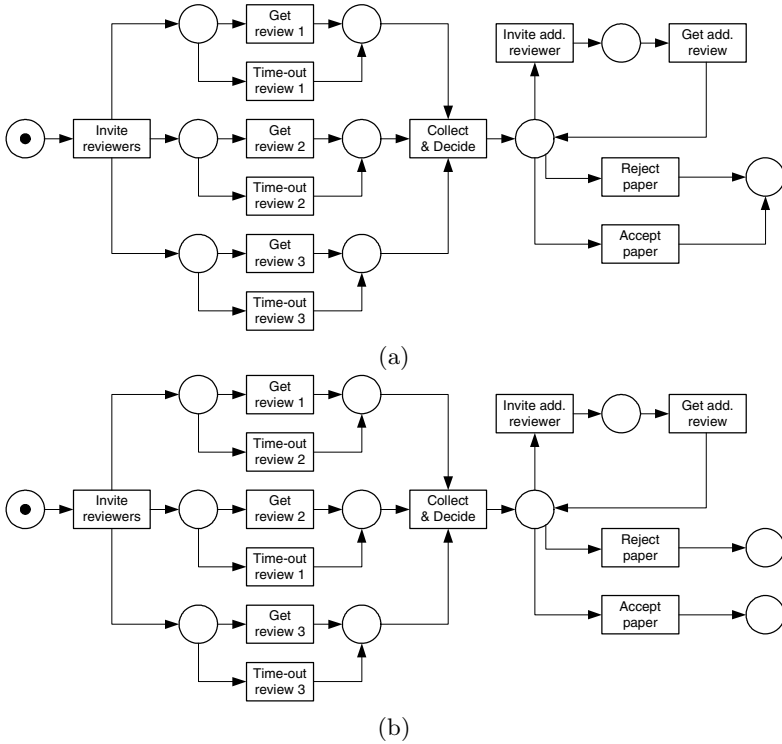


Fig. 13. Two possible aggregated nets, both obeying the constraints of Table 2

example shows, there is no unique Petri net satisfying all constraints. Instead, there is a class of nets satisfying all constraints.

The condition provided in Item 9 of Definition 4.9 provides the key to constructing the actual elements of the aggregation class. By considering all minimal segments in the provided runs that refer to the same transitions, possible sets of places can be identified that satisfy this condition. However, in this paper, we do not provide construction steps for constructing the aggregation class. Instead,

in the next section, we show that if a set of runs is generated by a system net, then that system net is a member of the aggregation class.

## 5 Correctness of the Aggregation Algorithms

In Section 4, we described three scenarios for which we can construct an aggregated net from a set of runs. In Section 4.1, we showed the ALK algorithm, which assumes that in the runs, all conditions and events are labeled with the corresponding places and transitions of the aggregated net. In Section 4.2, we showed that in case some transition labels are duplicated or missing, we can still use the ALK algorithm after relabeling the transitions using the surrounding places. Finally, in Section 4.3, we presented the NCL algorithm that provides an aggregation class of nets that are all capable of reproducing the given set of runs in which none of the conditions is labeled.

In this section, we formally prove correctness of the ALK and the NCL algorithms.

### 5.1 Correctness of the ALK Algorithm

The ALK algorithm defines a single aggregated net for a given set of runs. In order to prove its correctness, we show that the runs used as input can be generated by the resulting aggregated net.

**Property 5.1. (The ALK algorithm is correct)**

For all  $0 \leq i < n$  and  $N_i = (C_i, E_i, K_i, S_i)$ , the tuple  $(N_i, \alpha_i, \beta_i)$  is indeed a run of  $N = (P, T, F, M_0)$  (i.e., the requirements stated in Definition 3.11 are fulfilled).

*Proof.* Since we assumed that all causal nets  $N_i = (C_i, E_i, K_i, S_i)$  are elements of the causal set  $\Phi$ , we need to prove the following for each  $\alpha_i$  and  $\beta_i$ .

1.  $\alpha_i$  is a function from  $C_i$  onto  $P$ . This trivially follows from Definition 4.5.
2.  $\beta_i$  is a function from  $E_i$  onto  $T$ . This trivially follows from Definition 4.5.
3.  $\alpha_i(S_i) = M_0$  holds by definition, since it holds for  $S_0$  and for all causal nets,  $\alpha_i(S_i)$  is the same.
4. For each event  $e \in E_i$ , the mapping  $\alpha_i$  induces a bijection from  $\bullet e$  to  $\bullet\beta_i(e)$  and a bijection from  $e\bullet$  to  $\beta_i(e)\bullet$ .

Let  $e \in E_i$ . We start by showing that  $\alpha_i(\bullet^N e) = \bullet^N \beta_i(e)$  and  $\alpha_i(e^N \bullet) = \beta_i(e)^N \bullet$ . Assume  $p \in \alpha_i(\bullet^N e) \setminus \bullet^N \beta_i(e)$ , i.e. there exists a  $c \in C_i$  with  $(c, e) \in K_i$ , such that  $p = \alpha_i(c)$ ,  $\beta_i(e) = t$  and  $(p, t) \notin F$ . Clearly this contradicts with the definition of  $F$  in Definition 4.5. Now assume  $p \in \bullet^N \beta_i(e) \setminus \alpha_i(\bullet^N e)$ , i.e. there is a  $(p, t) \in F$  such that  $\beta_i(e) = t$  and there is no  $c \in C_i$  with  $\alpha_i(c) = p$ , such that  $(c, e) \in K_i$ . If this is the case in all causal nets for  $0 \leq i < n$ , then this leads to a contradiction since this would imply  $(p, t) \notin F$  (cf. Definition of  $F$  in Definition 4.5). If there is a  $0 \leq j < n$ , such that  $(c', e') \in K_j$  with

$\beta_j(e') = t$  and  $\alpha_j(c') = p$ , then there has to be a  $c \in C_i$  such that  $(c, e) \in K_i$ , since  $\alpha_i(\overset{N_i}{\bullet}e) = \alpha_j(\overset{N_i}{\bullet}e')$  (cf. Definition 4.4). Combined with the fact that  $\alpha_i$  and  $\beta_i$  are labeling functions,  $\alpha_i(\overset{N_i}{\bullet}e) = \overset{N_i}{\bullet}\beta_i(e)$  and  $\alpha_i(e \overset{N_i}{\bullet}) = \beta_i(e) \overset{N_i}{\bullet}$  yields the bijection. Similar arguments apply for the post-set. □

Property 5.1 shows that the ALK algorithm indeed results in a system net of which the causal nets used as input are runs.

### 5.2 Correctness of the NCL Algorithm

In case that no condition labels are present, the NCL algorithm defines an equivalence class of aggregated nets. In this section, we show that for each net in this aggregation class, the causal nets used as inputs can be considered runs. Furthermore, we show that if we take the runs of a sound workflow model as input, then that model is part of the aggregation class.

Definition 4.9 defines a *finite* class of WF-nets for a causal set. What remains to be given are the conditions under which it is a finite *non-empty* class of Petri nets and the proof that each Petri net with its mappings is indeed a system net for the causal set. To prove this, we first introduce the concept of a *condition graph*.

#### Definition 5.2. (Condition graph)

Let  $N_i = (C_i, E_i, K_i, S_i)$  be a causal net. The undirected graph  $\Delta_{N_i} = (C_i, A)$ , with  $A = \{(c_1, c_2) \in C_i \times C_i \mid \exists_{e \in E_i} \{c_1, c_2\} \subseteq \overset{N_i}{\bullet}e \vee \{c_1, c_2\} \subseteq e \overset{N_i}{\bullet}\}$  is called a *condition graph*. Note that  $(c_1, c_2) \in A$  implies that  $(c_2, c_1) \in A$ .

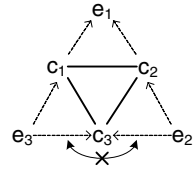
We use condition graphs to prove that each Petri net with its mappings in the aggregation class of a causal set is indeed a system net for that causal set. For this, we first introduce some lemmas on these condition graphs that show the relation between condition graphs and causal nets. We start by showing that pre- and post-sets of events correspond to complete subgraphs in the condition graph, i.e. subgraphs where each pair of nodes is connected by an edge.

#### Lemma 5.3. (Pre- and post sets relate to complete subgraphs in condition graphs)

Let  $N_i = (C_i, E_i, K_i, S_i)$  be a causal net and  $\Delta_{N_i} = (C_i, A)$  its condition graph. We show that, for each  $e \in E_i$ , holds that  $\Delta_{N_i}$  restricted to  $\overset{N_i}{\bullet}e$  is a complete subgraph and  $\Delta_{N_i}$  restricted to  $e \overset{N_i}{\bullet}$  is a complete subgraph. Furthermore, for each complete subgraph  $(C', A')$ , there exists an  $e \in E_i$  such that  $C' \subseteq \overset{N_i}{\bullet}e$  or  $C' \subseteq e \overset{N_i}{\bullet}$ .

*Proof.* Since for all  $\{c_1, c_2\} \subseteq \overset{N_i}{\bullet}e$ , holds that  $(c_1, c_2) \in A$  by definition, the first part is correct. The same applies to  $e \overset{N_i}{\bullet}$ . Now assume  $(C', A')$  is a complete subgraph. Assume  $\{c_1, c_2\} \subseteq C'$ . and  $c_1 \neq c_2$ . Since we are looking at a complete subgraph, we know  $(c_1, c_2) \in A'$ , therefore there exists an  $e_1 \in E_i$ , such that  $\{c_1, c_2\} \subseteq \overset{N_i}{\bullet}e_1$  or  $\{c_1, c_2\} \subseteq e_1 \overset{N_i}{\bullet}$ .

Assume  $\{c_1, c_2\} \subseteq \bullet^{N_i} e_1$  (The proof is symmetrical for  $e_1 \bullet^{N_i}$ ). Now assume  $c_3 \in C'$  such that  $c_1 \neq c_3$  and  $c_2 \neq c_3$ . Let  $c_3 \notin \bullet^{N_i} e_1$ . We show that this leads to a contradiction. Since for all  $c \in C$  holds that  $|c \bullet^{N_i}| \leq 1$  and  $\{c_1, c_2\} \subseteq \bullet^{N_i} e_1$ , we know that there must be an  $e_2 \in E_i$ , such that  $\{c_2, c_3\} \subseteq e_2 \bullet^{N_i}$ .



Similarly, we know that there is an  $e_3 \in E_i$ , such that  $\{c_1, c_3\} \subseteq e_3 \bullet^{N_i}$ . However, since  $|\bullet^{N_i} c_3| \leq 1$ , this implies that  $e_2 = e_3$  and thus  $\{c_1, c_2, c_3\} \subseteq e_2 \bullet^{N_i}$ .  $\square$

Using the fact that each pre- and post-set correspond to a complete subgraph, we can infer that each minimal segment in a causal net corresponds to a connected subgraph in the condition graph, i.e. a subgraph such that there is a path between each two nodes. Furthermore, we show that these connected subgraphs are maximal, i.e. all nodes in the subgraph are only connected to nodes inside the subgraph.

**Lemma 5.4. (Minimal segments correspond to maximal connected subgraphs in condition graphs)**

Let  $N_i = (C_i, E_i, K_i, S_i)$  be a causal net and  $\Delta_{N_i} = (C_i, A)$  its condition graph. Let  $(C', E_{in}, E_{out})$  be a minimal segment in  $N_i$ . We show that  $(C', A \cap (C' \times C'))$  is a maximal connected subgraph of  $\Delta_{N_i}$ .

*Proof.* From Definition 4.8 we know that the graph  $(C' \cup E_{in} \cup E_{out}, K_i \cap ((C' \cup E_{in} \cup E_{out}) \times (C' \cup E_{in} \cup E_{out})))$  is a connected graph. Now, let  $c \in C'$  be a condition in the minimal segment and assume that  $\{e_{in}\} = \bullet^{N_i} c$  and  $\{e_{out}\} = c \bullet^{N_i}$ . From Lemma 5.3, we know that  $e_{in} \bullet^{N_i}$  and  $\bullet^{N_i} e_{out}$  make up a complete subgraph in  $\Delta_{N_i}$  and since  $c \in \bullet^{N_i} e_{out} \cap e_{in} \bullet^{N_i}$  that these two complete subgraphs make up a connected subgraph. By induction over the elements of  $C'$ , it is easy to show that  $C'$  makes up a connected subgraph in  $\Delta_{N_i}$ . Therefore, each minimal segment defines a complete subgraph  $G'$  in  $\Delta_{N_i}$ . Furthermore, let  $G' = (C', A)$  be the connected subgraph of  $\Delta_{N_i}$  corresponding to the segment. Let  $c \in C_i \setminus C'$  and assume there exists a  $c' \in C'$ , such that  $(c, c') \in A$ . This implies that there is an  $e \in E_i$ , such that  $\{c, c'\} \subseteq \bullet^{N_i} e$  or  $\{c, c'\} \subseteq e \bullet^{N_i}$ . However, this implies that  $e \in E_{in}$  or  $e \in E_{out}$ , either of which imply that  $c \in C'$ . Therefore, such a  $c$  does not exist and  $G'$  is maximal.  $\square$

At this point, we look at the definitions of Section 3 again. If we assume that we have a system net and the causal behavior of this system net, we can derive the next lemma using Definition 3.4

**Lemma 5.5. (System nets color condition graphs)**

Let  $N = (P, T, F, M_0)$  be a system net and  $(N_i, \alpha_i, \beta_i)$  be a run of that system net, with  $N_i = (C_i, E_i, K_i, S_i)$ . Furthermore, let  $\Delta_{N_i} = (C_i, A)$  be the condition graph of  $N_i$ . The mapping  $\alpha_i : C_i \rightarrow P$  is a coloring function of  $\Delta_{N_i}$ , with the set of colors being  $P$ .

*Proof.* Let  $n_1, n_2 \in C_i$  be two nodes in  $\Delta_{N_i}$  with  $n_1 \neq n_2$ . For  $\alpha_i$  to be a coloring,  $\alpha_i(n_1) \neq \alpha_i(n_2)$  should hold if  $(n_1, n_2) \in A$ . Assume  $(n_1, n_2) \in A$ . This



means that there is an  $e \in E_i$  such that  $\{n_1, n_2\} \subseteq \bullet^{N_i} e$  or  $\{n_1, n_2\} \subseteq e^{N_i}$ . From Definition 3.11, we know that  $\alpha_i$  induces a bijection from  $\bullet^{N_i} e$  to  $\bullet^{N_i} \beta_i(e)$  and from  $e^{N_i}$  to  $\beta_i(e)^{N_i}$ . Therefore,  $\alpha_i(n_1) \neq \alpha_i(n_2)$ .  $\square$

We have shown that system nets color condition graphs. However, we can go one step further and introduce the concept of a condition coloring, which is a coloring on the condition graph, such that the coloring function, when applied to the conditions in a causal net, induces local bijections for the input and output sets of events.

**Definition 5.6. (Condition coloring)**

Let  $\Phi$  be a causal set and let  $\mathcal{A}_\Phi$  be the aggregation class of  $\Phi$ . Moreover, let  $(N, \mathcal{B}) \in \mathcal{A}_\Phi$ , with  $N = (P, T, F, M_0)$  and let  $N_i = (C_i, E_i, K_i, S_i) \in \Phi$  be a causal net and  $\Delta_{N_i} = (C_i, A)$  be the condition graph of  $N_i$ . Assume  $\alpha_i : C_i \rightarrow P$  is a function, such that  $\alpha_i$  is a coloring on  $\Delta_{N_i}$  and for all  $c \in C_i$  holds that  $\alpha_i(c) \in \{p \in P \mid \beta(\bullet^{N_i} c) \subseteq \bullet^N p \wedge \beta(c^{N_i}) \subseteq p^{N_i}\}$ .<sup>2</sup> We then call  $\alpha_i$  a condition coloring of  $\Delta_{N_i}$ .

The concept of a condition coloring we introduced here is often referred to in mathematics as a list coloring.

**Lemma 5.7. (Condition coloring induces bijections)**

Let  $\Phi$  be a causal set and let  $\mathcal{A}_\Phi$  be the aggregation class of  $\Phi$ , and let  $(N, \mathcal{B}) \in \mathcal{A}_\Phi$ , with  $N = (P, T, F, M_0)$ . Let  $N_i = (C_i, E_i, K_i, S_i) \in \Phi$  be a causal net and  $\Delta_{N_i} = (C_i, A)$  be the condition graph of  $N_i$ . Let  $\alpha_i : C_i \rightarrow P$  be a condition coloring of  $\Delta_{N_i}$ . We show that for all  $e \in E_i$ ,  $\alpha_i$  induces a bijection from  $\bullet^{N_i} e$  to  $\bullet^{N_i} \beta_i(e)$  and from  $e^{N_i}$  to  $\beta_i(e)^{N_i}$ .

*Proof.* The requirements stated in Definition 4.9, imply that  $|\bullet^{N_i} e| = |\bullet^N \beta(e)|$ . Furthermore, since  $\Delta_{N_i}$  restricted to  $\bullet^{N_i} e$  is a complete graph (Lemma 5.3), and  $\alpha_i$  is a coloring function (Lemma 5.5), we know that  $|\alpha_i(\bullet^{N_i} e)| = |\bullet^{N_i} e|$ . Since all elements in  $\bullet^{N_i} e$  are mapped to different colors. Combining both implies that  $|\alpha_i(\bullet^{N_i} e)| = |\bullet^N \beta_i(e)|$ .

For all  $c \in \bullet^{N_i} e$  holds that  $\alpha_i(c) \in \{p \in P \mid \beta_i(\bullet^{N_i} c) \subseteq \bullet^N p \wedge \beta_i(c^{N_i}) \subseteq p^{N_i}\}$  (Definition 5.6) and  $c^{N_i} = \{e\}$ , because  $N_i$  is a causal net we know that  $\alpha_i(c) \in \{p \in P \mid \beta_i(e) \in p^{N_i}\}$  and thus  $\alpha_i(c) \in \bullet^N \beta_i(e)$ . Since this holds for all  $c \in \bullet^{N_i} e$ , we can conclude that  $\alpha_i(\bullet^{N_i} e) \subseteq \bullet^N \beta_i(e)$ . By combining the above, we can conclude that  $\alpha_i(\bullet^{N_i} e) = \bullet^N \beta_i(e)$ , and thus that  $\alpha_i$  induces a bijection from  $\bullet^{N_i} e$  to  $\bullet^N \beta_i(e)$ . A similar proof holds for the mapping from  $e^{N_i}$  to  $\beta_i(e)^{N_i}$ .  $\square$

At this point we still need to prove the following for an arbitrary WF-net in the aggregation class. For each causal net in a causal set, we should be able to color its condition graph using a condition coloring. If we are able to construct such a coloring, we have satisfied the first requirement stated in Definition 3.11.

<sup>2</sup> Note that  $\beta$  is generalized, i.e. for a set  $E$  holds that  $\beta(E) = \{\beta(e) \mid e \in E\}$ .

**Lemma 5.8. (Condition coloring exists)**

Let  $\Phi$  be a causal set, let  $\mathcal{A}_\Phi$  be the aggregation class of  $\Phi$ , and let  $(N, \mathcal{B}) \in \mathcal{A}_\Phi$ , with  $N = (P, T, F, M_0)$ . Let  $N_i = (C_i, E_i, K_i, S_i) \in \Phi$  be a causal net and  $\Delta_{N_i} = (C_i, A)$  be the condition graph of  $N_i$ . Let  $\beta_i \in \mathcal{B}$  be the labeling function belonging to  $N_i$ . We show that we can construct a mapping  $\alpha_i : C_i \rightarrow P$ , such that  $\alpha_i$  is a condition coloring of  $\Delta_{N_i}$ .

*Proof.* First, we look at the initial condition, i.e. the initially marked source condition. Assume  $c \in C_i$  such that  $\overset{N_i}{\bullet}c = \emptyset$ . We call  $c \overset{N_i}{\bullet} = \{e\}$ . From the definition of a causal set (Def. 4.3), we know that  $\{c\} = \overset{N_i}{\bullet}e$  and thus that there is no  $c' \in C_i$  with  $c \neq c'$  and  $(c, c') \in A$ . We know that  $\overset{N_i}{\bullet}\beta_i(e) = \{p_{ini}\}$  (Def. 4.9). By setting  $\alpha_i(c) = p_{ini}$ , we have a correct coloring for the initial condition  $c$  in  $N$ .

Second, we look at the final conditions, i.e. the sink conditions. Assume  $c \in C_i$  such that  $c \overset{N_i}{\bullet} = \emptyset$ . We call  $\overset{N_i}{\bullet}c = \{e\}$ . From the definition of a causal set (Def. 4.3), we know that  $\{c\} = e \overset{N_i}{\bullet}$  and thus that there is no  $c' \in C_i$  with  $c \neq c'$  and  $(c, c') \in A$ . We know that  $|\beta_i(e) \overset{N_i}{\bullet}| = 1$  (Def. 4.9). We say that  $\beta_i(e) \overset{N_i}{\bullet} = \{p\}$ . By setting  $\alpha_i(c) = p$ , we have a correct coloring for any final condition  $c$  in  $N_i$ .

Finally, we split the graph up into two subgraphs. Let  $A_{in} = \{(c_1, c_2) \in A \mid \overset{N_i}{\bullet}c_1 = \overset{N_i}{\bullet}c_2\}$  and let  $A_{out} = \{(c_1, c_2) \in A \mid c_1 \overset{N_i}{\bullet} = c_2 \overset{N_i}{\bullet}\}$ . Using the definition of a condition graph it is easy to see that  $A_{in} \cup A_{out} = A$ . We now show that for each subgraph  $\delta_{in}(N_i) = (C, A_{in})$  and  $\delta_{out}(N_i) = (C, A_{out})$  we can construct at least one condition coloring. Then, we show that there is at least one condition coloring that is the same for both subgraphs after which we can use Lemma 3.5 to show that this is a condition coloring on the complete graph.

Consider the subgraph  $\delta_{in}(N_i) = (C, A_{in})$ . Using Lemma 5.3, it is easy to see that this graph consists of several complete components and that each component is a complete graph. Let  $e \in E_i$ . We know that  $e \overset{N_i}{\bullet} \subseteq C$  and that  $e \overset{N_i}{\bullet}$  defines a complete component in  $\delta_{in}(N_i)$ . Now, let  $V_1, \dots, V_n$  be maximal sets, such that for each  $0 < i \leq n$  holds that  $V_i \subseteq E \overset{N_i}{\bullet}$  and for all  $c_1, c_2 \in V_i$  holds that  $c_1 \overset{N_i}{\bullet} = c_2 \overset{N_i}{\bullet}$ . For each  $V_i$  and  $c \in V_i$ , we say that  $V_{i,in} = \{e\}$  and  $V_{i,out} = c \overset{N_i}{\bullet}$ .

From Definition 5.6, we know that for each  $c \in V_i$  must hold that  $\alpha_i(c) \in \{p \in P \mid \beta_i(\{e\}) \subseteq \overset{N_i}{\bullet}p \wedge \beta(V_{i,out}) \subseteq p \overset{N_i}{\bullet}\}$ . Using Item 7 of Definition 4.9, we first prove a necessary condition for this. Assume  $\beta_i(\{e\}) = \{t\}$ , and  $\beta_i(V_{i,out}) = T' = \{t'\}$ . Item 7 shows us that  $|t \overset{N_i}{\bullet} \cap \overset{N_i}{\bullet}t'| \geq \sum_{e' \in V_{i,out}} |e \overset{N_i}{\bullet} \cap \overset{N_i}{\bullet}e'|$ . From the definition of partition  $V$ , we know that  $\sum_{e' \in V_{i,out}} |e \overset{N_i}{\bullet} \cap \overset{N_i}{\bullet}e'| = |V_i|$ . Furthermore,  $t \overset{N_i}{\bullet} \cap \overset{N_i}{\bullet}t' = \{p \in P \mid \beta_i(V_{i,in}) \subseteq \overset{N_i}{\bullet}p \wedge \beta_i(V_{i,out}) \subseteq p \overset{N_i}{\bullet}\}$ . Therefore we know that there are at least enough colors available for each partition  $V_i$ . The same way of reasoning can be used to show that there are at least enough colors available for each set of partitions  $v \subseteq \{V_1, \dots, V_n\}$ . (The latter requires the use of Item 8 instead of 7 of Definition 4.9). Therefore, there exists at least one condition coloring for the entire subgraph  $\delta_{in}(N_i)$ . Similarly, this can be shown for  $\delta_{out}(N_i)$ .

At this point, we have shown that we can construct condition colorings for two subgraphs of  $\delta(N_i)$ , namely  $\delta_{in}(N_i)$  and  $\delta_{out}(N_i)$ . The final part of the

proof use Item 9 of Definition 4.9, since we now have to show that *the same* condition coloring can be constructed for both subgraphs. For this purpose, we consider a segment  $(C', E_{in}, E_{out})$  in  $N_i$ . Since segments correspond to connected components of  $\delta(N_i)$ , it is sufficient to show that the same condition coloring can be constructed for  $\delta_{in}(N_i)$  and  $\delta_{out}(N_i)$ , restricted to  $C'$ , which we call  $\delta'_{in}(N_i)$  and  $\delta'_{out}(N_i)$ . From the definition of a segment, it is clear that this restriction does not disturb the structure of  $\delta_{in}(N_i)$  and  $\delta_{out}(N_i)$ , i.e. in both graphs, each connected component is still a complete subgraph. Now consider a possible condition coloring on  $\delta'_{in}(N_i)$ . Each color given to a condition in that graph refers to a place in the causal net. However, multiple conditions can be mapped onto each place, namely one condition for each token that was produced in that place by an element of  $E_{in}$ . The same holds for  $\delta'_{out}(N_i)$ , i.e. multiple condition can be mapped onto each place, namely one condition for each token that was consumed by a succeeding element of  $E_{out}$ . Since Item 9 of Definition 4.9 states that the tokens produced by  $E_{in}$  are the tokens consumed by  $E_{out}$ , it must be possible to construct the same condition coloring  $\alpha_i$  for both  $\delta'_{in}(N_i)$  and  $\delta'_{out}(N_i)$ . Using Lemma 3.5, we then know that this coloring  $\alpha_i$  is a condition coloring on  $\delta(N_i)'$ , i.e. the restriction of  $\delta(N_i)$  to  $C'$ .

Since we can now provide a condition coloring on each connected component of  $\delta(N_i)$ , we have shown that we can construct a condition coloring on the entire graph  $\delta(N_i)$ .  $\square$

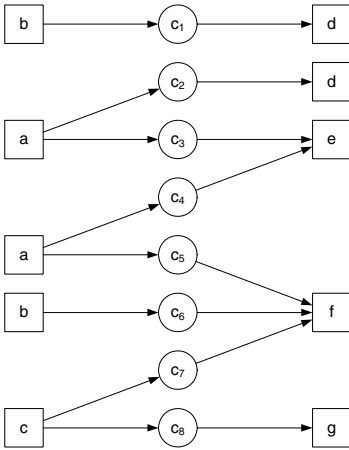
To clarify the rather complex proof of Lemma 5.8 we use an example. Consider a causal net containing the fragment of a WF-net presented in Figure 14. We numbered the conditions 1 through 8 to be able to distinguish them. Now, assume that the two Petri nets presented in Figure 15 are parts of two alternative system nets appearing in the aggregation class of that causal net.

The proof of Lemma 5.8 depends on the condition graph of a run. Therefore, in Figure 16 we present the condition graph of the run presented in Figure 14. Note that we labeled the edges to show from which event the edge was derived.

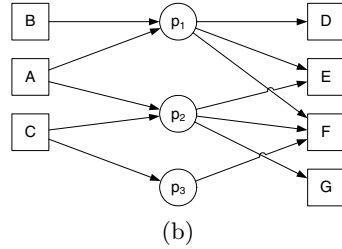
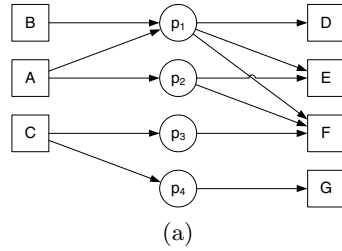
In Lemma 5.8, the condition graph of Figure 16 (i.e.  $\delta(N_i)$  in the lemma) is split up into two subgraphs, namely one for the input side of events (i.e.  $\delta_{in}(N_i)$ , see Figure 17) and one for the output sides of events (i.e.  $\delta_{out}(N_i)$ , see Figure 18).

Then the proof continues, by showing that for each of these two subgraphs it is possible to provide a condition coloring. Figure 19 shows the possible labels for each subgraph and both Petri nets from Figure 15. It is easy to see that this indeed leads to several possible colorings in each graph.

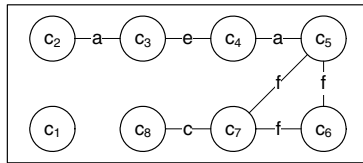
At this point it is proven that it is always possible to construct two coloring functions on the input and output subgraph that give the same label to each condition in both graphs. If we look at Figure 19 and we take the input subgraph shown in Figure 15(a) (i.e. the left-top figure) then it is easy to see that it is possible to label  $c_4$  with  $p_2$  and  $c_5$  with  $p_1$ . This however is not possible in the output subgraph, since neighbor  $c_6$  has to be mapped onto  $p_1$ . Instead, there is only one mapping that is the same for both subgraphs. The last part of the proof uses the fact that for each segment the input enables the output. This implies that the token that is placed in  $p_1$  has to be consumed from there again.



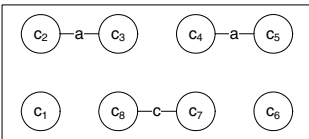
**Fig. 14.** A part of a run containing two segments



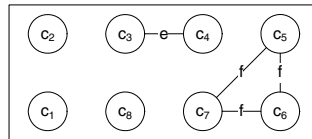
**Fig. 15.** Two parts of system nets in the aggregation class of Figure 14



**Fig. 16.** Part of the condition graph of the run of Figure 14



**Fig. 17.** Input subgraph of Figure 16



**Fig. 18.** Output subgraph of Figure 16

Therefore, if we would label  $c_5$  with  $p_1$  then this would be the same as saying that transition  $A$  produces a token in  $p_1$  which is consumed by transition  $F$  again. However, transition  $F$  also consumes another token from  $p_1$ , namely the one corresponding to  $c_6$ , i.e. coming from transition  $B$ . This violates the fact that only one edge can exist between a place and a transition.

Figure 20 shows the only possible condition coloring of the condition graph of Figure 16, using the labels provided by the system net of Figure 15(a) and Figure 21 shows the only possible condition coloring of the condition graph of Figure 16, using the labels provided by the system net 15(b). Note that in general additional condition colorings may be possible.

From Figures 20 and 21, we can conclude that both system nets depicted in Figure 15 are indeed capable of producing the causal net of Figure 14, since we can construct a condition coloring on the condition graphs.

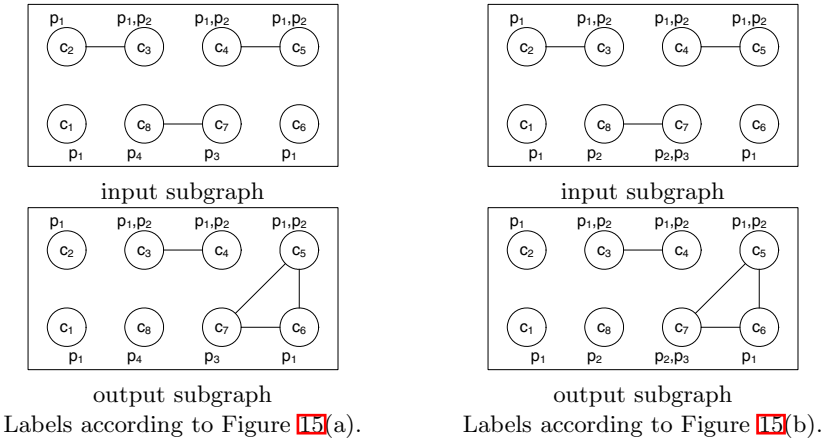
What remains to be shown is that the condition coloring also fulfills the last part of the definition of a run, namely the demand with respect to the initial marking. Furthermore, we conclude that at least three places are needed in the system net and that, for example, the place between  $C$  and  $G$  could also be  $p_1$ .

**Lemma 5.9. (Initial marking can be mapped)**

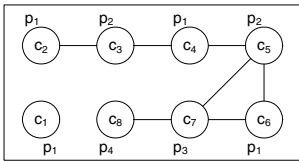
Let  $\Phi$  be a causal set, let  $\mathcal{A}_\Phi$  be the aggregation class of  $\Phi$  and let  $(N, \mathcal{B}) \in \mathcal{A}_\Phi$  with  $N = (P, T, F, M_0)$ . Let  $N_i = (C_i, E_i, K_i, S_i) \in \Phi$  be a causal net and  $\Delta_{N_i} = (C_i, A)$  be the condition graph of  $N_i$ . Let  $\alpha_i : C_i \rightarrow P$ , such that  $\alpha_i$  is a condition coloring of  $\Delta_{N_i}$ . We show that  $\alpha_i(S_i) = M_0$ .

*Proof.* From Definition 4.9, we know that  $M_0 = [p_{ini}]$ . Furthermore, from Definition 4.3, we know that there is exactly one  $c \in C_i$  with  $S(c) = 1$ . Moreover, using Lemma 5.8, we conclude that  $\alpha_i(c) = p_{ini}$  and thus  $\alpha_i(S_i) = [p_{ini}] = M_0$ . □

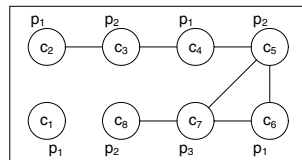
Finally, we can combine everything and state that each WF-net in an aggregation class is indeed a system net of a causal set.



**Fig. 19.** Possible condition colorings for the subgraphs of figures 17 and 18



**Fig. 20.** The condition coloring of Figure 16 according to Figure 15(a)



**Fig. 21.** The condition coloring of Figure 16 according to Figure 15(b)

**Property 5.10. (Aggregation class only contains system nets)**

Let  $\Phi$  be a causal set, let  $\mathcal{A}_\Phi$  be the aggregation class of  $\Phi$  and let  $(N, \mathcal{B}) \in \mathcal{A}_\Phi$  with  $N = (P, T, F, M_0)$ . Let  $N_i = (C_i, E_i, K_i, S_i) \in \Phi$  be a causal net with event labeling function  $\beta_i \in \mathcal{B}$ , condition graph  $\Delta_{N_i} = (C_i, A)$  and  $\alpha_i : C_i \rightarrow P$  a condition coloring of  $\Delta_{N_i}$ . Then  $(N_i, \alpha_i, \beta_i)$  is a run of  $N$ .

*Proof.* This result combines Lemma 5.7 which shows that for all  $e \in E_i$ ,  $\alpha_i$  induces a bijection from  ${}^{\bullet}e$  to  ${}^{\bullet}\beta_i(e)$  and from  $e^{\bullet}$  to  $\beta_i(e)^{\bullet}$  and Lemma 5.9 which shows that  $\alpha_i(S_i) = M_0$ . □

We have shown that it is possible to take a set of causal nets and construct a system net such that each causal net is a run of that system net, as long as the causal nets have one initially marked condition. What we did not show are the conditions under which the aggregation class is not empty. These conditions however, cannot be given based on a set of causal nets. Even if these causal nets belong to one causal set, this is still not enough. What we *can* show however, is that if we start from a *sound* WF-net as a system net, generate a set of runs and remove the labels of places, the original WF-net is in the aggregation class. For the full definition of soundness, we refer to [14].

**Property 5.11. (A system net is in the aggregation class of its runs)**

Let  $N = (P, T, F, M_0)$  be a sound WF-net. We consider  $N$  to be a system net. Let  $\mathcal{B} = \{(N_i, \alpha_i, \beta_i) \mid 0 \leq i < n\}$  be the causal behavior of that system net, such that each  $(N_i, \alpha_i, \beta_i)$  is a run of that system net, with  $N_i = (C_i, E_i, K_i, S_i)$ . Let  $\mathcal{B} = \{\beta_i \mid 0 \leq i < n\}$  and  $\Phi = \{N_i \mid 0 \leq i < n\}$  be a causal set. We show that  $(N, \mathcal{B}) \in \mathcal{A}_\Phi$ .

*Proof.* We show that all conditions of Definition 4.9 are satisfied.

1.  $T = \bigcup_{0 \leq i < n} \text{rng}(\beta_i)$  is the set of transitions. Since the WF-net is sound, there are no dead transitions thus implying that in its causal set each transition appears as an event at least once.
2. For all  $p \in P$  holds that  ${}^{\bullet}p \cup p^{\bullet} \neq \emptyset$ . Since every sound WF-net is connected, this condition is satisfied,
3.  $M_0 = [p_{ini}]$  and  ${}^{\bullet}p_{ini} = \emptyset$ . Since  $N$  is a WF-net, there is exactly one place  $p_{ini} \in P$ , such that  ${}^{\bullet}p_{ini} = \emptyset$  and  $M_0 = [p_{ini}]$ ,
4.  $\mathcal{B}$  is the set of all labeling functions, i.e.  $\mathcal{B} = \{\beta_i \mid 0 \leq i < n\}$ .
5. For each causal net  $N_i$ , with  $e \in E_i$  and  $\beta_i(e) = t$  and  ${}^{\bullet}e = \{c\}$ , holds that if  $S_i(c) = 1$  then  $p_{ini} \in {}^{\bullet}t$ . Since  $S_i(c) = 1$ , we know that  ${}^{\bullet}c = \emptyset$ . Now assume  $\alpha_i(c) = p$ . The fact that for all  $e' \in E_i$ ,  $\alpha_i$  induces local bijections from  $e'^{\bullet}$  to  $\beta_i(e')^{\bullet}$  implies that  ${}^{\bullet}p = \emptyset$  and since  $N$  is a workflow net, this implies that  $p = p_{ini}$ . Moreover, the fact that for all  $\alpha_i$  induces local bijections from  ${}^{\bullet}e$  to  ${}^{\bullet}t$  implies that  $p_{ini} \in {}^{\bullet}t$ ,
6. For each causal net  $N_i$ , with  $e \in E_i$  and  $\beta_i(e) = t$  holds that  $|t^{\bullet}| = |e^{\bullet}|$  and  ${}^{\bullet}t = |{}^{\bullet}e|$ . Since  $\alpha_i$  induces bijections from  $e^{\bullet}$  to  $t^{\bullet}$  and from  ${}^{\bullet}e$  to  ${}^{\bullet}t$ , this condition is satisfied,

7. For each causal net  $N_i$ , with  $e \in E_i$ ,  $\beta_i(e) = t$  and  $T' \subseteq T$  holds that  $|t \bullet \cap \bigcup_{t' \in T'} (\bullet t')| \geq \sum_{e' \in E_i, \beta_i(e') \in T'} |e \bullet \cap \bullet e'|$ . Let  $e \in E_i$  with  $\beta_i(e) = t$  and let  $T' \subseteq T$ . Assume that there  $|t \bullet \cap \bigcup_{t' \in T'} (\bullet t')| = m$ , i.e. there are  $m$  places between  $t$  and  $T'$ . Furthermore, assume that  $\sum_{e' \in E_i, \beta_i(e') \in T'} |e \bullet \cap \bullet e'| < m$ . Since for all  $e' \in E_i$  with  $\beta_i(e') = t_i$ ,  $\alpha_i$  induces local bijections from  $\bullet e_i$  to  $\bullet t_i$ , we know that there are at least two  $c_1, c_2 \in e \bullet$  that are mapped onto the same  $p \in P$ . However, since  $p \in t \bullet$  this violates the local bijection property of  $\alpha_i$ ,
8. For each causal net  $N_i$ , with  $e \in E_i$ ,  $\beta_i(e) = t$  and  $T' \subseteq T$  holds that  $|\bigcup_{t' \in T'} (t' \bullet) \cap \bullet t| \geq \sum_{e' \in E_i, \beta_i(e') \in T'} |e' \bullet \cap \bullet e|$ . The proof for this property is similar to the previous one.
9. For each causal net  $N_i$  and any segment  $(C'_i, E_{in}, E_{out})$  of  $N_i$  holds that  $\biguplus_{e \in E_{in}} (\beta_i(e) \bullet) = \biguplus_{e \in E_{out}} (\bullet \beta_i(e))$ . This property relates to soundness. If one set of transitions produces tokens then these tokens will be consumed by another set of transitions (i.e. no tokens are “left behind” in the execution of a sound WF-net). The only exception is the transition that produces a token in the output place, but that transition cannot produce any tokens in any other place. Therefore, in each run, the input events of a segment will enable the output events of that segment.

□

The NLC algorithm takes a set of causal nets without condition labels as a starting point. From these nets, an aggregation class of WF-nets is defined. In this section, we have formally proven that every element of the aggregation class indeed is capable of constructing the causal nets used as input. Furthermore, if the runs were generated from some sound WF-net, then the WF-net itself is in that aggregation class.

## 6 Conclusion

In this paper, we looked at process mining from a new perspective. Instead of starting with a set of traces, we started with runs which constitute partial orders on events. We presented three algorithms to generate a Petri net from these runs. The first algorithm assumes that, for each run, all labels of both conditions and events are known. The second algorithm relaxes this by assuming that some transitions can have the same label (i.e. duplicate labels are allowed in the system net). This algorithm can also be used if only condition/place-labels were recorded. Finally, we provided an algorithm that does not require condition labels, i.e. the event/transition labels are known, the condition/place labels are unknown and duplicate transition labels are not allowed.

The results presented in this paper hold for a subclass of Petri nets, the so-called WF-nets. However, the first two algorithms presented here can easily be generalized to be applicable to any Petri net. For the third algorithm this can also be done, however, explicit knowledge about how the initial markings of

various runs relate is needed. When taking a set of runs as a starting point, this knowledge is not present in the general case.

## References

1. van der Aalst, W.M.P.: Workflow Verification: Finding Control-Flow Errors Using Petri-Net-Based Techniques. In: van der Aalst, W.M.P., Desel, J., Oberweis, A. (eds.) *Business Process Management*. LNCS, vol. 1806, pp. 161–183. Springer, Heidelberg (2000)
2. van der Aalst, W.M.P.: *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer (2011)
3. van der Aalst, W.M.P., Alves de Medeiros, A.K., Weijters, A.J.M.M.T.: Genetic Process Mining. In: Ciardo, G., Darondeau, P. (eds.) *ICATPN 2005*. LNCS, vol. 3536, pp. 48–69. Springer, Heidelberg (2005)
4. van der Aalst, W.M.P., van Hee, K.M., ter Hofstede, A.H.M., Sidorova, N., Verbeek, H.M.W., Voorhoeve, M., Wynn, M.T.: Soundness of Workflow Nets: Classification, Decidability, and Analysis. *Formal Aspects of Computing* 23(3), 333–363 (2011)
5. van der Aalst, W.M.P., Reijers, H.A., Weijters, A.J.M.M., van Dongen, B.F., Alves de Medeiros, A.K., Song, M., Verbeek, H.M.W.: *Business Process Mining: An Industrial Application*. *Information Systems* 32(5), 713–732 (2007)
6. van der Aalst, W.M.P., van Dongen, B.F., Herbst, J., Maruster, L., Schimm, G., Weijters, A.J.M.M.: Workflow Mining: A Survey of Issues and Approaches. *Data and Knowledge Engineering* 47(2), 237–267 (2003)
7. van der Aalst, W.M.P., Weijters, A.J.M.M., Maruster, L.: Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering* 16(9), 1128–1142 (2004)
8. Agrawal, R., Gunopulos, D., Leymann, F.: Mining Process Models from Workflow Logs. In: Schek, H.-J., Saltor, F., Ramos, I., Alonso, G. (eds.) *EDBT 1998*. LNCS, vol. 1377, pp. 469–483. Springer, Heidelberg (1998)
9. Badouel, E., Darondeau, P.: Theory of Regions. In: Reisig, W., Rozenberg, G. (eds.) *APN 1998*. LNCS, vol. 1491, pp. 529–586. Springer, Heidelberg (1998)
10. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Synthesis of Petri Nets from Finite Partial Languages. *Fundamenta Informatica* 88(1), 437–468 (2008)
11. Bergenthum, R., Desel, J., Mauser, S., Lorenz, R.: Synthesis of Petri Nets from Term Based Representations of Infinite Partial Languages. *Fundamenta Informatica* 95(1), 187–217 (2009)
12. Cook, J.E., Wolf, A.L.: Discovering Models of Software Processes from Event-Based Data. *ACM Transactions on Software Engineering and Methodology* 7(3), 215–249 (1998)
13. Datta, A.: Automating the Discovery of As-Is Business Process Models: Probabilistic and Algorithmic Approaches. *Information Systems Research* 9(3), 275–301 (1998)
14. Desel, J.: Validation of Process Models by Construction of Process Nets. In: van der Aalst, W.M.P., Desel, J., Oberweis, A. (eds.) *Business Process Management*. LNCS, vol. 1806, pp. 110–128. Springer, Heidelberg (2000)
15. Desel, J., Erwin, T.: Hybrid specifications: looking at workflows from a run-time perspective. *Computer Systems Science and Engineering* 5, 291–302 (2000)
16. Desel, J., Reisig, W.: The synthesis problem of Petri nets. *Acta Informatica* 33, 297–315 (1996)



17. van Dongen, B.F., van der Aalst, W.M.P.: Multi-phase Process Mining: Building Instance Graphs. In: Atzeni, P., Chu, W., Lu, H., Zhou, S., Ling, T.-W. (eds.) ER 2004. LNCS, vol. 3288, pp. 362–376. Springer, Heidelberg (2004)
18. van Dongen, B.F., van der Aalst, W.M.P.: Multi-Phase Process Mining: Aggregating Instance Graphs into EPCs and Petri Nets. In: PNCWB 2005 Workshop, pp. 35–58 (2005)
19. Ehrenfeucht, A., Rozenberg, G.: Partial (Set) 2-Structures - Part 1 and Part 2. *Acta Informatica* 27(4), 315–368 (1989)
20. Greco, G., Guzzo, A., Pontieri, L., Saccà, D.: Discovering Expressive Process Models by Clustering Log Traces. *IEEE Transaction on Knowledge and Data Engineering* 18(8), 1010–1027 (2006)
21. Harel, D., Kugler, H.-J., Pnueli, A.: Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements. In: Kreowski, H.-J., Montanari, U., Yu, Y., Rozenberg, G., Taentzer, G. (eds.) *Formal Methods (Ehrig Festschrift)*. LNCS, vol. 3393, pp. 309–324. Springer, Heidelberg (2005)
22. Herbst, J.: A Machine Learning Approach to Workflow Management. In: Lopez de Mantaras, R., Plaza, E. (eds.) *ECML 2000*. LNCS (LNAI), vol. 1810, pp. 183–194. Springer, Heidelberg (2000)
23. Herbst, J., Karagiannis, D.: Workflow mining with InWoLvE. *Computers in Industry* 53(3), 245–264 (2004)
24. Lorenz, R., Juhás, G.: Towards Synthesis of Petri Nets from Scenarios. In: Donatelli, S., Thiagarajan, P.S. (eds.) *ICATPN 2006*. LNCS, vol. 4024, pp. 302–321. Springer, Heidelberg (2006)
25. Alves de Medeiros, A.K., Weijters, A.J.M.M., van der Aalst, W.M.P.: Genetic Process Mining: An Experimental Evaluation. *Data Mining and Knowledge Discovery* 14(2), 245–304 (2007)
26. Roychoudhury, A., Thiagarajan, P.S.: Communicating Transaction Processes. In: Lilius, J., Balarin, F., Machado, R. (eds.) *Proceedings of Third International Conference on Application of Concurrency to System Design (ACSD 2003)*, pp. 157–166. IEEE Computer Society (2003)
27. Smith, E.: On Net Systems Generated by Process Foldings. In: Rozenberg, G. (ed.) *APN 1991*. LNCS, vol. 524, pp. 253–276. Springer, Heidelberg (1991)
28. Weijters, A.J.M.M., van der Aalst, W.M.P.: Rediscovering Workflow Models from Event-Based Data using Little Thumb. *Integrated Computer-Aided Engineering* 10(2), 151–162 (2003)
29. van der Werf, J.M.E.M., van Dongen, B.F., Hurkens, C.A.J., Serebrenik, A.: Process Discovery using Integer Linear Programming. *Fundamenta Informaticae* 94, 387–412 (2010)

# Author Index

- Aalst, Wil M.P. van der 334  
Arnold, Sonya 226
- Ben Salem, Ala-Eddine 94  
Billington, Jonathan 226  
Bruneo, Dario 277  
Buchs, Didier 123, 169
- Carmona, Josep 1  
Choppy, Christine 197  
Colange, Maximilien 169
- Darondeau, Philippe 24  
Dedova, Anna 197  
Desel, Jörg 334  
Dongen, Boudewijn F. van 334  
Duret-Lutz, Alexandre 94
- Evangelista, Sami 169, 197
- Foo, Ernest 251
- Hillah, Lom-Messan 46  
Hostettler, Steve 123
- Kindler, Ekkart 71  
Klaï, Kaïs 197  
Kordon, Fabrice 46, 94, 169
- Lakos, Charles 46  
Lampka, Kai 169  
Linard, Alban 123, 169
- Lohmann, Niels 169  
Longo, Francesco 277  
López Bóbeda, Edmundo 123
- Marechal, Alexis 123  
Marrone, Stefano 308  
Mazzocca, Nicola 308
- Nardone, Roberto 308
- Ouyang, Chun 251
- Paviot-Adet, Emmanuel 169  
Petrucci, Laure 46, 197  
Presta, Roberta 308  
Puliafito, Antonio 277
- Ricker, Laurie 24  
Romano, Simon Pietro 308
- Sedlmajer, Nicolas 123  
Suriadi, Suriadi 251
- Thierry-Mieg, Yann 169
- Vittorini, Valeria 308
- Westergaard, Michael 146  
Wimmel, Harro 169
- Youcef, Samir 197